DOE/ER/25009--T7

DE92 002761

Parallel Computing Works

Caltech Concurrent Computation Program
Pasadena, California 91125

October 23, 1991

DISCLAIMER

Contents

Chapter 1

Introduction

1.1 Federal High Performance Computing and Communication Initiative (FHPCCI) — The Vision (of book and national enterprise)

*** Contribution needed from G. C. Fox ***

1.2 C^3P History and How it Illustrates Some Aspects of FHPCCI

*** Contribution needed from G. C. Fox ***

1.3 The Outsider Point of View

1.3.1 Introduction

This chapter will discuss the history of the high performance computing activity as observed in the Applied Mathematical Sciences (AMS) research program in the Office of Energy Research (OER) in the Department of Energy (DOE) from Fiscal Years 1981 through 1990—the period of the author's tenure as manager of that program. Other programs will be mentioned, but there is no attempt here to cover the history of other agencies completely.

Here I quote from the author's description of the DOE/AMS research program in high performance systems on page seven of "Summaries of the

FY 1981 Applied Mathematical Sciences Research Program", DOE/ER-0115, December, 1981.

"High Performance Systems responds to the fact that computer hardware development has reached a fundamental limit imposed by finite signal propagation speeds. While new technologies may alleviate heat generation in circuitry and therefore permit more compact and thus faster systems, increased computing power will ultimately depend upon parallel execution of instructions. High Performance Systems, therefore, emphasizes two critical areas: the match between algorithmic structure of large-scale computational models and the possible architectural structure of future supercomputers, and the transportability of applications software between successive generations of supercomputers. Several activities in Advanced Computer Concepts are pursued jointly with the NSF's Computer Science Research Program and the DOD DARPA's Information Processing Techniques Research Program. For example, NSF is supporting investigations of fundamental changes in computer architectures and DARPA is supporting related implications of VLSI technology, while (sic) DOE supports their impact on the solution of partial differential equations describing energy systems on the supercomputers of the future.

At the level of electronic components, the art of computer building is advancing very rapidly. Single, fingernail-sized electronic 'chips' are already as powerful as yesterday's minicomputers, and by the end of this decade very small computers, each as powerful as some of today's largest, should become available. This technological progress is bound to trigger major changes in the way computer systems are designed and used. A particularly exciting possibility is to combine thousands or tens of thousands of these powerful one-chip computers into 'ultracomputers' capable of computing hundreds of times faster than is now feasible. This would allow computational study of many important chemical, meteorological, physical, and biological models that presently lie out of reach ... "

This program description, written nearly a decade earlier, was not so prophetic as it may seem here, for many people in the computational science research field believed these words, although many people may have the impression that these ideas are brand new. Here is the story of how the DOE/AMS program was managed during that decade and some of the outcome of that research.

1.3.2 Progress in the Department of Energy

During the period from FY 1981 through FY 1984, pressure was kept to a maximum to populate the research fields concerned with supercomputing (mainly computational physics based research) and with parallel computing (mainly the DOE/AMS research but also in High Energy Physics projects in quantum chromodynamics). In 1981, a "white paper" was written for the Director of Energy Research on the need for supercomputers in the Office of Energy Research (OER). OER includes not only the Office of Fusion Energy (OFE), which had established the National Magnetic Fusion Energy Computing Center at Lawrence Livermore National Laboratory in 1976, but also the Office of High Energy and Nuclear Physics, the Office of Health and Environmental Research, and the Office of Basic Energy Sciences.

In 1982, many Federal research managers noted that several documents on the importance of supercomputing had been written by specific disciplines: oceanography, nuclear physics, high energy physics, etc. None of these documents had the intended effect: to provide access to computational scientists in these disciplines. A group of managers collaborated on a document that became known as the "Lax Report", (after Professor Peter Lax, who drove the group relentlessly while managing a cat nap now and then). The Lax Report was multidisciplinary and was endorsed by nearly all concerned with supercomputing.

The next milestone in the saga is the allocation of 5% of the two CRAY 1 computers at the NMFECC to the "rest" of OER (*i.e.,* OFE kept 95%). This bold step provided an annual total of 800 CRAY 1 equivalent hours to approximately 33% of the researchers in the United States supported by Federal funds! Not long after, the NSF established its Phase I, and then Phase II centers that changed the availability of supercomputer cycles for U. S. researchers. In 1990, the name of the DOE center was changed to the National Energy Research Supercomputer Center (NERSC). The NERSC operates one CRAY X-MP-2/2, a CRAY 2-4/64, a CRAY 2-4/256, and the unique CRAY-28/64; the sixteen CRAY 2 processors provide over 244,000 CRAY 1 equivalent hours to ER researchers and this year OFE uses less than 50% of those hours. These numbers are incorporated here to indicate the interest in research in computational physics, chemistry, biology, geosciences, and mathematics spawned in the sphere of influence of the DOE. And yet, according to the passage quoted from the FY 1981 AMS book, the program manager believed then that massive parallelism would be inevitable.

This much activity prompted the Director of Energy Research to form

a new staff office reporting directly to the Director (a position in the DOE equivalent to an Assistant Secretary—*i.e.*, the Director of OER reports to the Deputy Secretary as do the Assistant Secretaries). The new position was called the Scientific Computing Staff (SCS) and this staff office was assigned two activities to manage: the basic research in mathematical sciences under the AMS program and the National Magnetic Fusion Energy Computing Center (now NERSC). The first year of this program, FY 1985, the basic budget for research went from $14 million in FY 1984 to $20 million, and the budget for managing the non-OFE funding portion of the NMFECC was $7 million (for the purchase of the CRAY X-MP and other functions). To the latter portion of the budget, congressional staff of the House Appropriations Committee, in concert with Representative Don Fuqua of Florida and his staff, added an additional $7 million to establish a supercomputer center at Florida State University.

1.3.3 The Start of High Performance Computing Research

The addition of $5 million to the AMS research budget enabled expansion of the idea of exploring appropriate parallel computer architectures. Twenty such research projects were considered—some with formal proposals, some with preproposals, some with site visits. Here is a list of the projects considered and the rules for the reviewers.

This review panel is to evaluate the scientific merit, level of effort, and management plans for several ongoing multiprocessor architecture projects. Technical evaluation of the underlying architectural ideas has already taken place. The task here is to determine at what level these projects should be continued in the future. Here are some of the issues to be addressed:

1. Is this project relevant to large scale scientific computation problems found in DOE mission programs?

2. What level and type of staff will be required to complete the proposed project in the proposed time?

3. What components of the project address the areas of system design, such as register transfer level, circuit level, interconnection techniques, processor design or acquisition, power and cooling requirements (packaging), input/output system (peripherals in general)? Are there adequate facilities available at the site (or off-site) at a reasonable cost?

4. How will the development of operating systems (including utilities, compilers, and other system software) proceed?

5. What classes of problems are intended for demonstration applications on this project (e.g., Monte Carlo, PDE's, logic programming, etc.)? Are the appropriate staff available and provided for?

6. What peculiarities will a given architecture have for issues such as numerical analysis, stability, fault tolerance, reproducibility, etc., and how are these issues addressed?

7. What are the provisions for collaboration with industry; with DOE laboratories; with similar projects at other universities?

8. Is there a rationale for multiple agency funding? Should one agency be a lead supporter?

9. Would it be useful to combine two or more of the projects for some components (say development of the operating system, math libraries, etc.).

December 11, 1984

Summary of Parallel Architecture Projects Review

Background

During FY 1983 and FY 1984, a wide variety of parallel architecture projects were considered for support under the FY 1985 Applied Mathematical Sciences research program. Preproposals, proposals, and presentations were received from the following organizations:

Site	Name of PI	Project
Argonne National Lab.	Hagstrom	Lemur
Los Alamos National Lab.	Buzbee	Pups
Lawrence Berkeley Lab.	Maples	Midas
Lawrence Livermore National Lab.	Woods	S1
Caltech	Fox	Cosmic Cube
CalState	Burger	3D dataflow
Columbia	Shaw	Non-Von
New York University	Kalos	Ultracomputer
University of Illinois	Kuck	Cedar
Yale	Schultz	ELI Circus
MIT	Dennis	Dataflow
NCSU	Agrawal	MCS
Princeton	Garcia-Molinas	Massive Memory Purdue
Rice	Pringle	
University of Texas	Browne	TRAC
BDM	Pritchard	Coarse grain dataflow

Other projects discussed informally included:

Site	Name of PI	Project
FNAL	Nash	HEP data analysis
SLAC	Kunz	3081E and 168E
Columbia	Christ	QCD machine
Cornell	Wilson	Gibbs project+
University of Maryland	Stewart	ZMOB
NASA Langley		MPP and FEM

Site visits were made by [Don] Austin, [Ed] Oliver, and in some cases, [Jim] Decker to the following sites: ANL (2), LANL (2), LBL (2), LLNL (2), Caltech (1), NYU (2), U. Illinois (1), Yale (1), MIT (2), U. Texas (1).

Formal proposals were received from LBL, Caltech, NYU, U. Illinois, Yale, MIT, Purdue, and BDM for building large scale parallel experimental machines. Design and algorithm studies were already supported by the AMS program FY 1983 (and before) at LANL, Caltech, NYU, U. Illinois, MIT, and U. Texas. LANL and U. Texas decided not to pursue further implementation of their machines. The Purdue proposal was withdrawn. The Yale and BDM proposals were submitted too late to be considered in this review.

The status of the projects already supported by AMS at the time of the management review (discussed below) was as follows:

Caltech	64 processor hypercube based on Intel 8086/8087, Unix
NYU	8 processor ultracomputer based on Motorola 68010, Unix
MIT	8 processor dataflow based on AMD 2901, PDP 11 front end
U. Texas	4 processor TRAC based on AMD 2901, in house OS
U. Illinois	no machine, but very advanced restructuring compiler
LANL	4 processor Pups based on Intel 8086/8087, Intel OS

Other projects not supported by the AMS program include:

LBL	11 processor Midas based on Modcomp, Modcomp OS
ANL	8 processor Lemur based on NS 16032, Unix based

The AMS program considered for support projects that included a wide spectrum of disciplines covering numerical algorithms, numerical analysis, computational science, languages, operating systems, and architecture experts. Projects focused on special purpose applications, such as QCD or Ising machines, physics data processing machines, and artificial intelligence applications were not considered. The LBL Midas project, originally supported by Nuclear Physics for data analysis, was considered for support based on a formal proposal submitted.

1.3.4 Management Review

In October, 1984, a management review panel was convened to review four current projects requesting long range support for implementing experimental parallel architectures and the appropriate research activities that accompany such activity. In addition, the LBL Midas project was also considered. The panel consisted of program managers from Federal agencies that support research projects of this nature: DOE, DARPA, NSF, NASA, NSA, ONR, and NBS. The panel members all have direct experience with reviewing and managing such projects, and nearly all had seen presentations of all five projects at various workshops and conferences.

The evaluation criteria were:

1. Relevance to DOE basic research program in large scale parallel processor computers for scientific computation;

2. Adequacy of proposed staffing plans for:

   (a) algorithm development for computational science;

   (b) operating systems and utilities;

   (c) system design (fault tolerance, custom components, etc.);

   (d) system fabrication and testing;

   (e) system engineering (building and debugging);

   (f) system operation (local and remote access).

3. Plans for collaboration with computing industry and national labs;

4. Overall scientific quality, originality, and uniqueness.

Proposals were circulated to the panel shortly before the review and each presenter was allowed one hour to address the above points. The projects were scored on a 10 point system.

Here are the averaged scores from five reviewers (non-DOE) that provided written comments:

	Caltech	NYU	MIT	U. Ill.	LBL
1.	8.6	9.4	8.0	8.4	4.3
2.	8.3	8.9	4.3	7.3	3.8
3.	7.6	7.8	4.6	5.8	4.3
4.	7.4	8.8	7.6	7.1	3.0
Avg.	8.0	8.7	6.1	7.2	3.9

In the oral discussions among the ten panel members following the presentations, the general feeling was that the Caltech group was off to a very strong start and that funding by multiple agencies (DOE, DARPA, and possibly NSA) in conjunction with JPL, gave a high probability of producing useful research results as described in their proposal. The NYU collaboration with IBM Yorktown Heights, again with joint funding from IBM and possibly DARPA, also provided a strong project with reachable milestones and excellent probability of producing good research results. The University of Illinois team was judged as equally strong as the first two, and since their approach was to do very little engineering and concentrate instead on sophisticated software and algorithm research, they also offered a high probability of producing good research results.

The MIT proposal was judged weak in the area of inhouse design of a special purpose processor and fabrication of special boards without adequate support staff or industry collaboration. The recommendation was to continue this project at the current level for another year, with the possibility of using the DARPA supported emulation hardware under construction at MIT for further testing of the dataflow concepts.

The LBL project was judged weakest in all four areas. The impression of the panel was that this is a special purpose machine designed explicitly for analysis of nuclear physics data. The successful use of this machine appeared to be primarily in processing such data because it includes a lot of special purpose hardware designed for that purpose. The consensus of the panel was that the project should continue to be supported by the physics community that has benefited most from its construction.

Donald M. Austin, ER-7
Applied Mathematical Sciences
Scientific Computing Staff
Office of Energy Research

1.3.5 The Caltech Concurrent Computing Program

So one of the three major projects selected was the C^3P project. Here is how this five year grant was justified to the DOE management.

Date: June 5, 1985

From: ER-7

Subject: Award of a New Grant to California Institute of Technology, entitled "The Caltech Concurrent Computation Program", Geoffrey Fox, P.I.

To: James F. Decker, ER-7

I recommend that this grant be initiated for a period of five years beginning August 1, 1985, through July 31, 1990, with $6,800,000 of operating funds.

Amount of Funding	Funding Period
$1,400,000	FY 1985
1,400,000	FY 1986
2,000,000	FY 1987
1,000,000	FY 1988
1,000,000	FY 1989
$6,800,000	Total

Scope: This grant will provide support for faculty, research staff and graduate students working on energy related computational science topics in the areas of algorithms, languages, operating systems, and mathematical software for parallel architectures. A portion of these funds may be made available on subcontract to JPL or other suitable subcontractor for work resulting in the delivery of protytype research computers as specified by the P.I. in design and requirements documentation.

1.3.6 Programmatic Justification

In accordance with the ER Special Research Grant Program Rule (10 CFR Part 605), the subject award is based upon the following evaluation.

— Overall Scientific and Technical Merit of the Project

This new project, called the Caltech Concurrent Computation Project (C^3P) is a new approach to homogeneous architectures begun in the Caltech Cosmic Cube Project. An important lesson learned from that project is the importance of having a significant amount of computational power in each of the nodes (including a large enough memory local to the node to contain a significant computational problem) and the effectiveness of offloading the communication processes to coprocessors, including broadcast capabilities. The new nodes proposed, built from off the shelf components available from the vendors, will have these important features. In particular, the independent communication coprocessors should be capable of demonstrating the scalability of the hypercube architecture to much larger machines and also the feasibility of using heterogeneous processors in a hypercube configuration. They should be able to demonstrate true MIMD capability with this new processor node—a difficult thing to do on the current cosmic cube.

The C^3P group is widely acknowledged to be one of the most advanced groups working on problem decomposition and algorithm research for parallel architecture. They have published a large number of papers in both refereed journals and scientific magazines. The broad spectrum of scientific disciplines represented in this project is extraordinary (possibly unique). They have been successful in demonstrating techniques for mapping a long series of computational methods onto the hypercube architecture. Given the importance to DOE research and development programs of future parallel architectures for supercomputers, the Caltech project has pioneered a novel method for analyzing the structure of scientific models and devising new (and revising old) parallel algorithms to maximize the efficient use of these new machines.

—Relevance of the Objectives to DOE's Applied Mathematical Sciences Subprogram

The AMS subprogram has recently focused on understanding the basic principles involved in computational modeling on parallel architectures. New supercomputers with up to 16 processors sharing a large fast memory are already being ordered by the national laboratories. Research in large grained parallelism appropriate for these machines is already providing useful results. However, it is clear that in the long term, hundreds or thousands of processors will be required to supply the orders of magnitude increase in computing power needed to solve high dimensional problems modeling realistic physical systems. The Caltech project is proposing to investigate issues of fine grained parallelism relevant for future supercomputers. By addressing all issues involved in solving scientific problems, this project should provide valuable insight into computational science for large-scale parallel architectures.

— Appropriateness of the Proposed Method or Approach

This proposal addresses the problems associated with decomposing computational models into a large number of smaller models that can be processed independently for some time before intermediate results need to be communicated amongst the processors. Their approach is to analyze candidate algorithms for efficiency in terms of the communication overhead, redundant operations, duplication of storage and any other parameters required to describe synchronization protocols. Because the hypercube architecture is eminently scalable, results demonstrated with small problems on small (32–128 processors) hypercubes can be extrapolated to analyze re-

quirements for large problems. These methods show great promise for future supercomputing.

— Competence and Experience and Known Past Performance of the Applicant, Principal Investigator and/or Key Personnel

The P.I. is Dean of Educational Computing and a full professor in the Physics Department. He was co-P.I. on the Caltech Cosmic Cube project and primarily responsible for the success of that project. His collaborators include Professor Aron Kuppermann of the Chemistry Department, Professor Robert Clayton of Geophysics, and Professor Herb Keller of the Applied Mathematics Department, among others. These faculty and their postdoctoral and graduate students, represent a most competent and experienced research group—possibly the best of its kind. Their expertise relevant to this project is in the common computational methods used in exploring the various disciplines.

— Evaluation and Review

Professor Fox's C^3P was one of five projects covered in the management review held in October 1984. The review team consisted of 10 program managers from all Federal agencies involved in funding such research projects: DARPA, DOC (NBS), DOE, NASA, NSA, NSF. The project rated very high and the panel recommended it be considered for funding.

A site visit was made in May 1985, by a subgroup of the review team consisting of Donald Austin, Jim Greenberg, Ed Oliver of DOE's Scientific Computing Staff, John Riganati of NBS's Institute for Computer Science and Technology, and Paul Schneck, Director of NSA's Supercomputing Research Institute. There were presentations by Caltech scientists and JPL engineers. The review team later met at DOE to discuss the review and make recommendations.

The recommendations were, for practical purposes, unanimous: watch the JPL effort very closely, insist on a design review and several milestones from the JPL effort; proceed with the Caltech portion as proposed to the extent that funds are available.

In telephone conversations with Ed Stone, Division Head for Physics, Mathematics, and Astronomy, and with Professor Fox confirmed their intention to proceed exactly as recommended.

— Adequacy of the Applicant's Facilities and Resources

The C^3P Project currently has a Mark II five-cube (32 processors), the world's first seven-cube (128 processors) and an adequate complement of

front end machines (Vaxes and Elxsi) to support the initial work on this grant. The proposal calls for the delivery of the five-cube Mark III by February 1986. This should provide an adequate resource until additional five-cubes are delivered in subsequent years. This grant should be able to provide for at least four of these modules which, when coupled with the experimental fibre optic link, will become a Mark III seven-cube. Caltech management has been responsive in offering to upgrade 2500 square feet of office and lab space for the project.

— Appropriateness and Adequacy of the Proposed Budget

The support for senior personnel is small because major support for faculty comes from their own grants. They will dedicate only about 12% of their time in supervising graduate students and postdocs working full time on this project. This seems to be an ideal way to get the interdisciplinary exchange of ideas. Of course, major support for the P.I. (35%) is included in the revised budget.

There should be enough funds in the revised budget to provide four Mark III five-cubes, which can be configured as a seven-cube. Other funding sources and/or donations of equipment by industry, may help boost this to the first eight-cube (256 processors).

Support staff to maintain the equipment and develop and maintain system software are adequately provided for.

Transactional Determination

The proposed research is of national interest, will contribute to the understanding of computational science on parallel architectures of interest to the DOE, and will help maintain a supply of trained scientists required to meet future research needs.

Accordingly, I have determined that the principal purpose of this transaction is assistance and that based upon minimal DOE involvement during performance, a Grant is the most appropriate instrument for this transaction.

Donald M. Austin, ER-7
Applied Mathematical Sciences
Scientific Computing Staff
Office of Energy Research

Thus began the support of the C^3P by the DOE/AMS program. At the conclusion of the project in July, 1990, many changes had taken place in

science, computing, algorithms, project and program management, and in people's attitudes toward computational science as a science. The equivalence of this field with theoretical and experimental science is still a matter of debate in most disciplines. One need only observe the role of large scale scientific computing facilities available for computational science to realize the great leap that has been realized over the last decade. The C^3P has played a major role in demonstrating that this is a realm of quality science and, more important, a realm with an exciting future that must do things differently to survive and flourish—*i.e.,* explore massive parallelism techniques as well as build and use the new parallel machines.

1.4 Message of Parallel Computing Works

*** Contribution needed from G. C. Fox ***

Chapter 2

Technical Backdrop

*** P. C. Messina has first draft of this from his "Capri talk" ***

2.1 Hardware

2.2 Software

2.3 Algorithms

2.4 Performance

2.5 Other Issues

Chapter 3

Problem Structure — A Framework for Quantifying "When Parallel Computing Works"

*** Contribution needed from G. C. Fox ***

Chapter 4

Synchronous Applications I

4.1 The Synchronous Problem Architecture and the First C^{3}P Applications

*** Contribution needed from G. C. Fox ***

4.2 Computational Lattice Gauge Theory — QCD

4.2.1 Introduction

Computational lattice gauge theory, in particular QCD, was one of the original motivating forces behind the original Cosmic Cube and has used much CPU time on the hypercubes at Caltech over the years. Table 4.1 lists some of the calculations performed—starting in 1982 with the original four-node 8086-based prototype, right up to the present day with the 128-node Mark IIIfp. Below, we shall discuss the development of the Caltech hypercubes and other such "QCD machines", but first we should explain why QCD calculations are important and then discuss the work done at Caltech.

Gauge theories are ubiquitous in elementary particle physics: the electromagnetic interaction between electrons and photons is described by quantum electrodynamics (QED) based on the gauge group U(1), the strong force between quarks and gluons is believed to be explained by quantum chromodynamics (QCD) based on SU(3), and there is a unified description of the weak and electromagnetic interactions in terms of the gauge group SU(2) × U(1). The strength of these interactions is measured by a coupling

constant. This coupling constant is small for QED, so very accurate analytical calculations can be performed using perturbation theory. However, for QCD, the coupling constant appears to increase with distance so that perturbative calculations are only possible at short distances. In order to solve QCD at longer distances, Wilson [Wilson:74a] introduced lattice gauge theory in which the space-time continuum is discretized to provide a cut-off that regulates ultraviolet divergences non-perturbatively. This discretization onto a lattice, which is typically hypercubic, also makes the gauge theory amenable to numerical simulation by computer.

Most of the work on lattice gauge theory has been directed towards solving lattice QCD and, thus, deriving the hadron mass spectrum from first principles. This would confirm QCD as the theory of the strong force. Other calculations have also been performed, in particular, the properties of QCD at finite temperature and/or finite baryon density have been determined. Unfortunately, in order to simulate lattice QCD on a computer, one must integrate out the quark variables (because they are fermions, *i.e.*, anticommuting elements of a Grassmann algebra rather than numbers) leaving a highly non-local fermion determinant for each flavor of quark. Physically, this determinant arises from closed quark loops. The simplest way to proceed is to ignore these quark loops and work in the so-called quenched approximation with no flavors of quark present. (This may be valid for heavy quarks.) Current state-of-the-art quenched QCD calculations are performed on lattices of size $24^3 \times 40$.

However, to investigate the physically more realistic fully interacting QCD, with the inclusion of dynamical quarks, we must go beyond the quenched approximation and tackle the problem of the fermion determinant. We shall return to this later.

4.2.2 Monte Carlo

In order to explain the computations for QCD, we use the Feynman path integral formalism. For any field theory described by a Lagrangian density $\mathcal{L}$, the dynamics of the fields $\phi(x)$ are determined through the action functional

$$S(\phi) = \int dx \mathcal{L}(\phi). \tag{4.1}$$

Authors	Project	Hypercube	Reference
Brooks, Fox, Otto, Randeria, Athas, De Benedictis, Newton, Seitz	Glueball mass (SU(2)) $4^3 \times 8$ lattice	Mark I 4 node	[Brooks:83a]
Otto, Randeria	Glueball mass, modified action $4^3 \times 8$ lattice	Mark I 4 node	[Otto:83a]
Otto, Stack	Heavy quark potential $12^3 \times 16$ lattice	Mark I 64 node	[Otto:84a]
Otto, Stolorz	Glueball mass, enhanced statistics $12^3 \times 16$ lattice	Mark I 64 node	[Otto:85b]
Patel, Otto, Gupta	Monte Carlo renormalization group, nonperturbative β-function	Mark I 64 node	[Patel:85a]
Fucito, Soloman	Chiral symmetry breaking, mass spectrum, deconfinement transition, pseudo-fermions	Mark II 64 node	[Fucito:84a], [Fucito:85b], [Fucito:85c], [Fucito:85d] [Fucito:86a]
Flower, Otto	Field distribution $10^2 \times 12^2$ lattice	Mark II 32 node	[Flower:85a]
Flower, Otto	Heavy quark potential 20^4 lattice	Mark II 128 node	[Flower:86b]
Kolawa, Furmanski	Glueball mass (SU(2)), Hamiltonian "loop" formalism	Mark II 32 node	[Furmanski:87c]
Stolorz, Otto	Microcanonical renormalization group (SU(2))	Mark I 64 node	[Stolorz:86b]
Flower	Baryons on the lattice 20^4	Mark II 128 node	[Flower:87e]
Chiu	Random block lattice	NCUBE 1024 node	[Chiu:88a], [Chiu:88c], [Chiu:88e], [Chiu:88f] [Chiu:89a]
Ding, Baillie, Fox	Heavy quark potential $24^3 \times 10$, $24^3 \times 12$, $12^3 \times 24$, $16^3 \times 24$	Mark IIIfp 32 node	[Ding:89a]
Baillie, Brickner, Gupta	QCD with dynamical Wilson fermions 16^4 lattice	TMC CM-2 16K	[Baillie:89e]

Table 4.1: QCD on the Hypercube

In this language, the measurement of a physical observable represented by an operator $\mathcal{O}$ is given as the expectation value

$$\langle \mathcal{O}(\phi) \rangle = \frac{1}{Z} \int D\phi \mathcal{O}(\phi) e^{-S(\phi)}, \tag{4.2}$$

where the partition function Z is

$$Z = \int D\phi e^{-S(\phi)}. \tag{4.3}$$

In these expressions, the integral $D\phi$ indicates a sum over all possible configurations of the field ϕ. A typical observable would be a product of fields $\mathcal{O} = \phi(x)\phi(y)$, which says how the fluctuations in the field are correlated, and in turn, tells us something about the particles that can propagate from point x to point y. The appropriate correlation functions give us, for example, the masses of the various particles in the theory. Thus, to evaluate most any quantity in field theories like QCD, one must simply evaluate the corresponding path integral. The catch is that the integrals are over an infinite-dimensional space.

To put the field theory onto a computer, we begin by discretizing space and time into a lattice of points. Then the functional integral is simply defined as the product of the integrals over the fields at every site of the lattice $\phi(n)$:

$$\int D\phi \rightarrow \int \prod_n [d\phi(n)]. \tag{4.4}$$

Restricting space and time to a finite box, we end up with a finite (but large) number of ordinary integrals, something we might imagine doing directly on a computer. However, the high multi-dimensionality of these integrals renders conventional mesh techniques impractical. Fortunately, the presence of the exponential e^{-S} means that the integrand is sharply peaked in one region of configuration space. Hence, we resort to a statistical treatment and use Monte Carlo type algorithms to sample the important parts of the integration region.

Monte Carlo algorithms typically begin with some initial configuration of fields, and then make pseudo-random changes on the fields such that the ultimate probability P of generating a particular field configuration ϕ is proportional to the Boltzmann factor,

$$P(\phi) = e^{-S(\phi)}, \tag{4.5}$$

where $S(\phi)$ is the action associated with the given configuration. There are several ways to implement such a scheme, but for many theories the simple Metropolis algorithm [Metropolis:53a] is effective. In this algorithm, a new configuration ϕ' is generated by updating a single variable in the old configuration ϕ and calculating the change in energy (action)

$$\Delta S = S(\phi') - S(\phi). \tag{4.6}$$

If $\Delta S \leq 0$, the change is accepted; if $\Delta S > 0$ the change is accepted with probability $\exp(-\Delta S)$. In practice, this is done by generating a pseudo-random number r in the interval [0,1] with uniform probability distribution and accepting the change if $r < \exp(-\Delta S)$. This is guaranteed to generate the correct (Boltzmann) distribution of configurations, provided "detailed balance" is satisfied. That condition means that the probability of proposing the change $\phi \rightarrow \phi'$ is the same as that of proposing the reverse process $\phi' \rightarrow \phi$. In practice, this is true if we never simultaneously update two fields which interact directly via the action. Note that this constraint has important ramifications for parallel computers as we shall see below.

Whichever method one chooses to generate field configurations, one updates the fields for some equilibration time of E steps, and then calculates the expectation value of $\mathcal{O}$ in equation 4.2 from the next T configurations as

$$\langle \mathcal{O} \rangle = \frac{1}{T} \sum_{i=E+1}^{E+T} \mathcal{O}(\phi_i). \tag{4.7}$$

The statistical error in Monte Carlo behaves as $1/\sqrt{N}$, where N is the number of effectively independent configurations. $N = T/2\tau$, where τ is the autocorrelation time. This autocorrelation time can easily be large, and most of the computer time is then spent in generating statistically independent configurations. The operator measurements then become a small overhead on the whole calculation.

4.2.3 QCD

Quantum Chromo-dynamics (QCD) is the theory describing the strong interaction in high energy physics. This interaction binds quarks together to form hadrons—the constituents of nuclear matter—for example, the proton and neutron. Currently, we know of five types (referred to as "flavors") of quark: up, down, strange, charm and bottom; and expect at least one more (top) to show up soon. In addition to having a "flavor," quarks can carry one

of three possible charges known as "color" (this has nothing to do with color in the macroscopic world!); hence, Quantum *Chromo*-dynamics. The strong color force is mediated by particles called gluons, just as photons mediate light in electromagnetism. Unlike photons, though, gluons themselves carry a color charge and, therefore, interact with one another. This is the root of the nonlinearity of QCD, which causes all the difficulties.

4.2.4 Lattice QCD

To put QCD on a computer, we proceed as follows: The four-dimensional space-time continuum is replaced by a four-dimensional hypercubic periodic lattice, of size $N = N_s \times N_s \times N_s \times N_t$ with the quarks living on the sites and the gluons living on the links of the lattice. N_s is the spatial and N_t is the temporal extent of the lattice. The lattice has a finite spacing a. The gluons are represented by 3×3 complex $SU(3)$ matrices associated with each link in the lattice. The 3 in $SU(3)$ reflects the fact that there are three colors of quarks, and SU means that the matrices are unitary with unit determinant (*i.e.*, "special unitary"). This link matrix describes how the color of a quark changes as it moves from one site to the next. For example, as a quark is transported along a link of the lattice it can change its color from, say, red to green; hence, a red quark at one end of the link can exchange colors with a green quark at the other end. The action functional for the purely gluonic part of QCD is

$$S_G = \beta \sum_P (1 - \frac{1}{3} ReTrU_P), \tag{4.8}$$

where

$$U_P = U_{i,\mu}\, U_{i+\mu,\nu}\, U^\dagger_{i+\nu,\mu}\, U^\dagger_{i,\nu} \tag{4.9}$$

is the product of link matrices around an elementary square or plaquette on the lattice — see Figure 4.1. Essentially, all of the time in QCD simulations of gluons is spent multiplying these $SU(3)$ matrices together. The main component of this is the $a \times b + c$ kernel (which most supercomputers can do very efficiently). As the action involves interactions around plaquettes, in order to satisfy detailed balance, we can update only half the links in any one dimension simultaneously, as shown in Figure 4.2 (in two dimensions for simplicity). The partition function for full lattice QCD including quarks is then

$$Z = \int D\psi D\bar{\psi} DU \exp(-S_G - \bar{\psi}(\not{D} + m)\psi), \tag{4.10}$$

Figure 4.1: A Lattice Plaquette

Figure 4.2: Updating the Lattice

where $\not{D} + m$ is a large sparse matrix the size of the lattice squared. Unfortunately, since the quark or fermion variables ψ are anticommuting Grassmann numbers, there is no simple representation for them on the computer. Instead, they must be integrated out, leaving a highly non-local fermion determinant:

$$Z = \int DU \det(\not{D} + m) \exp(-S_G). \tag{4.11}$$

This is the basic integral one wants to evaluate numerically.

The biggest stumbling block preventing large QCD simulations, including quarks, is the presence of the determinant $\det(\not{D} + m)$ in the partition function. There have been many proposals for dealing with the determinant. The first algorithms tried to compute the change in the determinant when

a single link variable was updated [Weingarten:81a]. This turned out to be prohibitively expensive. So instead, the approximate method of pseudo-fermions [Fucito:81a] was used. Today, however, the preferred approach is the so-called "Hybrid Monte Carlo" algorithm [Duane:87a] which is exact. The basic idea is to invent some dynamics for the variables in the system in order to evolve the whole system forward in (simulation) time, and then do a Metropolis accept/reject for the entire evolution on the basis of the total energy change. The great advantage is that the whole system is updated in one fell swoop. The disadvantage is that if the dynamics are not correct, the acceptance will be very small. Fortunately, (and this is one of very few fortuitous happenings where fermions are concerned) good dynamics can be found: the Hybrid algorithm [Duane:85a]. This is a neat combination of the deterministic microcanonical method [Callaway:83a], [Polonyi:83a] and the stochastic Langevin method [Parisi:81a], [Batrouni:85a], which yields a quickly evolving, ergodic algorithm for both gauge fields and fermions. The computational kernel of this algorithm is the repeated solution of systems of equations of the form,

$$(\not{D} + m)\phi = \eta, \tag{4.12}$$

where ϕ and η are vectors that live on the sites of the lattice. To solve these equations, one typically uses a conjugate gradient or one of its cousins, since the fermion matrix $(\not{D} + m)$ is sparse. For more details, see [Gupta:88a]. Such iterative matrix algorithms have as their basic component the $a \times b + c$ kernel, so again computers which do this efficiently will run QCD well.

4.2.5 Concurrent QCD Machines

Where is the computer power needed for QCD going to come from? Today, the biggest resources of computer time for research are the conventional supercomputers at the NSF and DOE centers. The centers are continually expanding their support for lattice gauge theory, but it may not be long before they are overtaken by several dedicated efforts involving concurrent computers. It is a revealing fact that the development of most high-performance parallel computers—the Caltech Cosmic Cube, the Columbia Machine, IBM's GF11, APE in Rome, the Fermilab Machine—was actually motivated by the desire to simulate lattice QCD. Geoffrey Fox and Chuck Seitz, at Caltech, built the first hypercube computer, the Cosmic Cube or Mark I, in 1983. It had 64 nodes, each of which was an Intel 8086/87 microprocessor with 128 KB of memory, giving a total of about 2 Mflops (measured for QCD). This was quickly upgraded to the Mark II hypercube with faster chips, twice

the memory per node, and twice the number of nodes in 1984. Now QCD is running on the latest Caltech hypercube, the 128-node Mark IIIfp (built by JPL), at 600 Mflops sustained [Ding:89a]. Each node of the Mark IIIfp hypercube contains two Motorola 68020 microprocessors, one for communication and the other for calculation, with the latter supplemented by one 68881 coprocessor and a 32-bit Weitek floating point processor. Norman Christ and Anthony Terrano, at Columbia, built their first parallel computer for doing lattice QCD calculations in 1984. It had 16 nodes, each of which was an Intel 80286/87 microprocessor, plus a TRW 22-bit floating point processor with 1 MB of memory, giving a total peak performance of 256 Mflops. This was improved in 1987 using Weitek rather than TRW chips so that 64 nodes give 1 Gflops peak. Very recently, at Columbia, they have finished building their third machine: a 256-node 16 Gflops lattice QCD computer. Don Weingarten, at IBM, has been building the GF11 since 1984—it is expected that he will start running this in 1990. The GF11 is an SIMD machine comprising 576 Weitek floating point processors, each performing at 20 Mflops to give the total 11 Gflops peak implied by the name. The APE (Array Processor with Emulator) computer is basically a collection of 3081/E processors (which were developed by CERN and SLAC for use in high energy experimental physics) with Weitek floating point processors attached. However, these floating point processors are attached in a special way—each node has four multipliers and four adders, in order to optimize the $a \times b + c$ calculations, which form the major component of all lattice QCD programs. This means that each node has a peak performance of 64 Mflops. The first small machine—Apetto—was completed in 1986 and had four nodes yielding a peak performance of 256 Mflops. Currently, they have a second generation of this machine with 1 Gflops peak from 16 nodes. By 1992, the APE collaboration hopes to have completed the 100 Gflops 4096-node "Apecento", based on specialized VLSI chips that are software compatible with the original APE. Not to be outdone, Fermilab is also using its high-energy experimental physics emulators in constructing a lattice QCD machine for 1991 with 256 of them arranged as a 2^5 hypercube of crates, with eight nodes communicating through a crossbar in each crate. All together, they expect to get 5 Gflops peak performance. These performance figures are summarized in Table 4.2. (The "real" performances are the actual performances obtained on QCD codes.)

Major calculations have also been performed on commercial SIMD machines, first on the ICL Distributed Array Processor (DAP) at Edinburgh

Computer	Year	Peak	Real
Caltech I	1983	3	2
Caltech II	1984	9	6
Caltech III	1989	2000	600
Columbia I	1984	256	20
Columbia II	1987	1000	200
Columbia III	1990	16000	6000
IBM GF11	1990	11000	10000*
APE I	1986	256	20
APE II	1988	1000	200
APE III	1992	100000	20000*
Fermilab	1991	5000	1200*

*All real times are measured except these predicted ones.

Table 4.2: Peak and Real Performances in Mflops of "Homebrew" QCD Machines

University during the period 1982–1987, and now on the TMC Connection Machine (CM-2); and on commercial distributed memory MIMD machines like the NCUBE hypercube at Caltech. In fact, currently, the Connection Machine is the most powerful commercial QCD machine available: running full QCD at a sustained rate of approximately 1 Gflop on a 64 K CM-2 [Baillie:89e].

It is interesting to note that when the various groups began building their "homebrew" QCD machines, it was clear they would out-perform all commercial (traditional) supercomputers; however, now that commercial parallel supercomputers have come of age [Fox:89n] the situation is not so obvious. To emphasize this, we describe QCD calculations on both the home-grown Caltech hypercube and on the commercially available Connection Machine.

4.2.6 QCD on the Caltech Hypercubes

To make good use of MIMD distributed memory machines like hypercubes, one should employ domain decomposition. That is, the domain of the problem should be divided into subdomains of equal size, one for each processor in the hypercube; and communication routines should be written to take

care of data transfer across the processor boundaries. Thus, for a lattice calculation, the N sites are distributed among the $P = 2^d$ processors using a decomposition, which ensures that processors assigned to adjacent subdomains are directly linked by a communication channel in the hypercube topology. Each processor then independently works through its subdomain of $n = N/P$ sites updating each one in turn, only communicating with neighboring processors when doing boundary sites. This communication enforces "loose synchronization," which stops any one processor from racing ahead of the others. Load balancing is achieved with equal-size domains. If the nodes contain at least two sites of the lattice, all the nodes can update in parallel, satisfying detail balance, since loose synchronicity guarantees that all nodes will be doing black, then red sites alternately. The characteristic timescale of the communication, t_{comm}, corresponds to roughly the time taken to transfer a single $SU(3)$ matrix from one node to its neighbor. Similarly, we can characterize the calculational part of the algorithm by a timescale, t_{calc}, which is roughly the time taken to multiply together two $SU(3)$ matrices. For all hypercubes built *without* floating point accelerator chips $t_{comm} << t_{calc}$ and, hence, QCD simulations are extremely "efficient," where efficiency is defined by the relation

$$\epsilon = \frac{T_1}{k\,T_k}, \tag{4.13}$$

where T_k is the time taken for k processors to perform the given calculation. Typically, such calculations have efficiencies in the range $\epsilon \geq .90$, which means they are ideally suited to this type of computation since doubling the number of processors nearly halves the total computational time required for solution. However, as we shall see (for the Mark IIIfp hypercube, for example), the picture changes dramatically when fast floating point chips are used; then $t_{comm} \simeq t_{calc}$ and one must take some care in coding to obtain maximum performance.

Rather than describe every calculation done on the Caltech hypercubes, we shall concentrate on one calculation that has been done several times as the machine evolved—the heavy quark potential calculation ("heavy" because the quenched approximation is used).

QCD provides an explanation of why quarks are confined inside hadrons in that lattice calculations reveal that the inter-quark potential rises linearly as the separation between the quarks increases. Thus, no matter how far apart quarks separate they still experience a non-vanishing attractive force. This force, called the "string tension", is carried by the gluons, which form a "string" between the quarks. On the other hand, at short distances, quarks

and gluons are "asymptotically free" and behave like electrons and photons, interacting via a Coulomb-like force. Thus, the quark potential V is written as

$$V(R) = -\frac{\alpha}{R} + \sigma R, \tag{4.14}$$

where R is the separation of the quarks, α is the coefficient of the Coulombic potential and σ is the string tension. Experimentally, in fitting charmonium data to this Coulomb plus linear potential, Eichten *et al.* [Eichten:80a] estimated that $\alpha = 0.52$ and σ=0.18GeV^2. Thus, one goal of the lattice calculations is to reproduce these numbers. Another goal is to show that the numbers from the lattice are "scaling", that is, if one calculates a physical observable on lattices with different spacings then one gets the same answer. This means that the artifacts due to the finiteness of the lattice spacing have disappeared and continuum physics can be extracted.

The first heavy quark potential calculation using a Caltech hypercube was performed on the 64-node Mark I in 1984 on a $12^3 \times 16$ lattice with β ranging from 5.8 to 7.6 [Otto:84a]. α was found to be 0.25 and the string tension (converting to the dimensionless ratio) $\sqrt{\sigma}/\Lambda = 106$. The numbers are quite a bit off from the charmonium data but the string tension did appear to be scaling, albeit in the narrow window $6.0 \leq \beta \leq 6.4$.

The next time around, in 1986, the 128-node Mark II hypercube was used on a 20^4 lattice with $\beta = 6.1, 6.3, 6.5, 6.7$ [Flower:86b]. The dimensionless string tension decreased somewhat to 83, but clear violations of scaling were observed: the lattice was still too coarse to see continuum physics.

Therefore, the latest (1989) calculation using the Caltech/JPL 32-node Mark IIIfp hypercube concentrated on one β value, 6.0, and investigated different lattice sizes: $24^3 \times 10$, $24^3 \times 12$, $12^3 \times 24$, $16^3 \times 24$ [Ding:89a]. Scaling was not investigated, however, the values of $\alpha = 0.58$ and $\sqrt{\sigma}/\Lambda = 77$, *i.e.*, σ=0.15 GeV^2 compare favorably with the charmonium data. This work is based on about 1300 CPU hours on the 32-node Mark IIIfp hypercube, which has a performance of roughly twice a CRAY X-MP processor. The whole 128-node machine performs at 0.6 Gflops. As each node runs at 6 Mflops, this corresponds to a speedup of 100, and hence, an efficiency of 78%. These figures are for the most highly optimized code. The original version of the code written in C ran on the Motorola chips at 0.085 Mflops and on the Weitek chips at 1.4 Mflops. The communication time, which is roughly the same for both, is less than a 2% overhead for the former but nearly 30% for the latter. When the computationally intensive parts of the calculation are written in assembly code for the Weitek, this overhead becomes almost 50%.

Programming Level	Calc. Time	Comm. Time	Total Time	Mflops
1. Motorola MC68020/68881 in C	52	0.86	53	0.085
2. Weitek XL all in C	2.25	0.90	3.15	1.4
3. Weitek XL parts in Assembly	0.94	0.90	1.84	2.4
4. Weitek XL Assembly, vec. comm.	0.94	0.26	1.20	3.8
5. Weitek XL Assembly, no comm.	0.94	0.0	0.94	4.8

Table 4.3: Link Update Time (msec) on Mark IIIfp Node for Various Levels of Programming

This 0.9 msec of communication, shown in lines two and three in Table 4.3, is dominated by the hardware/software message startup overhead (latency), because for the Mark IIIfp the node to node communication time, t_{comm}, is given by

$$t_{comm} \simeq (150 + 2 * W) \quad \mu\,\text{sec}, \tag{4.15}$$

where W is the number of words transmitted. To speed up the communication, we update all even (or odd) links (eight in our case) in each node, allowing us to transfer eight matrix products at a time, instead of just sending one in each message. This reduces the 0.9 msec by a factor of

$$\frac{8 * (150 + 18 * 2)}{150 + 8 * 18 * 2} = 3.4 \tag{4.16}$$

to 0.26 msec. On all hypercubes with fast floating point chips—and on most hypercubes without these chips for less computationally intensive codes—such vectorization of communication is often important. In Figure 4.3, the speedups for many different total lattice sizes are shown. For the largest lattice size, the speedup is 100 on the 128-node. The speedup is almost linear in number of nodes. As the total lattice volume increases, the speedup increases, because the ratio of calculation/communication increases. For more information on this performance analysis, see [Ding:89b].

4.2.7 QCD on the Connection Machine

The Connection Machine Model CM-2 is also very well-suited for large-scale simulations of QCD. The CM-2 is a distributed-memory, single-instruction

Figure 4.3: Speedups for QCD

multiple-data (SIMD) massively parallel processor comprising up to 65536 (64 K) processors [Hillis:85a] [Hillis:87a]. Each processor consists of an arithmetic-logic unit (ALU), eight or 32 Kbytes of random-access memory (RAM) and a router interface to perform communications among the processors. There are 16 processors and a router per custom VLSI chip, with the chips interconnected as a 12-dimensional hypercube. Communications among processors within a chip work essentially like a crossbar interconnect. The router can do general communications, but we require only local ones for QCD so we use the fast nearest-neighbor communication software called NEWS. The processors deal with one bit at a time. Therefore, the ALU can compute any two Boolean functions as output from three inputs, and all data paths are one-bit wide. In the current version of the Connection Machine (the CM-2), groups of 32 processors (two chips) share a 32-bit (or 64-bit) Weitek floating point chip, and a transposer chip, which changes 32 bits stored bit-serially within 32 processors into 32 32-bit words for the Weitek, and vice versa.

The high-level languages on the CM, such as *Lisp and CM-Fortran, compile into an assembly language called Parallel Instruction Set (Paris). Paris regards the 64 K bit-serial processors as the fundamental units in the machine. However, floating-point computations are not very efficient in the Paris model. This is because in Paris, 32-bit floating point numbers are stored "field-wise;" that is, successive bits of the word are stored at successive memory locations of each processor's memory. However, 32 processors share one Weitek chip, which deals with words stored "slice-wise"—stored across the processors, one bit in each. Therefore, to do a floating-point op-

eration, Paris loads in the field-wise operands, transposes them slice-wise for the Weitek (using the transposer chip), does the operation, and transposes the slice-wise result back to field-wise for memory storage. Moreover, every operation in Paris is an atomic process; that is, two operands are brought from memory and one result is stored back to memory, so no use is made of the Weitek registers for intermediate results. Hence, to improve the performance of the Weiteks, a new assembly language called CM Instruction Set (CMIS) has been written, which models the local architectural features much better. In fact, CMIS ignores the bit-serial processors and thinks of the machine in terms of the Weitek chips. Thus, data can be stored slice-wise, eliminating all the transposing back and forth. CMIS allows effective use of the Weitek registers, creating a memory hierarchy, which combined with the internal buses of the Weiteks, offers increased bandwidth for data motion.

Currently, the Connection Machine is the most powerful commercial QCD machine available: the "Los Alamos collaboration" is running full QCD at a sustained rate of almost 2 Gflops on a 64 K CM-2. As was the case for the Mark IIIfp hypercube, in order to obtain this performance, one must resort to writing assembly code for the Weitek chips *and* for the communication. Our original code, written entirely in the CM-2 version of *Lisp, achieved around 1 Gflop [Baillie:89e]. As shown in Table 4.4, this code spends 34% doing communication. When we rewrote the most computationally intensive part in the assembly language CMIS, this rose to 54%. In order to obtain maximum performance, we are now rewriting the communication part of our code to make use of "multi-wire NEWS," which will allow us to communicate in all eight directions on the lattice simultaneously, thereby reducing the communication time by a factor of eight, and speeding up the code by another factor of two.

4.2.8 Status and Prospects

The status of lattice QCD may be summed up as: underway. Already there have been some nice results in the context of the quenched approximation, but the lattices are still too coarse and too small to give definitive results. Results for full QCD are going to take orders of magnitude more computer time, but we now have an algorithm—Hybrid Monte Carlo—which puts real simulations within reach.

When will the computer power be sufficient? In Figure 4.4, we plot the horsepower of various QCD machines as a function of the year they started

Programming Level	Calc. time	Comm. time	Total Time	Mflops
All in *Lisp	8.7	4.5	13.2	900
Inner Loop in CMIS	3.3	3.9	7.2	1650
Multi-wire CMIS†	<3.3	0.5	<3.8	>3100

† Projected numbers.

Table 4.4: Fermion Update Time (sec) on 64 K Connection Machine for Various Levels of Programming

Figure 4.4: Megaflops for QCD Calculations

to produce physics results. The performance plotted in this case is the real sustained rate on actual QCD codes. The surprising fact is that the rate of increase is very close to exponential, yielding a factor of ten every two years! On the same plot, we show our estimate of the computer power needed to re-do this year's quenched calculations on a 128^4 lattice. This estimate is also a function of time, due to algorithm improvements.

Extrapolating both lines, we see the outlook for lattice QCD is rather bright. Reasonable results for the "harder" physical observables should be available within the quenched approximation in the mid-90's. With the same computer power, we will be able to re-do today's quenched calculations using dynamic fermions (but still on today's size of lattice). This will tell us how reliable the quenched approximation is. Finally, results for the full theory with dynamical fermions on a 128^4 lattice should follow early in the next century (!), when computers are two or three orders of magnitude more

powerful again.

4.3 Spin Models

4.3.1 Introduction

Spin models are simple statistical models of real systems, which exhibit the same behavior and, hence, provide an understanding of the physical mechanisms involved. Despite their apparent simplicity, most of these models are not exactly soluble by present theoretical methods. Hence, computer simulation is used. Usually, one is interested in the behavior of the system at a phase transition; the computer simulation reveals where the phase boundaries are, what the phases on either side are, and how the properties of the system change across the phase transition. There are two varieties of spins: discrete or continuously valued. In both cases, the spin variables are put on the sites of the lattice and only interact with their nearest neighbors. The partition function for a spin model is

$$Z = \int Ds \exp(S), \tag{4.17}$$

with the action being of the form

$$S = \beta \sum_{\langle ij \rangle} (1 - s_i . s_j), \tag{4.18}$$

where $\langle ij \rangle$ denotes nearest neighbors. A great deal of work has been done over the years in finding good algorithms for computer simulations of spin models; recently some new, much better, algorithms have been discovered. These so-called cluster algorithms are described in Section 12.6.

4.3.2 Ising Model

The Ising model is the simplest model for ferromagnetism that predicts phase transitions and critical phenomena. This model, introduced by Lenz in 1920 [Lenz:20a], was solved in one dimension by Ising in 1925 [Ising:25a], and in two dimensions by Onsager in 1944 [Onsager:44a]. However, it has not been solved analytically in three dimensions, so Monte Carlo computer simulation methods have been one of the methods used to obtain numerical solutions. One of the best available techniques for this is the Monte Carlo Renormalization Group (MCRG) method [Wilson:80a], [Swendsen:79a]. The Ising

model exhibits a second-order phase transition in $d = 3$ dimensions at a critical temperature T_c. As T approaches T_c, the correlation length ξ diverges as a power law with critical exponent ν:

$$\xi = \xi_0(T/T_c - 1)^{-\nu} \tag{4.19}$$

and the pair correlation function $G(r)$ at $T = T_c$ falls off to zero with distance r as a power law defining the critical exponent η:

$$G(r) = G_0 r^{-(d-2+\eta)}. \tag{4.20}$$

T_c, ν and η determine the critical behavior of the 3-D Ising model and it is their values we wish to determine using MCRG.

Six years ago, this was done by Pawley, Swendsen, Wallace and Wilson [Pawley:84a] in Edinburgh on the ICL DAP computer with high statistics. They ran on four lattice sizes—8^3, 16^3, 32^3 and 64^3—measuring seven even and six odd spin operators. We are essentially repeating their calculation on the new AMT DAP computer. Why should we do this? First, to investigate finite size effects—we have run on the biggest lattice used by Edinburgh, 64^3, and on a bigger one, 128^3. Second, to investigate truncation effects—qualitatively the more operators we measure for MCRG, the better, so we have included 53 even and 46 odd operators. Third, we are making use of the new cluster-updating algorithm due to Swendsen and Wang [Swendsen:87a], implemented according to Wolff [Wolff:89b]. Fourth, we would like to try to measure another critical exponent more accurately—the correction-to-scaling exponent ω.

In order to calculate the quantities of interest using MCRG, one must evaluate the spin operators S_α. In [Pawley:84a], the calculation was restricted to seven even spin operators and six odd; we are evaluating 53 and 46, respectively. Specifically, we have decided to evaluate the most important operators in a $3 \times 3 \times 3$ cube [Baillie:88h]. To determine the critical coupling (or inverse temperature), K_c, one performs independent MC simulations on a large lattice L of size M^3 and on smaller lattices S of size $(M/2^m)^3$, $m = 1, 2, \ldots$, and compares the operators measured on the large lattice blocked m times more than the smaller lattices. $K = K_c$ when they are the same. Since the effective lattice sizes are the same, unknown finite size effects should cancel. The critical exponents, y_a, are obtained directly from the eigenvalues, λ_a, of the stability matrix, $T_{\alpha\beta}$, according to $\lambda_a = 2^{y_a}$. In particular, the leading eigenvalue of $T_{\alpha\beta}$ for the even S_α gives ν from $y_1 = 1/\nu$, and similarly $y_1 = (d + 2 - \eta)/2$ from the odd $T_{\alpha\beta}$.

The Distributed Array Processor (DAP) is a SIMD computer consisting of $D \times D$ bit-serial processing elements (PEs) configured as a cyclic two-dimensional grid with nearest-neighbor connectivity. The Ising model computer simulation is well-suited to such a machine since the spins can be represented as single-bit (logical) variables. In 3-D, the system of spins is configured as an $M \times M \times M$ simple cubic lattice, which is "crinkle mapped" onto the $D \times D$ DAP by storing $N \times N$ pieces of each of M planes in each PE: $M \times M \times M = M \times (N \times D) \times (N \times D)$, with $N = M/D$. Our Monte Carlo simulation uses a hybrid algorithm in which each sweep consists of 10 standard Metropolis [Metropolis:53a] spin-updates followed by one cluster-update using Wolff's single-cluster variant of the Swendsen and Wang algorithm. On the 128^3 lattice, the autocorrelation time of the magnetization reduces from 73 ± 2 sets of 100 sweeps for Metropolis alone to 5.0 ± 0.2 sets of 10 Metropolis plus one cluster-update for the hybrid algorithm. In order to measure the spin operators, S_α, the DAP code simply histograms the spin configurations so that an analysis program can later pick out each particular spin operator using a look-up table. Currently, the code requires the same time to do one histogram measurement, one Wolff single-cluster-update or 100 Metropolis updates. Therefore, our hybrid of 10 Metropolis plus one cluster-update takes about the same time as a measurement.

We have run for about 5000 hours on the AMT DAP 510 belonging to the Advanced Computing Research Facility (ACRF) at Argonne National Laboratory, accumulating one hundred thousand measurements (that is, one million Metropolis sweeps plus one hundred thousand Wolff cluster-updates) on lattices of size 64^3 and 128^3 at $K = 0.221654$. In order to estimate statistical errors, we split this data set into 10 bins, each containing ten thousand measurements.

In analyzing our results, the first thing we have to decide is the order in which to arrange our 53 even and 46 odd spin operators. Naively, they can be arranged in order of increasing total distance between the spins [Baillie:88h] (as was done in [Pawley:84a]). However, the ranking of a spin operator is determined physically by how much it contributes to the energy of the system. Thus, we did our analysis initially with the operators in the naive order to calculate their energies, then subsequently we used the "physical" order dictated by these energies. This physical order of the first 20 even operators is shown in Figure 4.5 with six of Edinburgh's operators indicated; the seventh Edinburgh operator (E-6) is our 21st. This order is important, because to access the systematic effects of truncation, we are going to analyze our data as a function of the number of operators included. Specifically, we

Figure 4.5: Our Order for Even Spin Operators

successively diagonalize the 1×1, 2×2, ..., (for even, 46×46 for odd) stability matrix $T_{\alpha\beta}$ to obtain its eigenvalues and, thus, the critical exponents.

We present our results in terms of the eigenvalues of the even and odd parts of $T_{\alpha\beta}$. The leading even eigenvalue on the first four blocking levels starting from the 64^3 lattice is plotted against the number of operators included in the analysis in Figure 4.6, and on the first five blocking levels starting from the 128^3 lattice in Figure 4.7. Similarly, the leading odd eigenvalues for 64^3 and 128^3 lattices are shown in Figure 4.8 and Figure 4.9 respectively. First of all, note that there are significant truncation effects—the value of the eigenvalues do not settle down until at least thirty and perhaps forty operators are included. We note also that our value agrees with Edinburgh's when around seven operators are included—this is a significant verification that the two calculations are consistent. With most or all of the operators included, our values on the two different lattice sizes agree, and the agreement improves with increasing blocking level. Thus, we feel that we have overcome the finite size effects so that a 64^3 lattice is just large enough. However, the advantage in going to 128^3 is obvious in Figure 4.7 and Figure 4.9: there, we can perform one more blocking, which reveals that the results on the fourth and fifth blocking levels are consistent. This means that we have eliminated most of the transient effects near the fixed point in the MCRG procedure. From the value of the even eigenvalue with all operators included on the 128^3 lattice, 1.5715(74), we obtain the critical exponent $\nu = 0.636(3)$, which is a little larger than Edinburgh's value (0.629(4)). Similarly, from the odd eigenvalue, 2.4781(12), we get $\eta = 0.044(2)$, which is also larger than Edinburgh's value (0.031(5)). The reason that Edinburgh's values are

Figure 4.6: Leading Even Eigenvalue

Figure 4.7: Leading Even Eigenvalue

different from ours is that they extrapolate their results to both an infinitely sized lattice and an infinite number of blocking levels; since our results are consistent on both our lattice sizes and on the two highest blocking levels, we have not done this. We extract the correction-to-scaling exponent according to $\lambda_2 = 2^{-\omega}$ from the fifth blocking level on the 128^3 lattice as $\omega = 0.50(5)$. (The result from the fourth blocking level on the 64^3 lattice agrees with this within the statistical error.)

Finally, perhaps the most important number, because it can be determined the most accurately, is K_c. By comparing the fifth blocking level on the 128^3 lattice to the fourth on the 64^3 lattice, we calculate $K - K_c = -0.000001$ using the leading even spin operator. If we repeat this for all the operators (53 of them) and average, then we obtain an error

Figure 4.8: Leading Odd Eigenvalue

Figure 4.9: Leading Odd Eigenvalue

of four in the last decimal place. Thus, we report our best estimate (so far) as $K_c = 0.221655(4)$.

4.3.3 Potts Model

The q-state Potts model [Potts:52a] consists of a lattice of spins σ_i, which can take q different values, and whose Hamiltonian is

$$H = K \sum_{\langle i,j \rangle} \delta_{\sigma_i \sigma_j}. \tag{4.21}$$

For $q = 2$, this is equivalent to the Ising model. The Potts model is thus a simple extension of the Ising model; however, it has a much richer phase structure, which makes it an important testing ground for new theories and algorithms in the study of critical phenomena [Wu:82a].

Monte Carlo simulations of Potts models have traditionally used local algorithms such as that of Metropolis, *et al.* [Metropolis:53a], however, these algorithms have the major drawback that near a phase transition the autocorrelation time (the number of sweeps needed to generate a statistically independent configuration) increases approximately as L^2, where L is the linear size of the lattice. New algorithms have recently been developed that dramatically reduce this "critical slowing down" by updating clusters of spins at a time (these algorithms are described in Section 12.6). The original cluster algorithm of Swendsen and Wang was implemented for the Potts model [Swendsen:87a], and there is a lot of interest in how well cluster algorithms perform for this model. At present, there are very few theoretical results known about cluster algorithms, and theoretical advances are most likely to come from first studying the simplest possible models.

We have made a high statistics study of the SW algorithm and the single cluster Wolff algorithm [Wolff:89b], as well as a number of variants of these algorithms, for the $q = 2$ and $q = 3$ Potts models in two dimensions [Baillie:89l]. We measured the autocorrelation time τ in the energy (a local operator) and the magnetization (a global one) on lattice sizes from 8^2 to 512^2. About 10 million sweeps were required for each lattice size in order to measure autocorrelation times to within about 1%. From these values, we can extract the dynamic critical exponent z, given by $\tau(L) \sim L^z$, where $\tau(L)$ is measured at the infinite volume critical point (which is known exactly for the $2D$ Potts model).

The simulations were performed on a number of different parallel computers. For lattice sizes of 128^2 or less, it is possible to run independent

Figure 4.10: Energy Autocorrelation Times, $q = 2$

simulations on each processor of a parallel machine, enabling us to obtain 100% efficiency by running 10 or 20 runs for each lattice size in parallel. These calculations were done using a 32-processor Meiko Computing Surface, a 20-processor Sequent Symmetry, a 20-processor Encore Multimax, and a 96-processor BBN GP1000 Butterfly, as well as a network of SUN workstations, and took approximately 15,000 processor-hours. For the largest lattice sizes, 256^2 and 512^2, a parallel cluster algorithm was required, due to the large amount of calculation (and memory) required. We have used the self-labeling algorithm described in Section 12.6, which gives fairly good efficiencies of about 70% on the machines we have used (an NCUBE-1 and a Symult S2010), by doing multiple runs of 32 nodes each for the 256^2 lattice, and 64 nodes for 512^2. Since this problem does not vectorize, using all 512 nodes of the NCUBE gives a performance approximately five times that of a single processor CRAY X-MP, while all 192 nodes of the Symult is equivalent to about six CRAYs. The calculations on these machines have so far taken about 1000 hours.

Results for the autocorrelation times of the energy for the Wolff and SW algorithms are shown in Figure 4.10 for $q = 2$ and Figure 4.11 for $q = 3$. We do not yet have sufficient statistics on the 512^2 lattice to extract reliable results, so only data for lattice sizes up to 256^2 are shown. As can be seen, the Wolff algorithm has smaller autocorrelation times than SW. However, the dynamical critical exponents for the two algorithms appear to be identical, being approximately 0.26(1) and 0.56(1) for $q = 2$ and $q = 3$ respectively, compared to values of approximately two for the standard Metropolis algorithm.

Figure 4.11: Energy Autocorrelation Times, $q = 3$

Burkitt and Heermann [Heermann:89a] have suggested that the increase in the autocorrelation time is a logarithmic one, rather than a power law for the $q = 2$ case (the Ising model), *i.e.*, $z = 0$. It is very difficult to distinguish between a logarithm and a small power. We still do not have enough data to decide the matter conclusively, although we will soon have enough statistics to extract reliable numbers for a lattice of size 512^2, which should allow us to distinguish between the two possibilities. In any case, the performance of the cluster algorithms for the Potts model is quite extraordinary, with autocorrelation times for the 256^2 lattice, which are hundreds of times smaller than for the Metropolis algorithm. In the future, we hope to use the cluster algorithms to perform greatly improved Monte Carlo simulations of various Potts models, to study their critical behavior.

4.3.4 XY Model

The XY (or O(2)) model consists of a set of continuous valued spins regularly arranged on a two-dimensional square lattice. 15 years ago, Kosterlitz and Thouless (KT) predicted that this system would undergo a phase transition as one changed from a low temperature spin wave phase to a high temperature phase with unbound vortices. KT predicted an approximate transition temperature, T_c, and the following unusual exponential singularity in the correlation length and magnetic susceptibility:

$$\xi = a_\xi e^{b_\xi t^{-\nu}}, \qquad \chi = a_\chi e^{b_\chi t^{-\nu}}, \tag{4.22}$$

with

$$\nu = \frac{1}{2}, \qquad \eta = \frac{1}{4}, \tag{4.23}$$

where $t = T - T_c$ and the correlation function exponent η is defined by the relation $\chi = c\,\xi^{2-\eta}$.

Our simulation [Gupta:88a] was done on the 128-node FPS (Floating Point Systems) T-Series hypercube at Los Alamos. FPS software allowed the use of C with a similar software model (communication implemented by subroutine call) to that used on the hypercubes at Caltech. Each FPS node is built around Weitek floating point units, and we achieved 2 Mflops per node in this application. The total machine ran at 1/4 Gflops, or at about twice the performance of one processor of a CRAY X-MP for this application. We use a $1-D$ torus topology for communications, with each node processing a fraction of the rows. Each row is divided into red/black alternating sites of spins and the vector loop is over a given color. This gives a natural data structure of $(n \times 128)$ words for lattices of size $(n \times 256)$. The internode communications, in both lattice update and in measurement of observables, can be done asynchronously and are a negligible overhead.

Previous numerical work was unable to confirm the KT theory, due to limited statistics and small lattices. Our high-statistics simulations are done on 64^2, 128^2, 256^2, and 512^2 lattices using a combination of over-relaxed and Metropolis algorithms which decorrelates as $\tau \approx 0.15\xi^{1.2}$. (For comparison, a Metropolis algorithm decorrelates as $\tau \approx 5\xi^2$.) Each configuration represents N_{or} over-relaxed sweeps through the lattice followed by N_{met} Metropolis sweeps. Measurement of observables is made on every configuration. The over-relaxed algorithm consists of reflecting the spin at a given site about Σ where Σ is the sum of the nearest-neighbor spins, *i.e.*,

$$s_{new} \;=\; \Sigma\, s^{\dagger}_{old}\, \Sigma. \tag{4.24}$$

This implementation [Creutz:87a] [Brown:87a] of the over-relaxed algorithm is microcanonical, and it reduces critical slowing down even though it is a local algorithm. The "hit" elements for the Metropolis algorithm are generated as $e^{i\theta}$, where θ is a uniform random number in the interval $[-\delta, \delta]$, and δ is adjusted to give an acceptance rate of 50–60%. The Metropolis hits make the algorithm ergodic, but their effectiveness is limited to local changes in the energy. In Figure 4.12, we show the autocorrelation time τ vs. the correlation length ξ; for $N_{or} = 8$, $N_{met} = 2$ we extract $\tau = 0.15\xi^{1.48}$ and for $N_{or} = 15$, $N_{met} = 2$ we get $\tau = 0.15\xi^{1.20}$.

Figure 4.12: Autocorrelation Times for the XY Model

Figure 4.13: Correlation Length for the XY Model

We ran at 14 temperatures near the phase transition and made unconstrained fits to all 14 data points (four parameter fits according to equations (XY)), for both the correlation length (Figure 4.13) and susceptibility (Figure 4.14). The key to the interpretation of the data is the fits. We find that fitting programs (e.g., MINUIT, SLAC) move incredibly slowly towards the true minimum from certain points (which we label spurious minima), which, unfortunately, are the attractors for most starting points. We found three such spurious minima (KT1-3) and the true minimum KT4, as listed in Table 4.5.

Thus, our data was found to be in excellent agreement with the KT theory and, in fact, this study provides the first direct measurement of ν from both χ and ξ data that is consistent with the KT predictions.

Figure 4.14: Susceptibility for the XY Model

	KT1	KT2	KT3	KT4
a_χ	2.155	0.270	0.784	0.2727(3)
b_χ	1.088	2.809	1.844	2.7734(5)
T_c	0.701	0.911	0.867	0.8987(1)
ν	1.460	0.472	0.701	0.4995(11)
$\chi^2/_{dof}$	73.0	13.2	1.82	1.70
a_ξ	0.708	0.228	0.441	0.2280(3)
b_ξ	0.740	1.693	1.100	1.6725(8)
T_c	0.689	0.910	0.867	0.8967(2)
ν	1.412	0.471	0.700	0.5007(3)
$\chi^2/_{dof}$	7.94	2.62	0.89	0.37
η	0.53	0.34	0.324	0.34

Table 4.5: Results of the XY model fits: (a) χ in T, and (b) ξ in T assuming the KT form. The fits KT1-3 are pseudo-minima while KT4 is the true minimum. All data points are included in the fits and we give the $\chi^2/_{dof}$ for each fit and an estimate of the exponent η.

4.3.5 O(3) Model

The XY model is the simplest O(N) model, having $N = 2$; the O(N) model being a set of rotors (N-component continuous valued spins) on an N-sphere. For $N \geq 3$, this model is asymptotically free [Polyakov:75a], and for $N = 3$, there exist instant-on solutions. Some of these properties are analogous to those of gauge theories in four dimensions; hence, these models are interesting. In particular, the O(3) model in two dimensions should shed some light on the asymptotic freedom of QCD (SU(3)) in four dimensions. The predictions of the renormalization group for the susceptibility χ and inverse correlation length (*i.e.*, mass gap) m in the O(3) model are [Brezin:76a]

$$\chi = C\beta^{-4}\exp(4\pi\beta)(1 + O(1/\beta)) \tag{4.25}$$

and

$$m = \frac{1}{\xi} = C'\beta\exp(-2\pi\beta)(1 + O(1/\beta)), \tag{4.26}$$

respectively. If m and χ vary according to these equations, without the correction of order $1/\beta$, they are said to follow asymptotic scaling. Previous work was able to confirm that this picture is qualitatively correct, but was not able to probe deep enough in the area of large correlation lengths to obtain good agreement.

The combination of the over-relaxed algorithm and the computational power of the FPS T-Series allowed us to simulate lattices of size up to 1024^2. Most statistics were gathered at sizes of 512^2 and 768^2. We were, thus, able to simulate at coupling constants that correspond to correlation lengths up to 120, on lattices where finite size effects are negligible. We were also able to gather large statistics and, thus, obtain small statistical errors. Our simulation is in good agreement with the similar cluster calculations [Wolff:89b] [Wolff:90a]. Thus, we have validated and extended these results in a regime where our algorithm is the only known alternative to clustering.

We have made extensive runs at seven values of the coupling constant. At the lowest β, several hundred thousand sweeps were collected while for the largest values of β, between 50,000 and 100,000 sweeps were made. Each sweep consists of between 10 iterations through the lattice at the former end and 150 iterations at the latter. The statistics we have gathered are equivalent to about 200 days, using the full 128-node FPS machine.

Our results for the correlation length, and susceptibility for each coupling constant β and lattice size, are shown in Table 4.6. The autocorrelation time for some values of N_{or} are also shown. The quantities measured on different

Beta	L,T	τ	N_{or}	χ	ξ
1.50	256	1.3	12	176.4 ± 0.2	11.1
1.60	256	2.5	12	448.4 ± 0.7	19.0
1.70	256	7.0	12	1266 ± 4	34.4
	512	1.8	35	1264 ± 3	34.3
1.75	768	1.8	50	2197 ± 15	47.3
1.80	512	3.5	40	3850 ± 10	64.6
	768	3.0	45	3820 ± 20	65.2
1.85	768	2.3	80	6732 ± 25	89.3
1.90	1024, 512	2.0	150	11600 ± 60	121.4
	1024	1.5	200	11790 ± 80	122.1

Table 4.6: Lattice size, autocorrelation time, number of over-relaxed sweeps, susceptibility and correlation length for the O(3) model.

sized lattices at the same β agree, showing that the infinite volume limit has been reached.

In Figures 4.15 and 4.16, we show a comparison of the above equations with the observed behavior of χ and ξ, respectively. The lines pass through the point for $\beta = 1.9$ and follow the behavior of the equations. We see that asymptotic scaling does not set in for $\beta \leq 1.9$ ($\xi \leq 120$).

We gauged the speed of the algorithm in producing statistically independent configurations by measuring the autocorrelation time τ. We used this to estimate the dynamical critical exponent z, which is defined by $\tau = c \cdot \xi^z$. For constant $N_{or} = 12$, our fits give $z = 1.33(1)$. However, we discovered that by increasing N_{or} in rough proportion to ξ, we can improve the performance of the algorithm significantly. To compare the speed of decorrelation between runs with different N_{or}, we define a new quantity, which we call 'effort', $e = N_{or} \times \tau$. This measures the computational effort expended to obtain a decorrelated configuration. We define a new exponent z' from $e \sim \xi^{z'}$, where N_{or} is chosen to keep τ constant .

Figure 4.17 shows the relationship between effort and ξ for different values of N_{or}. The solid line is for constant $N_{or} = 20$, with slope z, while the dashed line is for $\tau = 2.2$, with slope z'. It is obvious that $z' < z$, and that this relates into a substantial improvement in performance. A fit to all our

Figure 4.15: The Susceptibility vs. the Coupling Constant for the O(3) Model

Figure 4.16: The Correlation Length vs. the Coupling Constant for the O(3) Model

Figure 4.17: Effort vs. Correlation Length for the O(3) Model

Figure 4.18: Effort vs. Number of Over-relaxed Sweeps for the O(3) Model

points (N_{or}, ξ, τ) using

$$\tau = C'' \cdot \xi^z \cdot N_{or}^{-z/z'} \tag{4.27}$$

gives $z = 1.30(1)$, $z' = 1.09(1)$. Figure 4.18 (effort vs. N_{or}) shows how varying the value of N_{or} reveals a minimum of the effort, which is broad and corresponds to $0.8 < \tau < 1.8$.

4.4 Seismic Wave Simulation

*** Contribution needed from R. Clayton

4.5 Coulomb Gas

*** Contribution needed from G. C. Fox

4.6 An Automata Model of Granular Materials

4.6.1 Introduction

Physical systems comprised of discrete, macroscopic particles or grains, which are not bonded to one another, are important in civil, chemical, and agricultural engineering, as well as in natural geological and planetary environments. Granular systems are observed in rock slides, sand dunes, clastic sediments, snow avalanches, and planetary rings. In engineering and industry they are found in connection with the processing of cereal grains, coal, gravel, oil shale, and powders, and are well-known to pose important problems associated with the movement of sediments by streams, rivers, waves, and the wind.

The standard approach to the theoretical modelling of multiparticle systems in physics has been to treat the system as a continuum and to formulate the model in terms of differential equations. As an example, the science of soil mechanics has traditionally focused mainly on quasi-static granular systems, a prime objective being to define and predict the conditions under which failure of the granular soil system will occur. Soil mechanics is a macroscopic continuum model requiring an explicit constitutive law relating, for example; stress and strain. While very successful for the low-strain quasi-static applications for which it is intended, it is not clear how it can be generalized to deal with the high-strain, explicitly time-dependent phenomena which characterize a great many other granular systems of interest. Attempts at obtaining a generalized theory of granular systems using a differential equation formalism [Johnson:87a] have met with limited success.

An alternate approach to formulating physical theories can be found in the concept of cellular automata, which was first proposed by Von Neumann in 1948. In this approach, the space of a physical problem would be divided up into many small, identical cells each of which would be in one of a finite number of states. The state of a cell would evolve according to a rule that is both local (involves only the cell itself and nearby cells) and universal (all cells are updated simultaneously using the same rule).

The Lattice Grain Model [Gutt:89a] (LGrM) we discuss here is a microscopic, explicitly time-dependent, cellular automata model, and can be ap-

plied naturally to high-strain events. LGrM carries some attributes of both particle dynamics models [Cundall:79a], [Haff:87a], [Haff:87b], [Walton:84a], [Werner:87a] (PDM), which are based explicitly on Newton's second law, and lattice gas models [Frisch:86a], [Margolis:86a] (LGM), in that its fundamental element is a discrete particle, but differs from these substantially in detail. Here we describe the essential features of LGrM, compare the model with both PDM and LGM, and finally discuss some applications.

4.6.2 Comparison to Particle Dynamics Models

The purpose of the lattice grain model is to predict the behavior of large numbers of grains (10,000 to 1,000,000) on scales much larger than a grain diameter. In this respect, it goes beyond particle dynamics calculations, which are limited to no more than $\sim 10,000$ grains by currently available computing resources [Cundall:79a], [Haff:87a], [Haff:87b], [Walton:84a], [Werner:87a]. The particle dynamics models follow the motion of each individual grain exactly, and may be formulated in one of two ways depending upon the model adopted for particle-particle interactions.

In one formulation, the interparticle contact times are assumed to be of finite duration, and each particle may be in simultaneous contact with several others [Cundall:79a], [Haff:87a], [Walton:84a], [Werner:87a]. Each particle obeys Newton's law, $F = ma$, and a detailed integration of the equations of motion of each particle is performed. In this form, while useful for applications involving a much smaller number of particles than LGrM allows, PDM cannot compete with LGrM for systems involving large numbers of grains because of the complexity of PDM "automata".

In the second, simpler formulation, the interparticle contact times are assumed to be of infinitesimal duration, and particles undergo only binary collisions (the hard-sphere collisional models) [Haff:87b]. Hard-sphere models usually rely upon a collision-list ordering of collision events to avoid the necessity of checking all pairs of particles for overlaps at each time step. In regions of high particle number density, collisions are very frequent; and thus in problems where such high density zones appear, hard-sphere models spend most of their time moving particles through very small distances using very small time steps. In granular flow, zones of stagnation where particles are very nearly in contact much of the time are common, and the hard-sphere model is therefore unsuitable, at least in its simplest form, as a model of these systems. LGrM avoids these difficulties because its time-stepping is controlled, not by a collision list but by a scan frequency, which in turn is

a function of the speed of the fastest particle and is independent of number density. Furthermore, although fundamentally a collisional model, LGrM can also mimic the behavior of consolidated or stagnated zones of granular material in a manner which will be described.

4.6.3 Comparison to Lattice Gas Models

LGrM closely resembles LGM [Frisch:86a], [Margolis:86a] in some respects. First, for two-dimensional applications, the region of space in which the particles are to move is discretized into a triangular lattice-work, upon each node of which can reside a particle. The particles are capable of moving to neighboring cells at each tick of the clock, subject to certain simple rules. Finally, two particles arriving at the same cell (LGM), or adjacent cells (LGrM) at the same time, may undergo a "collision" in which their outgoing velocities are determined according to specified rules chosen to conserve momentum.

Each of the particles in LGM has the same magnitude of velocity and is allowed to move in one of six directions along the lattice, so that each particle travels exactly one lattice spacing in each time step. The single velocity magnitude means that all collisions between particles are perfectly elastic and that energy conservation is maintained simply through particle number conservation. It also means that the temperature of the gas is uniform throughout time and space, thus limiting the application of LGM to problems of low Mach number. An exclusion principle is maintained in which no two particles of the same velocity may occupy one lattice point. Thus, each lattice point may have no more than six particles, and the state of a lattice point can be recorded using only six bits.

LGrM differs from LGM in that has many possible velocity states, not just six. In particular, in LGrM not only the direction but the magnitude of the velocity can change in each collision. This is a necessary condition because the collision of two macroscopic particles is always inelastic, so that mechanical energy is not conserved. The LGrM particles satisfy a somewhat different exclusion principle: no more than one particle at a time may occupy a single site. This exclusion principle allows LGrM to capture some of the volume-filling properties of granular material, in particular, to be able to approximate the behavior of static granular masses.

The determination of the time step is more critical in LGrM than in LGM. If the time step is long enough that some particles travel several lattice spacings in one clock tick, the problem of finding the intersection of particle trajectories arises. This involves much computation and defeats the purpose

of an automata approach. A very short time step would imply that most particles would not move even a single lattice spacing. Here we choose a time step such that the fastest particle will move exactly one lattice spacing. A "position offset" is stored for each of the slower particles, which are moved accordingly when the offset exceeds one-half lattice spacing. These extra requirements for LGrM automata imply a slower computation speed than expected in LGM simulations; but, as a dividend, we can compute inelastic grain flows of potential engineering and geophysical interest.

4.6.4 The Rules for the Lattice Grain Model

In order to keep the particle-particle interaction rules as simple as possible, all interparticle contacts, whether enduring contacts or true collisions, will be modeled as collisions. Collisions that model enduring contacts will transmit, in each time step, an impulse equal to the force of the enduring contact times the time step. The fact that collisions take place between particles on adjacent lattice nodes means that some particles may undergo up to six collisions in a time step. For simplicity, these collisions will be resolved as a series of binary collisions. The order in which these collisions are calculated at each lattice node, as well as the order in which the lattice nodes are scanned, is now an important consideration.

The rules of the Lattice Grain Model may be summarized as follows:

1. The particles reside on the nodes of a two-dimensional triangular lattice, obeying the exclusion principle that no node may have more than one particle.

2. Each particle has two components of velocity, which may take on any value. At the beginning of each time step, each particle's velocity is incremented due to the acceleration of gravity.

3. The size of each time step is set so that the fastest particle will travel one lattice spacing in that time step.

4. Two components of a "position offset" are maintained for each particle. This offset is incremented after the velocities in each time step according to gravitational acceleration and the particle's velocity:

$$\Delta q_i = v_i \Delta t + \frac{1}{2} g_i \Delta t^2 \tag{4.28}$$

where:

i = 1,2;
Δq_i = i*th* component of increment in position offset;
v_i = i*th* component of particle velocity;
g_i = i*th* component of gravitational acceleration;
Δt = current time step.

Once the offset exceeds half the distance to the nearest lattice node, and that node is empty, the particle is moved to that node, and its offset is decremented appropriately. Also, in a collision, the component of the offset along the line connecting the centers of the colliding particles is set to zero.

5. The order in which the lattice is scanned is chosen so as not to create a coupling between the scan pattern and the particle motions. Thus, the particle position updates are done on every third lattice point of every third row, with this pattern being repeated nine times to cover all lattice sites.

6. Particle collisions are calculated assuming that they are smooth hard disks with a given coefficient of restitution. Particles on adjacent nodes are assumed to collide if their relative velocity is bringing them together. The following order has been adopted for evaluating possible collisions on odd time steps: 3b, 3c, 3f, 2f, 2c, 2b, 4b, 4c, 4f, 1f, 1c, 1b; and for even time steps: 1b, 1c, 1f, 4f, 4c, 4b, 2b, 2c, 2f, 3f, 3c, 3b (where the lattice numbers and collision directions are defined in Figure 4.19).

7. In order to incorporate a container, wall, or other barrier within these rules, a second type of particle is introduced—the wall particle. This particle is similar to the movable particles, and interacts with them through binary collisions (with a separately defined inelasticity), but is regarded as having infinite mass. To allow for the introduction of shearing motion from a wall (as in a Couette flow problem), the particles making up the wall are given a common constant velocity, which is used in the usual fashion for calculating the results of collisions. However, the position of the wall particles in the lattice remains fixed throughout the simulation.

8. Though a single particle does not accurately predict the trajectory of a single grain, we still regard each particle as representing one grain

Figure 4.19: Definition of Lattice Numbers and Collision Directions

when we are extracting information from the simulation regarding the behavior of groups of grains. Thus, the size of one particle, as well as the spacing between lattice points, is taken to be a one-grain diameter.

The transmission of "static" contact forces within a mass of grains (as in grains at rest in a gravitational field) is handled naturally within the above framework. Though a particle in a static mass of grains may be nominally at rest, its velocity may be nonzero (due to gravitational or pressure forces); and it will transmit the appropriate force (in the form of an impulse) to the particles under it by means of collisions. When these impulses are averaged over several time steps, the proper weights and pressures will emerge.

4.6.5 Implementation on a Parallel Processor Computer

When implementing this algorithm on a computer, what is stored in the computer's memory is information concerning each point in the lattice, regardless of whether or not there is a particle at that lattice point. This allows for very efficient checking of the space around each particle for the presence of other particles (*i.e.*, information concerning the six adjacent points in a triangular lattice will be found at certain known locations in memory). The need to keep information on empty lattice points in memory does not entail as great a penalty as might be thought; many lattice grain model problems involve a high density of particles, typically one for every one to four lattice points, and the memory cost per lattice point is not large. The memory requirements for the implementation of LGrM as described here are five variables per lattice site: two components of position, two components of

velocity, and one status variable, which denotes an empty site, an occupied site, or a bounding "wall" particle. If each variable is stored using four bytes of memory, then each lattice point requires 20 bytes.

The standard configuration for a simulation consists of a lattice with a specified number of rows and columns bounded at the top and bottom by two rows of wall particles (thus forming the top and bottom walls of the problem space), and with left and right edges connected together to form periodic boundary conditions. Thus, the boundaries of the lattice are handled naturally within the normal position updating and collision rules, with very little additional programming. (Note: since the gravitational acceleration can point in an arbitrary direction, the top and bottom walls can become side walls for chute flow. Also, the periodic boundary conditions can be broken by the placement of an additional wall, if so desired.)

Because of the nearest-neighbor type interactions involved in the model, the computational scheme was well-suited to an NCUBE parallel processor. For the purpose of dividing up the problem, the hypercube architecture is unfolded into a two-dimensional array, and each processor is given a roughly equal-area section of the lattice. The only interaction between sections will be along their common boundaries; thus, each processor will only need to exchange information with its eight immediate neighbors. The program itself was written in C under the Cubix/CrOSIII operating system.

4.6.6 Simulations

The LGrM simulations performed so far have involved from $\sim 10^4$ to 10^6 automata. Trial application runs included 2D, vertical, time-dependent flows in several geometries, of which two examples are given here: Couette flow and flow out of an hourglass-shaped hopper.

The standard Couette flow configuration consists of a fluid confined between two flat parallel plates of infinite extent, without any gravitational accelerations. The plates move in opposite directions with velocities that are equal and that are parallel to their surfaces, which results in the establishment of a velocity gradient and a shear stress in the fluid. For fluids that obey the Navier-Stokes equation, an analytical solution is possible in which the velocity gradient and shear stress are constant across the channel. If, however, we replace the fluid by a system of inelastic grains, the velocity gradient will no longer necessarily be constant across the channel. Typically, stagnation zones or plugs form in the center of the channel with thin shear-bands near the walls. Shear-band formation in flowing granular

Figure 4.20: (a) Elastic Particle Couette Flow (b) X-component (1), y-component (2), and Second Moment (3) of Velocity

materials was analyzed earlier by Haff and others [Haff:83a], [Hui:84a] based on kinetic theory models.

The simulation was carried out with 5760 grains, located in a channel 60 lattice points wide by 192 long. Due to the periodic boundary conditions at the left and right ends, the problem is effectively infinite in length. The first simulation is intended to reproduce the standard Couette flow for a fluid; consequently, the particle-particle collisions were given a coefficient of restitution of 1.0 (*i.e.*, perfectly elastic collisions) and the particle-wall collisions were given a .75 coefficient of restitution. The inelasticity of the particle-wall collisions is needed to simulate the conduction of heat (which is being generated within the fluid) from the fluid to the walls. The simulation was run until an equilibrium was established in the channel (Figure 4.20(a)). The average x- and y-components of velocity and the second moment of velocity, as functions of distance across the channel are plotted in Figure 4.20(b).

The second simulation used a coefficient of restitution of .75 for both the particle-particle and particle-wall collisions. The equilibrium results are shown in Figure 4.21(a) and Figure 4.21(b). As can be seen from the plots, the flow consists of a central region of particles compacted into a plug, with each particle having almost no velocity. Near each of the moving walls, a region of much lower density has formed in which most of the shearing motion occurs. Note the increase in value of the second moment of velocity (the granular "thermal velocity") near the walls, indicating that grains in this area are being "heated" by the high rate of shear. It is interesting to note that these flows are turbulent in the sense that shear stress is a quadratic,

Figure 4.21: (a) Inelastic Particle Couette Flow (b) X-component (1), y-component (2), and Second Moment (3) of Velocity.

not a linear, function of shear rate.

In the second problem, the flow of grains through a hopper or an hourglass with an opening only a few grain diameters wide, was studied; the driving force was gravity. This is an example of a granular system which contains a wide range of densities, from groups of grains in static contact with one another to groups of highly agitated grains undergoing true binary collisions. Here, the number of particles used was 8310; and the lattice was 240 points long by 122 wide. Additional walls were added to form the sloped sides of the bin and to close off the bottom of the lattice to prevent the periodic boundary conditions from reintroducing the falling particles back into the bin (Figure 4.22(a)). This is a typical feature of automata modeling. It is often easier to configure the simulation to resemble a real experiment—in this case by explicitly "catching" spent grains—than by re-programming the basic code to erase such particles.

The hourglass flow, Figure 4.22(b), showed internal shear zones, regions of stagnation, free-surface evolution toward an angle of repose, and an exit flow rate approximately independent of pressure head, as observed experimentally [Tuzun:82a]. It is hard to imagine that one could solve a partial differential equation describing such a complex, multiple-domain, time-dependent problem, even if the right equation were known (which is not the case).

Figure 4.22: (a) Initial Condition of Hourglass (b) Hourglass Flow after 2048 Time Steps

4.6.7 Conclusion

These exploratory numerical experiments show that an automata approach to granular dynamics problems can be implemented on parallel computing machines. Further work remains to be done to assess more quantitatively how well such calculations reflect the real world, but the prospects are intriguing.

4.6.8 Acknowledgements

This work was supported in part by USDOE [DE-FG22-86-PC90959] and by USARO [DAAL03-86-K-0123] and [DAAL03-89-K-0089]. We thank Geoffrey Fox for making available to us the resources of the Caltech Concurrent Computation Project and Tom Tombrello for his interest and support.

Chapter 5

Express and CrOS — Loosely Synchronous Message Passing

5.1 Problem Requirements for Message Passing

*** Contribution needed from G. C. Fox

5.2 A "Packet" History of Message-passing Systems

The evolution of the various message-passing paradigms used on the Caltech/JPL machines involved three generations of hypercubes and many different software concepts, which ultimately led to the development of *Express*, a general, asynchronous buffered communication system for homogeneous multiprocessors.

Originally designed, developed and used by scientists with serious research goals, the Caltech/JPL system software was written to generate correct answers quickly. Unhindered by any preconceptions about the type of software that should be used for parallel processing, we simply built useful software and added it to the system library.

Hence, the software evolved from primitive hardware-dependent implementations into a sophisticated runtime library, which embodied the concepts of "loose synchronization," domain decomposition, and machine-

independence. By the time the commercial machines started to replace our homemade hypercubes we had evolved a programming model that allowed us to develop and debug code almost as easily as for conventional sequential computers, port it between different parallel computers, and run with minimal overheads.

How did this happen?

5.2.1 Pre-history

The original hypercubes, the Cosmic Cube [Seitz:85a], and Mark II [Tuazon:85a] machines had been designed and built as exploratory devices. We expected to be able to do useful physics and, in particular, were interested in high energy physics. At that time, we were merely trying to extract exciting physics from an untried technology. We never intended to revolutionize the field of interprocessor communication—the "field" didn't exist. These first machines came equipped with "64-bit FIFOs," meaning that at a software level two basic communication routines were available:

```
rdELT(packet, chan)
wtELT(packet, chan).
```

The latter pushed a 64-bit "packet" into the indicated hypercube channel, which was then extracted with the rdELT function. If the read happened before the write, the program in the reading node stopped and waited for the data to show up. If the writing node sent its data before the reading node was ready, it similarly waited for the reader.

To make contact with the world outside the hypercube cabinet, it had to be able to communicate with a "host" computer. Again, the FIFOs came into play with two additional calls

```
rdIH(packet)
wtIH(packet),
```

which allowed node 0 to communicate with the host.

This rigidly defined behavior, executed on a hypercubic lattice of nodes, resembled a crystal, so we called it the "Crystalline Operating System" (CrOS). Obviously, an operating system with only four system calls is quite far removed from most people's concept of the breed. Nevertheless, they were the only system calls available and the name stuck.

5.2.2 Application-driven Development

We began to build algorithms and methods to exploit using the power of parallel computers. With little difficulty, we were able to develop algorithms for solving partial differential equations [Brooks:82b], FFTs [Newton:82a], [Salmon:86b] and high energy physics problems [Brooks:83a].

As each person wrote applications; however, we learned a little more about the way problems were mapped onto the machines. Gradually, additional functions were added to the list to download and upload data sets from the outside world and to combine the operation of the `rdELT` and `wtELT` functions into something that exchanged data across a channel.

In each case, these functions were adopted, not because they seemed necessary to complete our operating system, but because they fulfilled a real need. At that time, debugging capabilities were nonexistent; a mistake in the program running on the nodes merely caused the machine to stop running. Thus, it was beneficial to build up a library of routines that performed common communication functions, which made re-inventing tools unnecessary.

5.2.3 Collective Communication

Up to this point, our primary concern was with communication between neighboring processors. Applications, however, tended to show two fundamental types of communication: local exchange of boundary condition data; and global operations connected with control or extraction of physical observables.

These two types of communication are generally believed to be fundamental to all scientific problems—the modelled application usually has some structure that can be mapped onto the nodes of the parallel computer and its structure induces some regular communication pattern. A major breakthrough, therefore, was the development of what have since been called the "collective" communication routines, which perform some action across all the nodes of the machine.

The simplest example is that of "`broadcast`"—a function that enabled node 0 to communicate one or more packets to all the other nodes in the machine. The routine "`concat`" enabled each node to accumulate data from every other node, and "`combine`" let us perform actions, such as addition on distributed data sets.

The development of these functions, and the natural way in which they

Figure 5.1: The "Message-passing" family tree

Figure 5.2: Glossary

Figure 5.3: Mapping a Two-dimensional World

could be mapped to the hypercube topology of the machines, led to great increases in both productivity on the part of the programmers and efficiency in the execution of the algorithms. CrOS quickly grew to a dozen or more routines.

5.2.4 Automated Decomposition—whoami

By 1985, the Mark II machine was in constant use and we were beginning to examine software issues that had previously been of little concern. Algorithms, such as the standard FFT, had obvious implementations on a hypercube [Salmon:86b]—the "bit-twiddling" done by the FFT algorithm could be mapped onto a hypercube by "twiddling" the bits in a slightly revised manner. More problematic was the issue of two- or three-dimensional problem solving. A two-dimensional problem could easily fit into a four-node system and with a slight stretch of the imagination might get up to eight nodes, but 128 nodes seemed out of the question. The human mind simply cannot perform the mapping of an $N \times N$ square grid onto 128 nodes connected to a seven-dimensional hypercube. For a problem based on a three dimensional world, the mapping was even more complex.

Another issue that quickly became apparent was the fact that no one had a good feel for the "chan" argument used by the primitive communication functions. Users wanted to think of a collection of processors each labeled by a number, with data exchanged between them, but unfortunately the software was driven instead by the hypercube architecture of the machine. The limit on the number of times that the explanation of "well, you XOR

the processor number with one shifted left by the..." was rapidly exceeded in all but the most stubborn cases.

Both problems were effectively removed by the development of **whoami** [Salmon:84b]. Using age-old techniques of Binary Grey-codes we discovered that the process of mapping two, three, or higher dimensional problems to the hypercube could be automated. The **whoami** function took the dimensionality of the physical system being modelled and returned all the necessary "**chan**" values to make everything else work out properly. No longer did the new user have to spend time learning about channel numbers, XORing and the "mapping problem"—everything was done by the magical call to **whoami**. Even on current hardware, where long-range communication is an accepted fact, the techniques embodied by **whoami** result in programs that run up to an order of magnitude faster than those using other mapping techniques.

5.2.5 "Melting"—a Non-crystalline Problem

Up to this point, we had concentrated on the most obvious scientific problems: FFTs; ordinary and partial differential equations; matrices; etc., which were all characterized by their amenability to the lock-step, short-range communication primitives available. Note that some of these, such as the FFT and matrix algorithms, are not strictly "nearest neighbor" in the sense of the communication primitives discussed earlier, since they require data to be distributed to nodes further than one step away. These problems, however, are amenable to the "collective communication" strategies.

Based on our success with these problems, we began to investigate areas that were not so easily cast into the crystalline methodology. A long-term goal was the creation of event-driven simulations, database machines and transaction-processing systems, which did not appear to be crystalline.

In the shorter term, we wanted to study the physical process of "melting" [Johnson:86a]. The melting process is different from the applications described thus far, in that it inherently involves some indeterminacies—the transition from an ordered solid to a random liquid involves complex and time-varying interactions. In the past, we had solved such an irregular problem—that of N-body gravity [Salmon:86b] by the use of what has since been called the "long-range-force" algorithm. This is a particularly powerful technique and leads to highly efficient programs that can be implemented with crystalline commands.

The melting process differs from the long-range force algorithm in that the interactions between particles do not extend to infinity, but are local-

Figure 5.4: Interprocessor communication requirements

ized to some domain whose size depends upon the particular state of the solid/fluid. As such, it is very wasteful to use the long-range force technique, but the randomness of the interactions makes a mapping to a crystalline algorithm difficult.

To effectively address these issues, it seemed important to build a communication system that allowed messages to travel between nodes that were not necessarily connected by "channels", yet didn't need to involve all nodes collectively.

At this point, an enormous number of issues came up—routing, buffering, queueing, interrupts, etc. The first cut at solving these problems was a system that never acquired a real name, but was known by the name of its central function "`rdsort`" [Johnson:85a]. The basic concept was that a message could be sent from any node to any other node, at any time, and the receiving node would have its program *interrupted* whenever a message arrived. At this point, the user provided a routine called "`rdsort`" which, as its name implies, needed to read, sort and process the data.

While simple enough in principle, this programming model was never generally adopted (although it produced an effective solution to the melting problem). To users who came from a number-crunching physics background, the concept of "interrupts" was quite alien. Furthermore, the issues of sorting, buffering, mutual exclusion, etc. raised by the asynchronous nature of the resulting programs proved hard to code. Without debugging tools, it was extremely hard to develop programs using these techniques.

5.2.6 The Mark III

The advent of the Mark III machine [Peterson:85a] generated a rapid development in software modelling. In the previous five years, the crystalline system had shown itself to be a powerful tool for extracting maximum performance from the machines, but the new Mark III encouraged us to look at some of the "programmability" issues, which had previously been of secondary importance.

The first and most natural development was the generalization of the CrOS system for the new hardware [Johnson:86c], [Kolawa:86d]. Christened "CrOS III" it allowed us the flexibility of arbitrary message lengths (rather than multiples of the FIFO size) and hardware-supported broadcast, etc. All of these enhancements, however, maintained the original concept of nearest-neighbor communication supported by collective communication routines that operated throughout (or on a subset) the machine.

5.2.7 Host Programs

At this point, the programming model for the machines remained pretty much as it had been in 1982. A program running on the host computer loaded an application into the hypercube nodes, and then passed data back and forth with routines similar in nature to the `rdIH` and `wtIH` calls described earlier. This remained the only method through which the nodes could communicate with the outside world.

During the Mark II period, it had soon become apparent that most users were writing host programs that, while differing in detail, were identical in outline and function. An early effort to remove from the user the burden of writing yet another host program was a system called c3po [Meier:84b], which had a generic host program providing a shell in which subroutines could be executed in the nodes. Essentially, this model freed the user from writing an explicit host program, but still kept the locus of control in the host.

Cubix [Salmon:87a], however, reversed this. The basic idea was that the parallel computer, being more powerful than its host, should play the dominant role. Programs running in the nodes should decide for themselves what actions to take, and merely instruct the host machine to intercede on their behalf. If, for example, the node program wished to read from a file it should be able to tell the host program to perform the appropriate actions to open the file and read data, package it up into messages and transmit it

back to the appropriate node. This was a sharp contrast to the older method in which the user was effectively responsible for each of these actions.

The basic outcome was that the user's host programs were replaced with a standard "file-server" program called `cubix`. A set of library routines were then created with a single protocol for transactions between the node programs and `cubix`, which related to such common activities as opening, read and writing files, interfacing with the user, etc.

This change produced dramatic results. Finally, the node programs could contain calls to such useful functions as `printf`, `scanf` and `fopen`, which had previously been forbidden. Debugging was much easier, albeit in the old-fashioned way of "let's print everything and look at it." Furthermore, programs no longer needed to be broken down into "host" and "node" pieces and, as a result, parallel programs began to look almost like the original sequential programs.

Once file I/O was possible from the individual nodes of the machine, graphics soon followed through Plotix [Flower:86c], resulting in the development system shown in the heart of the family tree. The ideas embodied in this set of tools: Cros III, Cubix and Plotix still remain the basis for the most successful parallel programs.

5.2.8 A Ray Tracer—and an "Operating System"

While the radical change that led to Cubix was happening, the non-crystalline users were still developing alternative communication strategies. As mentioned earlier, `rdsort` never became popular due to the burden of bookkeeping that was placed on the user and the unfamiliarity of the interrupt concept.

The "9-routines" [Johnson:86a] attempted to alleviate the most burdensome issues by removing the interrupt nature of the system and performing buffering and queueing internally. The resultant system was a simple generalization of the `wtELT` concept, which replaced the "channel" argument with a processor number. As a result, messages could be sent to non-neighboring nodes. An additional level of sophistication was provided by associating a "type" with each message. The recipient of the messages could then sort incoming data into functional categories based on this type.

The benefit to the user was a simplified programming model. The only remaining problem was how to handle this new-found freedom of sending messages to arbitrary destinations.

We had originally planned to build a ray-tracer from the tools developed

while studying melting. There is, however, a fundamental difference between the melting process and the distributed database searching that goes on in a ray-tracer. (Ray-tracing is trivial if the whole model can be stored in each processor, but we were considering the case of a geometric database larger than this.)

Melting posed problems because the exact nature of the interaction was known only statistically—we might need to communicate with all processors up to two hops away from our node, or three hops or some indeterminate number. Other than this, however, the problem was quite homogeneous, and every node could perform the same tasks as the others.

The database search is inherently non-deterministic and badly load-balanced because it is impossible to map objects into the nodes where they will be used. As a result, database queries need to be directed through a tree structure until they find the necessary information and return it to the calling node.

A suitable methodology for performing this kind of exercise seemed to be a real multitasking system where "processes" could be created and destroyed on nodes in a dynamic fashion which would then map naturally onto the complex database search patterns. We decided to create an operating system.

The crystalline system had been described, at least in the written word, as an operating system. The concept of writing a real operating system with file systems, terminals, multitasking, etc., was clearly impossible while communication was restricted to single hops across hypercube channels. The new system, however, promised much more. The result was the "Multitasking, Object-oriented, Operating System", (MOOOS, commonly known as MOOSE) [Salmon:86a].

The basic idea was to allow for multitasking—running more than one process per node, with remote task creation, scheduling, semaphores, signals—to include everything that a real operating system would have. The implementation of this system proved quite troublesome and strained the capabilities of our compilers and hardware beyond their breaking points, but was nothing by comparison with the problems encountered by the users.

The basic programming model was of simple, "light-weight" processes communicating through message box/pipe constructs. The overall structure was vaguely reminiscent of the standard UNIX multiprocessing system, `fork/exec` and pipes. Unfortunately, this was by now completely alien to our programmers, who were all more familiar with the crystalline methods previously employed. In particular, problems were encountered with nam-

Figure 5.5: A "MOOSE" Process Tree

ing. While a process that created a child would automatically know its child's "ID," it was much more difficult for siblings to identify each other, and hence, to communicate. As a natural result, it was reasonably easy to build tree structures but difficult to perform domain decompositions. Despite these problems, useful programs were developed including the parallel ray-tracer with a distributed database that had originally motivated the design [Goldsmith:86a], [Goldsmith:87a].

An important problem was that architectures offered no memory protection between the lightweight processes running on a node. The user had to guess how much memory to allocate to each process, which complicated debugging when the user guessed wrong. Later, the INTEL iPSC implemented the hardware memory protection, which made life simpler [Koller:88b].

In using MOOSE, we wanted to explore dynamic load balancing issues. A problem with standard domain decompositions is that irregularities in the work loads assigned to processors lead to inefficiencies since the entire simulation, proceeding at lock-step, executes at the speed of the slowest node. The Crystal Router, developed at the same time as MOOSE, offered a simpler strategy.

5.2.9 The Crystal Router

By 1986, we began to classify the algorithms to generalize the performance models and identify applications that could be expected to perform well using the existing technology. This connection, led to the idea of "loosely synchronous" programming.

The central concept is one in which the nodes compute for a while, then synchronize and communicate, continually alternating between these two types of activity. This computation model was very well-suited to our crystalline communication system, which enforced synchronization automatically. In looking at some of the problems we were trying to address with our asynchronous communication systems (The 9 routines and MOOSE), we found that although the applications were not naturally loosely synchronous at the level of the individual messages, they followed the basic pattern at some higher level of abstraction.

In particular, we were able to identify problems in which it seemed that messages would be generated at a fairly uniform rate, but in which the moment when the data had to be physically delivered to the receiving nodes was synchronized. A load-balancer, for example, might use some type of simulated-annealing [Flower:87a] or neural-network [Fox:86i] approach to identify work elements that should be relocated to a different processor. As each decision is made, a message can be generated to tell the receiving node of its new data. It would be inefficient, however, to physically send these messages one at a time as the load-balancing algorithm progresses, especially since the results need only be acted upon once the load-balancing cycle has completed.

We developed the "crystal router" to address this problem [Fox:86b]. The idea was that messages would be buffered on their node of origin until a synchronization point was reached when a single system call sent every message to its destination in one pass. The results of this technology were basically twofold.

- Since the messages were accumulated locally and then sent *en masse*, much longer communication streams were generated than would be the case were each message sent individually. As a result, the effects of latency were minimized.

- The act of sending the messages involved all the nodes of the machine at once, so that messages could be routed to nodes other than those to which direct connections existed.

The resultant system had some of the attractive features of the 9 routines, in that messages could be sent between arbitrary nodes, but maintained the high efficiency of the crystalline system by performing all its internode communication synchronously.

5.2.10 DIME

The crystal router, and its various generalizations, enabled us to address a set of problems that had been too difficult to manage with purely local synchronous communication. In particular, the problems of irregularly shaped domains became tractable. The ultimate product in this regard is the DIME system—the "Distributed Irregular Mesh Environment" [Williams:88a].

This system is a library, which enables the user to construct and manipulate triangular meshes on arbitrary two-dimensional manifolds. An example is the flow of a fluid around a complex structure. This type of problem is difficult to tackle with standard domain decompositions because the irregular topology of the domain makes for poor (static) load balancing. Similarly, interesting problems in fluid flow evolve dynamically and require mesh refinement in order to achieve good numerical solutions. Again, this problem is hard to approach by conventional techniques since it may require refinement of the mesh in some areas, but not others, which leads to a dynamic load imbalance.

DIME resolves all of these issues by providing tools to create simple meshing of a region which the user can then load into a workstation or one of several types of parallel computers for refinement. As the mesh is refined to a useful granularity, dynamic load balancing is achieved using the crystal router. The user does not have to be concerned with the details of mesh creation and manipulation, but provides the physics behind the simulation. The system even takes care of such complex issues as displaying contour plots of physical quantities on the manifold.

5.2.11 Portability

In all of the software development cycles, one of our primary concerns was portability. Not only did we want our programs to be portable between various types of parallel computers, we also wanted them to be portable between parallel and sequential computers. It was in this sense that Cubix was such a breakthrough, since it allowed us to leave all the standard runtime system calls in our programs. In most cases, Cubix programs will run either on a supported parallel computer or on a simple sequential machine through the provision of a small number of dummy routines for the sequential machine. Using these tools, we were able to implement our codes on all of the commercially and locally built hypercubes.

The next question to arise, however, concerned possible extensions to

Figure 5.6: Express System Components

alternative architectures, such as shared-memory or mesh-based structures. The crystal router offered a solution. By design, messages in the crystal router can be sent to any other node. This step merely involves construction of a set of appropriate queues. When the interprocessor communication is actually invoked, the system is responsible for routing messages between processors—a step in which potential differences in the underlying hardware architecture can be concealed. As a result, applications using the crystal router can conceivably operate on any type of parallel or sequential hardware.

5.2.12 Express

At the end of 1987, ParaSoft Corporation was founded by a group from the hypercube project with the goal of providing a uniform software base—a set of portable programming tools—for all types of parallel processors.

The resultant system, Express [ParaSoft:88a], is a merger of the best of the C^3P ideas developed into a homogeneous system that can be supported on all types of parallel computers. The basic components are:

- a "long-range" message-passing system similar in concept to the 9 routines;
- extensions to the collective communication routines to support nonhypercube architectures;
- implementation of the gridmap decomposition tools for non-hypercube topologies;

- enhanced versions of the Cubix and Plotix subsystems to deal with a broader range of problem styles;
- an implementation of the crystal router for non-hypercube topologies; and
- an implementation of the crystalline communication system tailored to non-hypercube topologies.

Additionally, we added:

- a message-based multitasking and remote task creation system;
- support for non-homogeneous architectures in which the nodes of the parallel machine may be of different types, and are potentially located at different physical sites.

ParaSoft extended the parallel debugger originally developed for the NCUBE hypercube [Flower:87c] and created a set of powerful performance analysis tools [Parasoft:88f] to help users analyze and optimize their parallel programs. This toolset, incorporating all of the concepts of the original C^3P work and available on a wide range of parallel computers, has been widely accepted and is the most commonly used system at Caltech. It is interesting to note that the most successful parallel programs are still built around the crystalline style of internode communication originally developed for the Mark II hypercube in 1982. While other systems occasionally seem to offer easier routes to working algorithms, we usually find that a crystalline implementation offers significantly better performance.

At the current stage of development, we also believe that parallel processing is reasonably straightforward. The availability of sophisticated debugging tools, and I/O systems has resulted in several orders of magnitude reduction in debugging time. Similarly, the performance evaluation system has proved itself very powerful in analyzing areas where potential improvements can be made in algorithms.

The final challenge will be the automation of the parallelization process. Again, in this context, the work at Caltech has proven vital. The development of algorithms using the crystalline programming model is well understood and can be automated. As this article goes to print, we are testing a tool which automatically inserts the appropriate crystalline communication calls into an existing C program [Ikudome:90a].

5.2.13 Other Message-passing Systems

It is interesting to compare the work of other organizations with that performed at Caltech. In particular, our problem-solving approach to the art of parallel computing has, in some cases, led us down paths which we have since abandoned but which are still actively pursued by other groups. Yet, a completely fresh look at parallel programming methods may produce a more consistent paradigm than our evolutionary approach. In any case, the choice of a parallel programming system depends on whether the user is more interested in machine performance or ease of programming.

These are several commercial systems that offer some or all of the features of Express, based on long-range communication by message passing:

i) Mercury/Centaur

JPL developed this message-passing system [Lee:86a] at the same time as we developed the 9 routines at Caltech. Mercury is similar to the 9 routines in that messages can be transmitted between any pair of nodes, irrespective of whether or not a hypercube channel connects them. Messages also have "types" and can be sorted and buffered by the system as in the 9 routines or Express. A special type of message allows one node to broadcast to all others.

Centaur is a simulation of CrOS III built on Mercury. This system was designed to allow programmers with crystalline applications the ability to operate either at the level of the hardware (with the CrOS III library) or within the asynchronous Mercury programming model. When operating in Centaur mode, CrOS III programs may use advanced tools, such as the debugger, which require asynchronous access to the communication hardware.

ii) VERTEX

VERTEX is the native operating system of the NCUBE. It shares with Express, Mercury and the 9 routines the ability to send messages, with types, between arbitrary pairs of processors. Only two basic functions are supported to send and receive messages. I/O is not supported in the earliest versions of VERTEX, although this capability has been added in support of the second generation NCUBE hypercube.

iii) The Reactive Kernel

The Reactive Kernel [Seitz:88b] is a message-passing system based on the idea that nodes will normally be sending messages in response to messages coming from other nodes. Like all the previously mentioned systems, the Reactive Kernel can send messages between any pair of nodes with a simple send/receive interface. However, the system call that receives messages is incapable of distinguishing between incoming messages. All sorting and buffering must be done by the user.

iv) "NX"

The NX system provided for the INTEL iPSC series of hypercubes is also similar in functionality to the previously described long-range communication systems. It supports message types and provides sorting and buffering capabilities similar to those found in Express. No support is provided for nearest-neighbor communication in the crystalline style, although some of the collective communication primitives are supported.

v) MACH

The Mach operating system [Accetta:86a] is a full implementation of UNIX for a shared-memory parallel computer. It supports all of the normally expected operating system facilities, such as multiuser access, disks, terminals, printers, etc., in a manner compatible with the conventional Berkeley UNIX.

While this provides a strong basis for multiuser processing, it offers only simple parallel processing paradigms, largely based on the conventional UNIX interprocess communication protocols, such as "pipes" and "sockets". As mentioned earlier in connection with MOOSE, these types of tools are not the easiest to use in tightly coupled parallel codes.

vi) Helios

Helios [DSL:89a] is a distributed-memory operating system designed for transputer networks—distributed-memory machines. It offers typical UNIX-like utilities, such as compilers, editors, and printers, which are all accessible from the nodes of the transputer system, although less than are supported by Mach. In common with Mach, however, the level of parallel processing support is quite limited. Users are generally encouraged to use pipes for interprocessor communication—no collective or crystalline communication support is provided.

vii) Linda

The basic concept used in Linda [Ahuja:86a] is the idea of a tuple-space (database) for objects of various kinds. Nodes communicate by dropping objects into the database, which other nodes can then extract. This concept has a very elegant implementation, which is very simple to learn, but which suffers from quite severe performance problems. This is especially so on distributed-memory architectures, where the database searching needed to find an "object" can require intensive internode communication within the operating system.

More recent versions of Linda [Gelernter:89a] have extended the original concept by adding additional tuple-spaces and allowing the user to specify to which space an object should be sent and from which it should be retrieved. This new style is reminiscent of a mailbox approach, and is thus, quite similar to the general programming models used in CrOS III or Express.

5.2.14 What Did We Learn?

- The loosely synchronous programming model leads to programs that are developed, debugged, modelled, and perform extremely well on parallel machines.
- Using high-level and collective communication primitives simplifies coding for the user and allows implementors the flexibility to generate high performance on arbitrary architectures.
- A good parallel model of I/O and graphics leads to programs that are portable, efficient and easily understood. They also adapt well to special-purpose hardware.
- You don't need "a real operating system" to get high performance from parallel computers.
- Portability, programmability, and performance are the most important message-passing system qualities, and are not mutually exclusive.

5.2.15 Conclusions

The history of our message-passing system work at Caltech is interesting in that its motivation departs significantly from that of most other institutions.

Since our original goals were problem-oriented rather than motivated by the desire to do parallel processing research, we tended to build utilities that matched our hardware and software goals rather than for our aesthetic sense. If our original machine had had multiplexed DMA channels and specialized routing hardware, we might have started off in a totally different direction. Indeed, this can be seen as motivation for developing some of the systems described in the previous section.

In retrospect, we may have been lucky to have such limited hardware available, since it forced us to develop tools for the user rather than relying on an all-purpose communication system. The resultant decomposition and collective communication routines still provide the basis for most of our successful work—even with the development of Express, we still find that we return again and again to the nearest-neighbor, crystalline communication style, albeit using the portable Express implementation rather than the old `rdELT` and `wtELT` calls. Even as we attempt to develop automated mechanisms for constructing parallel code, we rely on this type of technology.

The advent of parallel UNIX variants has not solved the issues of message passing—indeed these systems are among the weakest in terms of providing user-level support for interprocessor communication. We continually find that the best performance, both from our parallel programs and also from the scientists who develop them, is obtained when working in a loosely synchronous programming environment such as Express, even when this means implementing such a system on top of a native, "parallel UNIX."

We believe that the work done by C^3P is still quite unique, at least in its approach to problem solving. It is amusing to recall the comment of one new visitor to Caltech who, coming from an institution building sophisticated "parallel UNIXes," was surprised to see the low level at which CrOS III operated. From our point of view, however, it gets the job done in an efficient and timely manner, which is of paramount importance.

5.3 Basic Tools — Debugging

*** Contribution needed from J. Flower

5.4 Basic Tools — Performance Monitoring

*** Contribution needed from J. Flower

Chapter 6

Synchronous Applications II

6.1 Convectively-Dominated Flows and the Flux-Corrected Transport Technique

This work implemented a code on the NCUBE-1 hypercube for studying the evolution of two-dimensional, convectively-dominated fluid flows. An explicit finite difference scheme was used that incorporates the flux-corrected transport (FCT) technique developed by Boris and Book [Boris:73a]. When this work was performed in 1986–1987, it was expected that explicit finite difference schemes for solving partial differential equations would run efficiently on MIMD distributed memory computers, but this had only been demonstrated in practice for "toy" problems on small hypercubes of up to 64 processors. The motivation behind this work was to confirm that a *bona fide* scientific application could also attain high efficiencies on a large commercial hypercube. In addition, the work also allowed the capabilities and shortcomings of the newly-acquired NCUBE-1 hypercube to be assessed.

6.1.1 An Overview of the FCT Technique

Although first-order finite difference methods are monotonic and stable, they are also strongly dissipative, causing the solution to become smeared out. Second-order techniques are less dissipative, but are susceptible to non-linear, numerical instabilities that cause non-physical oscillations in regions of large gradient. The usual way to deal with these types of oscillation is to incorporate artificial diffusion into the numerical scheme. However, if this is applied uniformly over the problem domain and enough is added to dampen

spurious oscillations in regions of large gradient, then the solution is smeared out elsewhere. This difficulty is also touched upon in Section 12.3.1. The FCT technique is a scheme for applying artificial diffusion to the numerical solution of a convectively-dominated flow problem in a spatially nonuniform way. More artificial diffusion is applied in regions of large gradient, and less in smooth regions. The solution is propagated forward in time using a second-order scheme in which artificial diffusion is then added. In regions where the solution is smooth, some or all of this diffusion is subsequently removed, so the solution there is basically second-order. Where the gradient is large, little or none of the diffusion is removed, so the solution in such regions is first-order. In regions of intermediate gradient, the order of the solution depends on how much of the artificial diffusion is removed. In this way, the FCT technique prevents non-physical extrema from being introduced into the solution.

6.1.2 Mathematics and the FCT Algorithm

The governing equations are similar to those in Section 12.3.1, namely, the two-dimensional Euler equations,

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}_x}{\partial x} + \frac{\partial \mathbf{F}_y}{\partial y} = \mathbf{S}(x, y) \tag{6.1}$$

where,

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix}, \quad \mathbf{F}_x = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho u(E + p/\rho) \end{bmatrix}, \quad \mathbf{F}_y = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho v(E + p/\rho) \end{bmatrix}, \quad \mathbf{S}(x, y) = \begin{bmatrix} 0 \\ b_x \\ b_y \\ b_x u + b_y v \end{bmatrix}$$

Here ρ is the fluid mass density, E is the specific energy, u and v are the fluid velocities in the x and y directions, b_x and b_y are body force components, and the pressure, p, is given by,

$$p = \rho(\gamma - 1)\left(E - \frac{u^2}{2} - \frac{v^2}{2}\right) \tag{6.2}$$

where γ is the constant adiabatic index. The motion of the fluid is tracked by introducing massless marker particles and allowing them to be advected with the flow. Thus, the number density of the marker particles, ρ_α, satisfies,

$$\frac{\partial \rho_\alpha}{\partial t} + \frac{\partial}{\partial x}(u\rho_\alpha) + \frac{\partial}{\partial y}(v\rho_\alpha) = 0 \tag{6.3}$$

The equations are solved on a rectilinear two-dimensional grid. Second-order accuracy in time is maintained by first advancing the velocities by a half time step, and then using these velocities to update all values for the full time step. The size of the time step is governed by the Courant condition.

The basic procedure in each time step is to first apply a five-point difference operator at each grid point to convectively transport the field values. These field values are then diffused in each of the positive and negative x and y directions. The behavior of the resulting fields in the vicinity of each grid point is then examined to determine how much diffusion to remove at that point. In regions where a field value is locally monotonic, nearly all the diffusion previously applied is removed for that field. However, in regions close to extrema, the amount of diffusion removed is less.

6.1.3 Parallel Issues

The code used in this study parallelizes well for a number of reasons. The discretization is static and regular, and the same operations are applied at each grid point, even though the evolution of the system is nonlinear. Thus, the problem can be load balanced at the start of the code by ensuring that each processor's rectangular subdomain contains the same number of grid points. In addition, the physics, and hence the algorithm, is local so the finite difference algorithm only requires communication between nearest-neighbors in the hypercube topology. The extreme regularity of the FCT technique means that it can also be efficiently used to study convective transport on SIMD concurrent computers, such as the Connection Machine, as has been done by Oran, *et al.* [Oran:90a].

No major changes were introduced into the sequential code in parallelizing it for the hypercube architecure. Additional subroutines were inserted to decompose the problem domain into rectangular subdomains, and to perform interprocessor communication. Communication is necessary in applying the Courant condition to determine the size of the next time step, and in transfering field values at grid points lying along the edge of a processor's subdomain. Single rows and columns of field values were communicated as the algorithm required. Some inefficiency, due to communication latency, could have been avoided if several rows and/or columns were communicated at the same time, but in order to avoid wasting memory on larger communication buffers this was not done. This choice was dictated by the small amount of memory (about 476 Kbytes) available on each NCUBE-1 processor.

Figure 6.1: The above figures show the development of the Kelvin-Helmholtz instability at the interface of two fluids in shear motion. In these figures, the density of the massless marker particles normalized by the fluid density is plotted on a grey scale, with black corresponding to a density of one and white to a density of 0. Initially, all the marker particles are in the upper half of the domain, and the fluids in the lower and upper half domains have a relative shear velocity in the horizontal direction. An 80×80 finite difference grid was used.

6.1.4 Example Problem

As a sample problem, the onset and growth of the Kelvin-Helmholtz instability was studied. This instability arises when the interface between two fluids in shear motion is perturbed, and for this problem the body forces, b_x and b_y are zero. As shown in Figure 6.1, vortices form along the interface and interact before being lost to numerical diffusion. By processing the output from the NCUBE-1, a videotape of the evolution of the instability was produced. This sample problem demonstrates that the FCT technique is able to track the physical instability without introducing numerical instability.

6.1.5 Performance and Results

The code was timed for the Kelvin-Helmholtz problem for hypercubes with dimension ranging from zero to nine. The results for the 512-processor case are presented in Table 6.1, and show a speedup of 429 for the largest problem size considered. Subsequently, a group at Sandia National Laboratories, using a modified version of the code, attained a speedup of 1009 on a 1024-processor NCUBE-1 for a similar type of problem [Gustafson:88a].

n_x	n_y	n	T_{512}	T_1	$\epsilon(\%)$	f	S
6	6	36	0.68	156.67	44.8	1.232	230
12	12	144	1.78	626.69	68.6	0.457	352
18	18	324	3.57	1410.05	77.2	0.296	395
24	24	576	6.07	2506.75	80.7	0.239	413
28	28	784	8.11	3411.97	80.7	0.239	421
30	30	900	9.26	3916.80	82.6	0.210	423
32	32	1024	10.47	4456.45	83.1	0.203	426
36	36	1296	13.14	5640.19	83.8	0.193	429

Table 6.1: Timing results in seconds for a 512-processor and a 1-processor NCUBE-1. n_x and n_y are the numbers of grid points per processor in the x and y directions. The concurrent efficiency, overhead, and speedup are denoted by ϵ, f, and S.

The definitions of concurrent speedup, overhead, and efficiency are given in Section 13.

An analytic model of the performance of the concurrent algorithm was developed, and ignoring communication latency, the concurrent overhead was found to be proportional to $1/\sqrt{n}$, where n is the number of grid points per processor. This is in approximate agreement with the results plotted in Figure 6.2, that shows the concurrent overhead for a number of different hypercubes dimensions and grain sizes.

6.1.6 Credits and References

The FCT code was ported to the NCUBE-1 by David W. Walker. Gary Montry of Sandia National Laboratories supplied the original code, and made several helpful suggestions. The videotape of the evolution of the Kelvin-Helmholtz instability was produced by Jeff Goldsmith at the Image Processing Laboratory of the Jet Propulsion Laboratory.

C^{3}P References: [Walker:88b]

Figure 6.2: Overhead, f, as a function of $1/\sqrt{n}$, where n is the number of grid points per processor. Results are shown for NCUBE-1 hypercubes of dimension one to nine. The overhead for the 2-processor case (open circles) lies below that for the higher dimensional hypercubes. This is because the processors only communicate in one direction in the 2-processor case, whereas for hypercubes of dimension greater than one, communication is necessary in both the x and y directions.

6.2 Magnetism in the High Temperature Superconductor Materials

6.2.1 Introduction

Following the discovery of high-temperature superconductivity, two-dimensional quantum antiferromagnetic spin systems have received enormous attention from the physicists worldwide. It is generally believed that high-temperature superconductivity occurs in the CuO planes, which is shown in Figure 6.3. Many features can be explained [Anderson:87a] in the Hubbard theory of the strongly coupled electron which at half-filling, is reduced to spin-1/2 antiferromagnetic Heisenberg model:

$$H = J \sum_{\langle ij \rangle} (S_i^x S_j^x + S_i^y S_j^y + S_i^z S_j^z) \tag{6.4}$$

where S^a are quantum spin operators. Furthermore, the neutron scattering experiments on the parent compound, La_2CuO_4, reveal a rich magnetic structure which is also modeled by this theory.

Physics in two dimensions (as compared to three dimensions) is characterized by the large fluctuations. Many analytical methods work well in

Figure 6.3: The copper-oxygen plane, where the superconductivity is generally believed to occur. The arrows denote the quantum spins. P_π, P_σ, $d_{x^2-y^2}$ denote the wave functions which lead to the interactions among them.

three dimensions, but fail in two dimensions. For the quantum systems, this means additional difficulties in find solutions to the problem.

New analytical methods have been developed to understand the low-T behavior of these two-dimensional systems, and progress had been made. These methods are essentially based on a $1/S$ expansion. Unfortunately, the extreme quantum case $S = \frac{1}{2}$ lies in the least reliable region of these methods. On the other hand, given sufficient computer power, Quantum Monte Carlo simulation [Ding:90g] can provide accurate numerical solutions about the model theory and quantitative comparison with the experiment (see Figure 6.4). Thus, the simulation becomes a crucial tool in studying these problems. The work described here has made a significant contribution to the understanding of high-T_c materials, and has been well received by the science community [Maddox:90a].

6.2.2 The Computational Algorithm

Using the Suzuki-Trotter transformation, the two-dimensional quantum problem is converted into three-dimensional classical Ising spins with complicated interactions. The partition function becomes a product of transfer

Figure 6.4: Inverse correlation length of La_2CuO_4 measured in neutron scattering experiment, denoted by cross; and those measured in our simulation, denoted by squares (units in $(1.178\text{Å})^{-1}$). $J = 1450\,\text{K}$. At $T \approx 500\,\text{K}$, La_2CuO_4 undergoes a structural transition. The curve is the fit shown in Figure 6.10.

matrices for each four-spin interaction

$$W = \begin{vmatrix} e^K & 0 & 0 & 0 \\ 0 & e^{-K}ch(2K) & e^{-K}sh(2K) & 0 \\ 0 & e^{-K}sh(2K) & e^{-K}ch(2K) & 0 \\ 0 & 0 & 0 & e^K \end{vmatrix}$$

with $K = \beta/4m$. These four-spin squares go in the time direction on the three-dimensional lattice. This transfer matrix serves the probability basis for a Monte Carlo simulation. The zero elements are the consequence of the quantum conservation law. To avoid generating trial configurations with zero probability, thus wasting the CPU time because these trials will never be accepted, one should have the conservation law built into the updating scheme. Two types of local moves may locally change the spin configurations, as shown in Figure 6.5. A global move in the time direction flips all the spins along this time-line. This update changes the magnetization. Another global move in spatial directions changes the winding numbers.

This classical spin system in three dimensions is simulated using the Metropolis Monte Carlo algorithm. Starting with a given initial configuration, we locate a closed loop C of L spins, in one of the four moves. After checking that they satisfy the conservation law, we compute the probability before all L spins are flipped, P_i which is a product of the diagonal elements

Figure 6.5: (a) A "time-flip." The dashed 3×1 plaquette is a non-interacting one. The eight plaquettes surrounding it are interacting ones. (b) A "space-flip." The dashed plaquette is a non-interacting one lying in spatial dimensions. The four plaquettes going in time direction are interacting ones.

of the transfer matrix, the probability after the spins are flipped, and P_f which is a product of the off-diagonal elements of the transfer matrix along the loop C. The Metropolis procedure is to accept the flip according to the probability $P = P_f/P_i$.

We implemented a simple and efficient multi-spin coding method which facilitates vectorization and saves index calculation and memory space. This is possible because each spin only has two states, up (1) or down (0), which is represented by a single bit in a 32-bit integer. Spins along the t-direction are packed into 32-bit words, so that the boundary communication along the x or y direction can be handled more easily. All the necessary checks and updates can be handled by the bit-wise logical operations OR, AND, NOT, XOR. Note that this is a natural vectorization since AND operations for the 32 spins are carried out in the single AND operation by the CPU. The index calculations to address these individual spins are also minimized, because one only computes the index once for the 32 spins. The same principles are applied for both local and global moves. Figure 6.6 shows the case for time-loop coding.

6.2.3 Parallel Implementation and Performance

The fairly large three-dimensional lattices (usually $128 \times 128 \times 192$) is partitioned into a ring of M processors with x-dimension which is uniformly dis-

Figure 6.6: A vectorization of eight "time-flips." Spins along the t-direction are packed into computer words. The two 32-bit words, S1 and S2, contain eight "time plaquettes," indicated by the dashed lines.

tributed among the M processors. The local updates are easily parallelized since the connection is, at most, next-nearest neighbor (for the time-loop update). The needed spin-word arrays from its neighbor are copied into the the local storage by the *shift* routine in the CROS communication system [Fox:88a] before doing the update. One of the global updates, the time-line, can also be done in the same fashion. The communication is very efficient in the sense that a single communication shift, $Ny * Nt$, spins instead of Nt spins in the case where the lattice is partitioned into a two-dimensional grid. The overhead/latency associated with the communication is, thus, significantly reduced.

The winding-line global update along the x-direction is difficult to do in this fashion, because it involves spins on all the M nodes. In addition, we need to compute the correlation functions which have the same difficulty. However, since these operations are not used very often, we devised a fairly elegant way to parallelize these global operations. A set of *gather-scatter* routines, based on the *cread* and *cwrite* in CROS, is written. In *gather*, the subspaces on each node are gathered into complete spaces on each node, preserving the original geometric connection. Parallelism is achieved now since the global operations are done on each node just as in the sequential computer, with each node only doing the part it originally covers. In *scatter*, the updated (changed) lattice configuration on a particular node (number zero) is scattered (distributed) back to all the node in the ring, exactly according to the original partition. Note that this scheme differs from the

Figure 6.7: The configuration of the hypercube nodes. In the example, 32 nodes are configured as four independent rings, each consisting of eight nodes. Each ring does an independent simulation.

earlier decomposition scheme [Fox:84a] for the gravitation problem, where memory size constraint is the main concern.

The hypercube nodes were divided into several *independent* rings, each ring holding an independent simulation, as shown in Figure 6.7. At higher temperatures, a spin system of 32×32 is enough, so that we can simulate several independent systems at the same time. At low temperatures, one needs larger systems, such as 128×128—all the nodes will then be dedicated to a single large system. This simple parallelism makes the simulation very flexible and efficient. In the simulation, we used a parallel version of the Fibonacci additive random numbers generator [Ding:88d], which has a period larger that 2^{127}.

We have made a systematic performance analysis by running the code on different sizes and different numbers of nodes. The timing results for a realistic situation (20 sweeps of update, one measurement) are measured [Ding:89d]. The *speedup*, t_1/t_M, where t_1 (t_M) is the time for the same size spins system to run same number operations on one (M) node, are plotted in Figure 6.8. One can see that speedup is quite close to the ideal case, denoted by the dashed line. For the 128×128 quantum spin system, the 32-node hypercube speedup computation by a factor of 26.6, which is a very good result. However, running the same spin system on a 16-node is more efficient, because we can run two independent systems on the 32-node hypercube with a total speedup of $2 \times 14.5 = 29$ (each speedup a factor 14.5). This is better described by *efficiency*, defined as speedup/nodes, which is

Figure 6.8: Speedup of the parallel algorithm for lattice systems 64×64, 96×96 and 128×128. The dashed line is the ideal case.

Figure 6.9: Efficiency of the parallel algorithm.

plotted in Figure 6.9. Clearly, the efficiency of the implementation is very high, generally over 90%.

Comparison with other supercomputers are interesting. For this program, the CRAY X-MP speed is approximately two-node. This indicates that our 32-node Mark IIIfp performs better than the CRAY X-MP by about a factor of (32/2)*90% = 14! We note that our code is written in "C" and the vectorization is limited to the 32-bit inside the words. Rewriting the code in Fortran (Fortran compilers on the CRAY are more efficient) and fully vectorizing the code, one may gain a factor of about five on the CRAY. Nevertheless, this quantum Monte Carlo code is clearly a good example, in that parallel computers easily (*i.e.*, at same programming level) outperform

the conventional supercomputers.

6.2.4 Physics Results

We obtained many good results which were previously unknown. Among them, the correlation functions are perhaps the most important. First, the results can be directly compared with experiments, thus, providing new understandings to the magnetic structure of the high temperature superconducting materials. Second, and no less important, is the behavior of the correlation function we obtained which gives a crucial test of the assessment of various approximate methods.

In the large spin-S (classical) system, the correlation length goes as

$$\xi = A \cdot \exp(2\pi S^2 J/T) \tag{6.5}$$

at low temperatures. This predicts a too large correlation length, compared with experimental results. As $S \rightarrow \frac{1}{2}$, the quantum fluctuations in the system become significant. Several approximate methods [Chakravarty:88a], [Auerbach:88a] predict a similar low-T behavior. $\xi = (A/T^p) \cdot \exp(2\pi\rho_s/T)$, $\rho_s = Z_\xi^S S(S+1)J$, and $p = 0$ or 1. $Z_\xi^S \leq 1$ is a quantum renormalization constant.

Our extensive quantum Monte Carlo simulations were performed [Ding:90g] on the spin-$\frac{1}{2}$ system as large as 128×128 at low temperature range $T/J = 0.27$–1.0. The correlation length, as a function of $1/T$, is plotted in Figure 6.10. The data points fall onto a straight line, surprisingly well, throughout the whole temperature range, leading naturally to the pure exponential form:

$$\xi(T)/a = 0.276 e^{1.25J/T} \tag{6.6}$$

where a is the lattice constant. This provides a crucial support to the above mentioned theories. Quantitatively,

$$Z_\xi^{1/2} = 0.265(2) \tag{6.7}$$

or

$$\rho_s = Z_\xi^S S(S+1)J = 0.199(2)J. \tag{6.8}$$

Direct comparison with experiments will not only test the validity of the Heisenberg model, but also determine the important parameter, the exchange coupling J. The spacing between Cu atoms in CuO plane is $a = 3.79$ Å. Setting $J = 1450$ K, the Monte Carlo data is compared with

Figure 6.10: Correlation length measured at various temperatures. The straight line is the fit.

those from neutron scattering experiments [Endoh:88a] in Figure 6.4. The agreement is very good. This provides strong evidence that the essential magnetic behavior is captured by the Heisenberg model. The quantum Monte Carlo result is an accurate first principle calculation, no adjustable parameter is involved. Comparing directly with the experiment, the only adjustable parameter is J. This gives an independent determination of the *effective* exchange coupling:

$$J = 1450 \pm 30\,\mathrm{K}. \tag{6.9}$$

Note that near $T_N \simeq 200\,\mathrm{K}$, the experimentally measured correlation is systematically smaller than the theoretical curve, Equation 6.4. This is a combined result of small effects: frustration, anisotropies, inter-layer coupling, etc.

Various moments of the Raman spectrum are calculated using series expansions and compared with experiments [Singh:89a]. This gives an estimate: $J = 1030 \pm 50\mathrm{cm}^{-1}$ ($1480 \pm 70\,\mathrm{K}$), which is quite close to the above value determined from correlation functions. Raman scattering probes the short wavelength region, whereas neutron scattering measures the long-range correlations. The agreement of J's obtained from these two rather different experiments is another significant indication that the magnetic interactions are dominated by the Heisenberg model.

Equation 6.4 is valid for all the quantum AFM spins. The classic two-dimensional antiferromagnetic system discovered twenty years ago [Birgeneau:71a], K_2NiF_4, is a spin-one system with $J = 104\,\mathrm{K}$. Very re-

Figure 6.11: Correlation length of $K_2 Ni F_4$ measured in neutron scattering experiment with the fit.

cently, Birgeneau [Birgeneau:90a] fitted the measured correlation lengths to

$$\frac{\xi(T)}{a} = \frac{0.123 e^{5.31J/T}}{(1 + T/5.31J)} \tag{6.10}$$

The fit is very good, as shown in Figure 6.11. The factor $(1 + T/5.31J)$ comes from integration of the two-loop β-function without taking the $T \rightarrow 0$ limit, and could be neglected if T is very close to 0. For the spin-$\frac{5}{2}$ AFM $Rb_2 Mn_x Cr_{1-x} Cl_4$, Equation 6.4 also describes the data quite well [Higgins:88a].

A common feature from Figure 6.10 and Figure 6.11 is that the scaling equation Equation 6.4, which is derived near $T_c = 0$, is valid for a wide range of T, up to $T \sim 2J$. This differs drastically from the range of criticality in three-dimensional systems, where the width $t \equiv (T - T_c)/T_c$ is usually about 0.2 or less. This is a consequence of the crossover temperature T_{cr} [Chakravarty:88a], where the Josephson length scale becomes compatible with the thermal wave length, being relatively high, $T_{cr} \sim J$. This property is a general character in the low critical dimensions. In the quantum XY model, a Kosterlitz-Thouless transition occurs [Ding:90b] at $kT_c/J = 0.350$ and the critical behavior remains valid up to $kT/J = 0.7$.

As emphasized by Birgeneau, the spin wave value

$$2\pi\rho_s = 2\pi J S^2 (1 + 0.158/2S)^2 (1 - 0.552/2S) \tag{6.11}$$

$S = 1$, $2\pi\rho_s = 5.30$, fits the experiment quite well, whereas for $S = 1/2$, spin wave value $2\pi\rho_s = 0.944$ differs significantly from the correct value 1.25

Figure 6.12: Energy measured as a function of temperature. Squares are from our work. The curve is the 10th order high-T expansion.

as in Equation 6.4. This indicates that the large quantum fluctuations in the spin-$\frac{1}{2}$ system are not adequately accounted for in the spin wave theory, whereas for the spin-one system, they are.

Figure 6.12 shows the energy density at various temperatures. At higher T, the high temperature series expansion accurately reproduces our data. At low T, E approaches a finite ground state energy. Another useful thermodynamical quantity is uniform susceptibility, which is shown in Figure 6.13. Again, at high-T, series expansion coincides with our data. The maximum point occurs at $kT/J = 0.93$ with $\chi_{\max} J/N_0 g^2 \mu_B = 2.825(2)$. This is useful in determining J and N_0 for the material.

6.2.5 Remarks

In conclusion, the quantum AFM Heisenberg spins are now well understood theoretically. The data from neutron scattering experiments for both $S = 1/2$, La_2CuO_4 and $S = 1$, K_2NiF_4 compare quite well. For La_2CuO_4, this leads to a direct determination $J = 1450 \pm 30K$.

Quantum spins are well suited for hypercube computers. Its spatial decomposition is straight-forward; the short range nature (excluding the occasional long range ones) of interaction makes the extension to large numbers of processors simple. Hypercube connections made the use of the node computer efficient and flexible. High speedup can be achieved with reasonable ease, provided one improves the algorithm to minimize the communications.

Figure 6.13: Uniform susceptibility measured as a function of temperature. Symbols are similar to Figure 6.12.

6.2.6 Credits

The work described here is the result of the collaboration between H. Q. Ding and M. S. Makivic.

6.3 Phase Transitions in Two-dimensional Quantum Spin Systems

In this section, we discuss two further important developments based on the previous section (Section 6.2) on the isotropic Heisenberg quantum spins. These extensions are important to treat the observed phase transitions in the two-dimensional magnetic systems. Theoretically, two-dimensional isotropic Heisenberg quantum spins remain in paramagnetic state at all temperatures [Mermin:66a]. However, all crystals found in nature with strong two-dimensional magnetic characters go through phase transitions into ordered states [Birgeneau:71a], [DeJongh:74a]. These include the recently discovered high-T_c materials, La_2CuO_4 and $YBa_2Cu_3O_6$, despite the presence of large quantum fluctuations in the spin-$\frac{1}{2}$ antiferromagnets.

We consider the cases where the magnetic spins interact through

$$H = \sum_{[ij]} JS_i \cdot S_j + hS_i^z S_j^z \tag{6.12}$$

In the case $h \ll J$, the system goes through an Ising-like antiferromagnetic transition, very similar to those that occur in the high-T_c materials. In the

case $h = -J$, *i.e.*, the XY model, the system exhibits a Kosterlitz-Thouless type of transition. In both cases, our simulation provides convincing and complete results for the first time. These results have broad implications in two-dimensional physical systems in particular, and the statistical systems in general.

6.3.1 The case of $h \ll J$: Antiferromagnetic Transitions

The popular explanation for the antiferromagnetic ordering transitions in these high-T_c materials emphasizes the very small coupling, J', between the two-dimensional layers, J'/J, and is estimated to be about 10^{-5}. However, all these systems exhibit some kind of in-plane anisotropies, which is of order 10^{-3}. An interesting case is the spin-one crystal, $K_2 Ni F_4$, discovered twenty years ago [Birgeneau:71a]. The magnetic behavior of $K_2 Ni F_4$ exhibits very strong two-dimensional characters with an exchange coupling $J = 104\,\mathrm{K}$. It has a Néel ordering transition at $T_N = 97K$, induced by an Ising-like anisotropy, $h^A \approx 0.002$.

Our simulation provides clear evidence to support the picture that the in-plane anisotropy is also quite important in bringing about the observed antiferromagnetic transition at the most interesting spin-$\frac{1}{2}$ case. Adding an anisotropy energy as small as $h^A = 0.0025$ will induce an ordering transition at $T_c/kJ = 0.295$. This striking effect and related results agree well with a wide class of experiments, and provides some insights into these types of materials.

Origin of the Interaction

In the antiferromagnetic spin system, superexchange lead to the dominant isotropic coupling. One of the high-order effects, due to crystal field, is written as $-DS_z^2$, which is a constant for these spin-$\frac{1}{2}$ high-T_c materials. Another second order effect is the spin-orbital coupling. This effect will pick up a preferred direction and lead to a $S_i^z S_j^z$ term, which also arises due to the lattice distortion in $La_2 Cu O_4$. More complicated terms, like the antisymmetric exchange, can also be generated. For simplicity and clarity, we focus the study on the antiferromagnetic Heisenberg model with an Ising-like anisotropy as in Equation 6.12. The anisotropy parameter h relates to the usual reduced anisotropy energy h^A through $h^A = h/4J$. In the past, the anisotropy field model, $\Sigma\, \epsilon_i\, H_A S_i^z$, is also measured. However, its origin is less clear and, furthermore, the Ising symmetry is explicitly broken.

Figure 6.14: (a) The specific heat for different size systems of $h = 1$. (b) Finite size scaling for $T_c(L) - T_c \propto L^{-1}$.

Simulation Results

For the large anisotropy system, $h = 1$, the specific heat C_V are shown for several spin systems in Figure 6.14(a). The peak becomes sharper and higher as the system size increases, indicating a divergent peak in an infinite system, similar to the two-dimensional Ising model. Defining the transition temperature $T_c(L)$ at the peak of C_V for the finite $L \times L$ system, the finite size scaling theory [Landau:76a] predicts that $T_c(L)$ relates to T_c through the scaling law

$$T_c(L) - T_c \propto L^{-\nu}. \tag{6.13}$$

Setting $\nu = 1$, the Ising exponent, a good fit with $T_c = 1.063 \pm 0.003$ is shown in Figure 6.14(b). A different scaling with the same exponent for the correlation length,

$$\xi \propto (T - T_c)^{-\nu}, \tag{6.14}$$

is also satisfied quite well, resulting in $T_c = 1.05 \pm 0.01$. The staggered magnetization drops down near T_c, although the behaviors are rounded off on these finite sizes systems. All the evidence clearly indicates that an Ising-like antiferromagnetic transition occurs at $T_c = 1.06$, with a divergent specific heat. In the smaller anisotropy case, $h = 0.1$, similar behaviors are found. The scaling for the correlation length is shown in Figure 6.15, indicating a transition at $T_c = 0.44$. However, the specific heat remains finite at all temperatures.

The most interesting case is $h = 0.01$ (or $h^4 = 0.0025$, very close to those in K_2NiF_4 [Birgeneau:71a]). Figure 6.16 shows the staggered cor-

Figure 6.15: The inverse correlation lengths for $h = 0.1$ system ($\diamond$), for $h = 0.01$ system ($\sqcap$), and for $h = 0$ system ($\times$) for the purpose of comparison. The straight lines are the scaling relation: $\xi^{-1} \propto T - T_c$. From it we can pin down T_c.

relation function at $T = 0.3$ compared with those on the isotropic model [Ding:90g]. The inverse correlation length measured, together with those for the isotropic model ($h = 0$) are shown in Figure 6.15. Below $\xi^{-1} \approx 0.1$, the Ising behavior of a straight line becomes clear. Clearly, the system becomes antiferromagnetically ordered around $T = 0.3$. The best estimate is

$$T_c = 0.295, \quad h = 0.01. \tag{6.15}$$

Theoretical Interpretation

It may seem a little surprising that a very small anisotropy can lead to a substantially high T_c. This may be explained by the following argument. At low T, the spins are highly correlated in the isotropic case. Since no direction is preferred, the correlated spins fluctuate in all directions, resulting in zero net magnetization. Adding a very small anisotropy into the system introduces a preferred direction, so that the already highly correlated spins will fluctuate around this direction, leading to a global magnetization.

More quantitatively, the crossover from the isotropic Heisenberg behavior to the Ising behavior occurs at T_{cr}, where the correlation length is of order of some power of the inverse anisotropy. From the scaling arguments [Riedel:69a], $\xi \sim h^{-\nu/\phi} \approx h^{-1/2}$ where ϕ is the crossover exponent. In the

Figure 6.16: The correlation function on the 96×96 system at $T = 0.3$ for $h = 0.01$ system. It decays with correlation length $\xi \approx 120$. Also shown is the isotropic case $h = -$, which has $\xi = 17.5$.

two-dimensional model, both ν and ϕ are infinite, but the ratio is approximately 1/2. For $h = 0.01$, this relation indicates that the Ising behavior is valid for $\xi^{-1} \leq 0.1$, which is clearly observed in Figure 6.15. Similar crossover around $\xi^{-1} \approx 0.3$ for $h = 0.1$ is also observed in Figure 6.15. At low T, for the isotropic quantum model, the correlation length behaves as [Ding:90g] $\xi \sim e^{2\pi\rho/T}$ where $\rho = Z_{\xi}^{(S)} \cdot S(S+1)$. Therefore, we expect

$$T_c \approx \frac{Z^{(S)} S(S+1)}{\log(h^{-1})} \tag{6.16}$$

where $Z^{(S)}$ is spin-S dependent constant of order one. Therefore, even a very small anisotropy (h) will induce a phase transition at a substantially high temperature ($T_c \gg h$). This crude picture, suggested a long time ago to explain the observed phase transitions, is now confirmed by the extensive quantum Monte Carlo simulations for the first time. Note that this problem is an extreme case both because it is an antiferromagnet (more difficult to become ordered than the ferromagnet) and because it has the largest quantum fluctuations (spin-$\frac{1}{2}$). Since $\log(h^{-1})$ varies slowly with h, we can estimate $Z^{(S)}$ at $h = 0.01$:

$$Z^{(1/2)} \simeq 1.9. \tag{6.17}$$

Comparison with Experiments

This simple result correctly predicts T_c for a wide class of crystals found in Nature, assuming the same level of anisotropy, *i.e.*, $h^A \sim 0.002$. The high-T_c superconductor $Y\,Ba_2\,Cu_3\,O_{6.1}$ exhibits a Néel transition at $T_N = 435$ K. With $J \approx 1400$ K, our results give quite a close estimate: $T_c = 420$ K. Similar close predictions hold for other $S = 1/2$ systems, such as superconductor $Er\,Ba_2\,Cu_3\,O_7$ and insulator $K_2\,Co\,F_4$. For the high-T_c material $La_2\,Cu\,O_4$, $J = 1450$ K [Ding:90g]. This material undergoes a Néel transition at $T_N \simeq 220$ K. Our prediction of T_c=428 K is in the same range of T_N (comparing to the naive expectation that $T_c \sim h \sim 10$ K). In this crystal, there is some degree of frustration (see below), so the actual transition is pushed down. These examples clearly indicate that the in-plane anisotropy could be quite important to bring the system to the Néel order for these high-T_c materials. For the $S = 1$ system, $K_2\,Ni\,F_4$, our results predict a $T_c = 81$ K, quite close to the observed $T_N = 97$ K.

These results have direct consequences regarding the critical exponents. The onset of transition is entirely due to the Ising like anisotropy. Once the system becomes Néel ordered, different layers in the three-dimensional crystals will order at the same time. Spin fluctuations, in different layers, are incoherent so that the critical exponents such as β, γ, and ν will be the two-dimensional Ising exponents instead of three-dimensional Ising exponents. $Er\,Ba_2\,Cu_3\,O_7$ and $K_2\,Co\,F_4$ show clearly such behaviors. However, the interlayer coupling, *albeit* very small (much smaller than the in-plane anisotropy), could induce coherent correlations between the layers, so that the critical exponents will be somewhere in between the two-dimensional and the three-dimensional Ising exponents. $La_2\,Cu\,O_4$ and $Y\,Ba_2\,Cu_3\,O_6$ seem to belong to this category.

Whether the ground state of the spin-$\frac{1}{2}$ antiferromagnet spins has the long-range Néel order, is a longstanding problem [Anderson:87a]. The existence of the Néel order is vigorously proved for $h \geq 0.78$. In the most interesting case ($h = 0$), numerical calculations on small lattices suggested the existence of the long-range order. Our simulation establishes the long-range order for $h \geq 0.01$.

The fact that near T_c, the spin system is quite sensitive to the tiny anisotropy could have a number of important consequences. For example, the correlation lengths measured in $La_2\,Cu\,O_4$ are systematically smaller than the theoretical prediction [Ding:90g] near T_c. The weaker correlations probably indicate that the frustrations, due to the next near neighbor inter-

action, come into play. This is consistent with the fact that T_N is below the T_c suggested by our results.

6.3.2 The Case of $h = -J$: Quantum XY Model and the Topological Transition

It is well known now that the two-dimensional (2D) classical (planar) XY model undergoes Kosterlitz-Thouless (KT) [Kosterlitz:73a] transition at $kT_c/J = 0.898$ [Gupta:88a], characterized by exponentially divergent correlation length and in-plane susceptibility. The transition, due to the unbinding of vortex-antivortex pairs, is weak; the specific heat has a finite peak above T_c.

Does the two-dimensional quantum XY model go through a phase transition? If yes, what is the type of the transition? This is a longstanding problem in statistical physics. The answers are relevant to a wide class of two-dimensional problems such as magnetic insulators, superfluidity, melting, and possibly to the recently discovered high-T_c superconducting transition. Physics in two dimensions is characterized by large fluctuations. Changing from the classical model to the quantum model, additional quantum fluctuations (which are particularly strong in the case of spin-1/2) may alter the physics significantly. A direct consequence is that the already weak KT transition could be washed out completely.

A Brief History

The quantum XY model was first proposed [Matsubara:56a] in 1956 to study the lattice quantum fluids. Later, high temperature series studies raised the possibility of a divergent susceptibility for the two-dimensional model. For the classical planar model, the remarkable theory of Kosterlitz and Thouless [Kosterlitz:73a] provided a clear physical picture and correctly predicted a number of important properties. However, much less is known about the quantum model. In fact, it has been controversial. Using a large-order high-temperature expansion, Rogiers, *et al.* [Rogiers:79a] suggested a second order transition at $kT_c/J = 0.39$ for spin-1/2. Later, real space renormalization group analysis was applied to the model with results contradictory and inconclusive. DeRaedt, *et al.* [DeRaedt:84a] then presented an exact solution and Monte Carlo simulation both based on the Suzuki-Trotter transformation with small Trotter number m. Their results, both analytical and numerical, supported an *Ising*-like (second order) transition at the Ising

point $kT_c/J = 1/2\log(1+\sqrt{2}) = 0.567$, with a logarithmically divergent specific heat. Loh, *et al.* [Loh:85a] simulated the system with an improved technique. They found that specific peak remains finite and argued that a phase transition occurs at $T_c = 0.4$–0.5 by measuring the change of the "twist energy" from the 4×4 lattice to the 8×8 lattice. The dispute between DeRaedt, *et al.*, and Loh, *et al.*, centered on the importance of using a large Trotter number m and the global updates in small size systems, which move the system from one subspace to another. Recent attempts to solve this problem still add fuel to the controversy.

Evidence for the Transition

The key to pin down the existence and the type of transition is a study of correlation length and in-plane susceptibility, because their divergences constitute the most direct evidence of a phase transition. These quantities are much more difficult to measure, and large lattices are required in order to avoid finite size effects. These key points are lacking in the previous works. and are the focus of our study. By extensive use of the Mark IIIfp Hypercube, we are able to measure spin correlations and thermodynamic quantities accurately on very large lattices (96×96). Our work provides convincing evidence that a phase transition does occur at a finite temperature in the extreme quantum case, spin-1/2. At transition point, $kT_c/J = 0.350 \pm 0.004$, the correlation length and susceptibility diverge exactly according to the form of Kosterlitz-Thouless (see Equation 6.18).

We plot the correlation length, ξ, and the susceptibility, χ, in Figure 6.17 and Figure 6.18. They show a tendency of divergence at some finite T_c. Indeed, we fit them to the form predicted by Kosterlitz and Thouless for the classical model

$$\xi(T) = Ae^{B/(T-T_c)^\nu}, \quad \nu = \frac{1}{2} \tag{6.18}$$

The fit is indeed very good (χ^2 per degree of freedom is 0.81), as shown in Figure 6.17. The fit for correlation length gives

$$A_\xi = 0.27(3), \quad B_\xi = 1.18(6), \quad T_c = 0.350(4). \tag{6.19}$$

A similar fit for susceptibility, χ is also very good ($\chi^2/\mathrm{DOF} = 1.06$):

$$A_\chi = 0.060(5), \quad B_\chi = 2.08(6), \quad T_c = 0.343(3) \tag{6.20}$$

as shown in Figure 6.18. The good quality of both fits and the closeness of T_c's obtained are the main results of this work. The fact that these fits also

Figure 6.17: Correlation length and fit. (a) xi vs. T. The vertical line indicates ξ diverges at T_c; (b) $\log(\xi)$ vs. $(T - T_c)^{-1/2}$. The straight line indicates $\nu = 1/2$.

reproduce the expected scaling behavior $\chi \propto \xi^{2-\eta}$ with

$$\eta = 2 - B_\chi / B_\xi = 0.24 \pm 0.10 \tag{6.21}$$

is a further consistency check. These results strongly indicate that the spin-1/2 XY model undergoes a Kosterlitz-Thouless phase transition at $T_c = 0.350 \pm 0.004$. We note that this T_c is consistent with the trend of the "twist energy" [Loh:85a] and that the rapid increase of vortex density near $T = 0.35 - 0.40$ is due to the unbinding of vortex pairs. Figure 6.17 and Figure 6.18 also indicate that the critical region ΔT is quite wide ($\sim T_c$), which is very similar to the spin-1/2 Heisenberg model, where the $T \rightarrow 0$ behavior holds up to $T \sim 2J$. These two-dimensional phenomena are in sharp contrast to the usual second-order transitions in three dimensions.

The algebraic exponent η is consistent with the Ornstein-Zernike exponent $(d-1)/2 = 1/2$ at higher T. As $T \rightarrow T_c$, η shifts down slightly and shows signs of approaching $1/4$, the value at T_c for the classical model. This is consistent with Equation 6.21.

We measured energy and specific heat, C_V (for $T \leq 0.41$ we used a 32×32 lattice). The specific heat is shown in Figure 6.19. We found that C_V has a peak above T_c, at around $T = 0.45$. The peak clearly shifts away from $T = 0.52$ on the much smaller 8×8 lattice. DeRaedt, *et al.* [DeRaedt:84a] suggested a logarithmic divergent C_V in their simulation, which is likely an artifact of their small m values. One striking feature in Figure 6.19 is a very steep increase of C_V at $T \approx T_c$. The shape of the curve is asymmetric near

Figure 6.18: Susceptibility and fit.

Figure 6.19: Specific heat C_V. For $T < 0.41$, the lattice size is 32×32.

the peak. These features of the C_V curve differ from that in the classical XY model [Gupta:88a].

Implications

Quantum fluctuations are capable of pushing the transition point from $T_c = 0.898$ in the classical model, down to T_c=0.35 in the quantum spin-1/2 case, although not strong enough to push it down to zero. They also reduced the constant B_ξ from 1.67 in the classical case to 1.18 in the spin-1/2 case.

The critical behavior in the quantum case is of KT-type, as in the classical case. This is a little surprising, considering the differences regarding the spin space. In the classical case, the spins are confined to the X-Y plane (thus the

model is conventionally called a "planar rotator" model). This is important for the topological order in KT theory. The quantum spins are not restricted to the X-Y plane, due to the presence of S^z for the commutator relation. The KT behavior found in the quantum case indicates that the extra dimension in the spin space (which does not appear in the Hamiltonian) is actually unimportant. These correlations are very weak and short-ranged. The out-of-plane susceptibility remains a small quantity in the whole temperature range.

These results for the XY model, together with those on the quantum Heisenberg model, strongly suggest that although quantum fluctuations at finite T can change the quantitative behavior of these non-frustrated spin systems with *continuous* symmetries, the qualitative picture of the classical system persists. This could be understood following universality arguments that, near the critical point, dominant behavior of the system is determined by long wavelength fluctuations which are characterized by symmetries and dimensionality. The quantum effects only change the short range fluctuations which, after integrated out, only enter as renormalization of the physical parameters, such as B_ξ.

Our data also show that, for the XY model, the critical exponents are spin-S independent, in agreement with universality. More specifically, ν in Equation 6.18 could, in principle, differ from its classical value 1/2. Our data are sufficient to detect any systematic deviation from this value. For this purpose, we plotted ξ in Figure 6.17(b), using $\log(\xi)$ vs. $(T - T_c)^{-1/2}$. As expected, data points all fall well on a straight line (except the point at $T = 0.7$ where the critical region presumably ends). A systematic deviation from $\nu = 1/2$ would lead to a slightly curved line instead of a straight line. In addition, the exponent, η at T_c, seems to be consistent with the value for the classical system.

Our simulations reveal a rich structure, as shown in the phase diagram (Figure 6.20) for these $S = 1/2$ quantum spins. The antiferromagnetic ordered region and the topological ordered region are especially relevant to the high-T_c materials.

6.4 Nuclear Matter

*** Contribution needed from G. C. Fox

Figure 6.20: Phase diagram for the spin-$\frac{1}{2}$ quantum system Equation 6.12. The solid points are from quantum Monte Carlo simulations. For large $|h|$, the system is practically an Ising system. Near $h = 0$ or $h = -2$, the logarithmic relation, Equation 6.16 holds.

6.5 Dimension of Dynamical Systems

*** Contribution needed from G. C. Fox

6.6 Viterbi Convolution

*** Contribution needed from G. C. Fox

6.7 Planetary Stereo Images

We describe an approach to estimating, on the hypercube, the three-dimensional physical properties of planetary objects from the two-dimensional images returned from NASA's planetary missions, and from the Hubble Space Telescope. The hypercube is used in a simple parallel mode with each node assigned calculations for a subset of the image pixels, with no internodal communication required. The estimation uses an iterative linear least-squares approach where the data are the pixel brightness values in the images; and partials of theoretical models of these brightness values are computed for use in a square root formulation of the normal equations. The underlying three-dimensional model of the object consists of a triaxial ellipsoid overlayed with a spherical harmonic expansion to describe

low- to mid-spatial frequency topographic or surface composition variations. Partials can be calculated for the parameters describing overall body shape, orientation, topography and reflection characteristics, as well as for camera pointing. We discuss some of the possible applications of the program, describe the essentials of the technical approach, and summarize our progress and results to date.

6.8 A Hierarchical Scheme for Surface Reconstruction and Discontinuity Detection

Vision (both biological and computer-based) is a complex process that can be characterized by multiple stages where the original iconic information is progressively distilled and refined. While the first researchers to approach the problem underestimated the difficulty of the task (after all it does not require a lot of effort for a human to open the eyes, form a model of the environment, recognize objects, move, ...), in the last years a scientific basis has been given to the first stages of the process (*low and intermediate-level vision*) and a large set of special-purpose algorithms are available for *high-level* vision.

It is already possible to execute low-level operations (like filtering, edge detection, intensity normalization) in real-time (30 frames/sec) using special purpose digital hardware (like digital signal processors). On the contrary, higher-level visual tasks tend to be specialized to the different applications, and require general purpose hardware and software facilities.

Parallelism and multi-resolution processing are two effective strategies to reduce the computational requirements of higher visual tasks (see for example [Battiti:89g], [Furmanski:88c], [Marr:76a]). We describe a general software environment for implementing medium-level computer vision on large-grain size MIMD computers. The purpose has been to implement a multi-resolution strategy based on iconic data structures (two-dimensional arrays that can be indexed with the pixels' coordinates) distributed to the computing nodes using *domain decomposition.*

In particular, the environment has been applied successfully to the *visible surface reconstruction* and *discontinuity detection* problems. Initial constraints are transformed into a robust and explicit representation of the space around the viewer. In the *shape from shading* problem, the constraints are on the orientation of surface patches, while in the *shape from motion problem* (for example), the constraints are on the depth values.

We will describe a way to compute the motion (*optical flow*) from the intensity arrays of images taken at different times in Section 6.10.

Discontinuities are necessary both to avoid mixing constraints pertaining to different physical objects during the reconstruction, and to provide a primitive perceptual organization of the visual input into different elements related to the human notion of objects.

6.8.1 Multigrid Method with Discontinuities

The purpose of early vision is to undo the image formation process, recovering the properties of visible three-dimensional surfaces from the two-dimensional array of image intensities.

Computationally, this amounts to solving a very large system of equations. In general, the solution is not unique or does not exist (and therefore, one must settle for a suitable approximation).

The class of admissible solutions can be restricted by introducing a priori knowledge: the desired "typical" properties are enforced transforming the inversion problem into the *minimization of a functional*. This is known as *regularization method* [Poggio:85a]. Applying the calculus of variations, the *stationary* points are found by solving the Euler-Lagrange equation partial differential equations.

In standard methods for solving PDE's, the problem is first discretized on a finite dimensional approximation space. The very large algebraic system obtained is then solved using, for example, "relaxation" algorithms which are local and iterative. The local structure is essential for the efficient use of parallel computation.

By the local nature of the relaxation process, solution errors on the scale of the solution grid step are corrected in a few iterations; on the contrary, larger-scale errors are corrected very slowly. Intuitively, in order to correct them, information must be spread over a large scale by the "sluggish" neighbor-neighbor influence. If we want a *larger spread of influence* per iteration, we need large-scale connections for the processing units, *i.e.*, we need to solve a simplified problem on a coarser grid.

The pyramidal structure of the multigrid solution grids is illustrated in Figure 6.21.

This simple idea and its realization in the *multigrid algorithm* not only leads to asymptotically optimal solution times (i.e. convergence in $O(n)$ operations) but also dramatically decreases solution times for a variety of practical problems, as shown in [Brandt:77a].

Figure 6.21: Pyramidal structure for multigrid algorithms and general flow of control

The multigrid "recipe" is very simple. First use *relaxation* to obtain an approximation with smooth error on a fine grid. Then, given the smoothness of the error, calculate corrections to this approximation on a coarser grid, and to do this first relax, then correct *recursively* on still coarser grids. Optionally, you can also use *nested iteration* (use coarser grids to provide a good starting point for finer grids) to speed up the initial part of the computation.

Historically, these ideas were developed starting from the sixties by Bakhvalov, Fedorenko and others (see Stüben, *et al.* [Stuben:82a]). The sequential multigrid algorithm has been used for solving PDE's associated with different early vision problems in [Terzopoulos:86a].

It is shown in [Brandt:77a] that, with a few modifications in the basic algorithms, *the actual solution* (not the error) can be stored in each layer (*full approximation storage algorithm*). This method is particularly useful for visual reconstruction where we are interested not only in the finest scale result, but also in the multiscale representation developed as a byproduct of the solution process.

6.8.2 Interacting Line Processes

Line processes [Marroquin:84a] are binary variables arranged in a two-dimensional array. An active line process (LP=1) between two neighboring pixels indicates that there is a physical discontinuity between them. Activation is, therefore, based on a measure of the difference in pixel properties but must also take into account the presence of other LPs. The idea is that

Figure 6.22: Interaction of line processes must favor consistent and clear structures across different scales.

continuous non-intersecting chains of LPs are preferred to discontinuous and intersecting ones, as it is shown in Figure 6.22.

We propose to combine the surface reconstruction and discontinuity detection phases *in time and scale-spuce*. To do this, we introduce line processes at different scales, "connect" them to neighboring *depth processes* (henceforth DPs) at the same scale and to neighboring LPs on the finer and coarser scale. The reconstruction assigns equal priority to the two process types.

This scheme not only greatly improves convergence speed (the typical multigrid effect) but also produces a more consistent reconstruction of the piecewise smooth surface at the different scales.

6.8.3 Generic Look-up Table and Specific Parametrization

Creation of discontinuities must be favored either by the presence of a "large" difference in the z values of the nearby DPs, or by the presence of a partial discontinuity structure that can be improved.

To measure the two effects in a quantitative way, it is useful to introduce two functions: *Cost* and *Benefit*. The *Benefit* function for a vertical LP is $(\partial z/\partial x)^2 \approx (z_{i+1,j} - z_{i,j})^2/h_k^2$, and analogously for a horizontal one. The idea is that the activation of one LP is beneficial when this quantity is large.

Cost is a function of *neighborhood* configuration. A given LP updates its value in a manner depending on the values of near LPs. These LPs constitute the *neighborhood*, and we will to refer to its members as the LPs *connected*

Figure 6.23: "Connections" between neighboring line processes, at the same scale and between different scales

to the original one. The neighborhood is shown in Figure 6.23.

The updating rule for the LPs derived from the above requirements is:

$$LP \leftarrow 1 \quad \text{iff} \quad Cost < Benefit$$

A LP is activated only if the benefit is greater than its cost!

Because *Cost* is a function of a limited number of binary variables, we used a *look-up table* approach to increase simulation speed and to provide a convenient way for simulating different heuristical proposals.

A specific parametrization for the values in the table is suggested in [Battiti:89d].

6.8.4 Pyramid on a Two-Dimensional Mesh of Processors

The multigrid algorithm described in the previous section can be executed in many different ways on a parallel computer. One essential distinction that has to be done is related to the number of processors available and the "size" of a single processor.

The drawback of implementations on fine grain-size SIMD computers (where we assign one processor to each grid point) is that when iteration is on a coarse scale, all the nodes in the other scales (*i.e.*, the majority of nodes) are idle, and the efficiency of computation is seriously compromised.

Furthermore, if the implementation is on a hypercube parallel computer and if the mapping is such that all the communications paths in the pyramid are mapped into communication paths in the hypercube with length bounded

Figure 6.24: Domain decomposition for multigrid computation. Processor communication is on a two-dimensional grid, each processor operates at all levels of the pyramid.

by two [Chan:86b], a fraction of the nodes is never used because the total number of grid points is not equal to a power of two. This fraction is one third for two-dimensional problems encountered in vision.

Fortunately, if we use a MIMD computer with powerful processors, sufficient distributed memory and two-dimensional internode connections (the hypercube contains a two-dimensional mesh), the above problems do not exist.

In this case, a two-dimensional *domain decomposition* can be used efficiently: a slice of the image with its associated pyramidal structure is assigned to each processor. All nodes are working all the time, switching between different levels of the pyramid as illustrated in Figure 6.24.

No modification of the sequential algorithm is needed for points in the image belonging to the interior of the assigned domain. Vice versa points *on* the boundary need to know values of points assigned to a nearby processor. With this purpose, the assigned domain is extended to contain points assigned to nearby processors, and a communication step before each iteration on a given layer is responsible for updating this strip so that it contains the correct (most recent) values. Two exchanges are sufficient.

The recursive multiscale call `mg(lay)` is based on an alternation of relaxation steps and discontinuity detection steps as follows (software is written in C language):

```
int mg(lay) int lay;
```

```
{
 int i;
 if(lay==coarsest)step(lay);
 else{
 i=na;while(i--)step(lay);
 i=nb;if(i!=0)
 {up(lay);while(i--)mg(lay-1);down(lay-1);}
 i=nc;while(i--)step(lay);
 }
}

int step(lay) int lay;
{
 exchange_border_strip(lay);
 update_line_processes(lay); relax_depth_processes(lay);
}
```

Each step is preceded by an exchange of data on the border of the assigned domains.

Because the communication overhead is proportional to the linear dimension of the assigned image portion, the efficiency is high as soon as the number of pixels in this portion is large. Detailed results are in [Battiti:90b].

6.8.5 Results for Orientation Constraints

An iterative scheme for solving the shape form shading problem has been proposed in [Horn:85a]. A preliminary phase recovers information about orientation of the planes tangent to the surface at each point by minimizing a functional containing the image irradiance equation and an *integrability constraint*, as follows:

$$E(p,q) = \int_{Image} (I(x,y) - R(p,q))^2 + \lambda(p_y - q_x)^2 \, dx \, dy$$

where $p = \partial z/\partial x$, $q = \partial z/\partial y$, I = measured intensity, and R = theoretical reflectance function.

After the tangent planes are available, the surface z is reconstructed minimizing the following functional:

$$E(z) = \int_{Image} (z_x - p)^2 + (z_y - q)^2 \, dx \, dy$$

Figure 6.25: Reconstruction of shape from shading: standard relaxation versus multigrid

Euler-Lagrange differential equations and discretization are left as an exercise to the reader.

Figure 6.25 shows the reconstruction of the shape of a hemispherical surface starting from a ray-traced image [1]. Left is the result of standard relaxation after 100 sweeps, right the "minimal multigrid" result (with computation time equivalent to 3–4 sweeps at the finest resolution).

This case is particularly hard for a standard relaxation approach. The image can be interpreted "legally" in two possible ways: either as a *concave* or as a *convex* hemisphere. Starting from random initial values, after some relaxations, some image patches typically "vote" for one or the other interpretation and try to extend the local interpretation to a global one. This is slow (given the local nature of the updating rule) and encounters an endless struggle in the regions that mark the border between different interpretations. The multigrid approach solves this "democratic impasse" on the coarsest grids (much faster because information spreads over large distances) and propagates this decision to the finer grids, that will now concentrate their efforts on refining the initial approximation.

In Figure 6.26, we show the reconstruction of the Mona Lisa face painted by Leonardo da Vinci.

[1] A simple Lambertian reflection model is used.

Figure 6.26: Mona Lisa in three dimensions

6.8.6 Results for Depth Constraints

For the surface reconstruction problem (see [Terzopoulos:86a]) the energy functional is:

$$E(z(x,y)) = \int_{Image} (z(x,y) - d(x,y))^2 + \lambda(z_x^2 + z_y^2)\, dx\, dy$$

A physical analogy is that of fitting the depth constraints $d(x,y)$ with a membrane pulled by springs connected to them. The effect of active discontinuities ($DN = 1$) is that of "cutting the membrane" in the proper places.

Figure 6.27 shows the simulation environment on the SUN workstation, and the reconstruction of a "Randomville" image (random quadrangular blocks placed in the image plane). The original surface, the surface corrupted by noise (25%) and reconstruction on different scales, are shown in this order.

For 129 x 129 "images" and 25% noise, a faithful reconstruction of the surface (within a few percent of the original one) is obtained after one single multiscale sweep (with V cycles) on four layers. The total computational time corresponds approximately to the time required by three relaxations on the finest grid. Because of the optimality of multiscale methods, time increaseslinearly with the number of image pixels.

6.8.7 Credits and C^3P References

The parallel simulation environment was written by Roberto Battiti. Geoffrey Fox, Christof Koch and Wojtek Furmanski contributed with many ideas and suggestions.

Figure 6.27: Simulation environment and reconstruction of a "Randomville" scenery

C3P References: [Battiti:89d], [Furmanski:88c]

6.9 Character Recognition by Neural Nets

Much of the current interest in neural networks can be traced to the introduction a few years ago of effective learning algorithms for these systems ([Denker:86a], [Parker:82a], [Rumelhart:86a]). In [Rumelhart:86a] Chapter 8, it was shown that for some problems using multi-layer perceptrons (MLP), back-propagation was capable of finding a solution very reliably and quickly. Back-propagation has been applied to a number of realistic and complex problems [Seinowski:87a], [Denker:87a].

Real world problems are inherently structured, so methods incorporating this structure will be more effective than techniques applicable to the general case. In practice, it is very important to use whatever knowledge one has about the form of possible solutions in order to restrict the search space. For multi-layer perceptrons, this translates into constraining the weights or modifying the learning algorithm so as to embody the topology, geometry, and symmetries of the problem.

Here, we are interested in determining how automatic learning can be improved by following the above suggestion of restricting the search space of the weights. To avoid high level cognition requirements, we consider the problem of classifying hand-printed upper-case Roman characters. This is a specific pattern recognition problem, and has been addressed by methods other than neural networks. Generally the recognition is separated into

two tasks: the first one is a pre-processing of the image using translation, dilation, rotations, etc., to bring it to a standardized form; in the second one, this pre-processed image is compared to a set of templates and a probability is assigned to each character or each category of the classification. If all but one of the probabilities are close to zero, one has a high confidence level in the identification. This second task is the more difficult one, and the performance achieved depends on the quality of the matching algorithm. Our focus is to study how well a MLP can learn a satisfactory matching to templates, a task one believes the network should be good at.

In regards to the task of pre-processing, MLPs have been shown capable [Rumelhart:86a] Chapter 8 of performing translations at least in part, but it is simpler to implement this first step using standard methods. This combination of traditional methods and neural network matching can give us the best of both worlds. In what follows, we suggest and test a learning procedure which preserves the geometry of the two-dimensional image from one length scale transformation to the next, and embodies the difference between coarse and fine scale features.

6.9.1 MLP in General

There are many architectures for neural networks; we shall work with Multi-Layer Perceptrons. These are feed-forward networks, and the network to be used in our problem is schematically shown in Figure 6.28. There are two processing layers: the output and hidden layers. Each one has a number of identical units (or "neurons"), connected in a feed-forward fashion by wires, often called weights because each one is assigned a real number w_i. The input to any given unit is $\sum w_i x_i$, where i labels incoming wires and x_i is the input (or current) to that wire. For the hidden layer, x_i is the value of a bit of the input image, and for the output layer, it is the output from a unit of the hidden layer.

Generally, the output of a unit is a non-linear, monotonic increasing function of the input. We make the usual choice and take

$$g(x) = \frac{1}{1 + e^{-x+\theta}}$$

to be our neuron input/output function. θ is the threshold and can be different for each neuron. The weights and thresholds are usually the only quantities which change during the learning period. We wish to have a network perform a mapping M from the input space to the output space.

Figure 6.28: A Multi-Layer Perceptron

Introducing the actual output $A(I)$ for an input I, one first chooses a metric for the output space, and then seeks to minimize $d(A(I), M(I))$, where d is a measure of the distance between the two points. This quantity is also called the error function, the energy, or (the negative of) the harmony function. Naturally, $A(I)$ depends on the w_i's. One can then apply standard minimization searches like simulated annealing [Kirkpatrick:83a] to attempt to change the w_i's so as to reduce the error. The most commonly used method is gradient descent, which for MLP is called back-propagation because the calculation of the gradients is performed in a feed-backwards fashion. Improved descent methods may be found in [Dahl:87a], [Parker:87a] and in Section 9.10 of this volume.

The minimization often runs into difficulties because one is searching in a very high-dimensional space, and the minima may be narrow. In addition, the straightforward implementation of back-propagation will often fail because of the many minima in the energy landscape. This process of minimization is referred to as learning or memorization as the network tries to match the mapping M. In many problems though, the input space is so huge that it is neither conceivable nor desirable to present all possible inputs to the network for it to memorize. Given part of the mapping M, the network is expected to guess the rest: this is called generalization. As shown clearly in [Denker:87a] for the case of a discrete input space, generalization is often an ill-posed problem: many generalizations of M are possible. To achieve the kind of generalization humans want, it is necessary to tell the network about the mapping one has in mind. This is most simply done by constraining the weights to have certain symmetries as in [Denker:87a]. Our

approach will be similar, except that the "outside" information will play an even more central role during the learning process.

6.9.2 Character Recognition using MLP

To do character recognition using a MLP, we take the input layer of the network to be a set of image pixels, which can take on analogue (or grey scale) values between 0 and 1. The two-dimensional set of pixels is mapped onto the set of input neurons in a fairly arbitrary way: for an $N \times N$ image, the top row of N pixels is associated with the first N neurons, the next row of N pixels is associated with the next N neurons, and so forth. At the start of the training process the network has no knowledge of the underlying two-dimensional structure of the problem (that is, if a pixel is on, nearby pixels in the two-dimensional space are also likely to be on). The network discovers the two-dimensional nature of the problem during the learning process.

We taught our networks the alphabet of 26 upper-case Roman characters. To encourage generalization, we show the net many different hand-drawn versions of each character. The 320 image training set is shown in Figure 6.29. These images were hand-drawn using a mouse attached to a SUN workstation. The output is encoded in a very sparse way. There are only 26 outputs we want the net to give, so we use 26 output neurons and map the output pattern: first neuron on, rest off, to the character "A"; the pattern: second neuron on, rest off, to "B"; etc. Such an encoding scheme works well here, but is clearly unworkable for mappings with large output sets such as chinese characters or Kanji. In such cases, one would prefer a more compact output encoding, with possibly an additional layer of hidden units to produce the more complex outputs.

As mentioned earlier, we do not feed images directly into the network. Instead, simple, automatic pre-processing is done which dilates the image to a standard size and then translates it to the center of the pixel space. This greatly enhances the performance of the system—it means that one can draw a character in the upper left-hand corner of the pixel space and the system easily recognizes it. If we did not have the pre-processing, the network would be forced to solve the much larger problem of character recognition of all possible sizes and locations in the pixel space. Two other worthwhile pre-processors are rotations (rotate to a standard orientation) and intensity normalization (set linewidths to some standard value). We do not have these in our current implementation.

The MLP is used only for the part of the algorithm where one matches to

Figure 6.29: The Training Set of 320 Hand-written Characters, Digitized on a 32 x 32 Grid

templates. Given any fixed set of exemplars, a neural network will usually learn this set perfectly, but the performance under generalization can be very poor. In fact, the more weights there are, the faster the learning (in the sense of number of iterations, not in the sense of CPU time), and the worse the ability to generalize. This was in part realized in [Gullichsen:87a], where the input grid was 16 x 16. If one has a very fine mesh at the input level, so that a great amount of detail can be seen in the image, one runs the risk of having terrible generalization properties because the network will tend to focus upon tiny features of the image, ones which humans would consider irrelevant.

We will show one approach to overcoming this problem. We desire the potential power of the large, high resolution net, but with the stable generalization properties of small, coarse nets. Though not so important for upper-case Roman characters, where a rather coarse grid does well enough (as we will see), a fine mesh is necessary for other problems such as recognition of Kanji characters or hand-writing. A possible "fix", similar to what was done for dealing with the problem of clump counting [Denker:87a] is to hard-wire the first layer of weights to be local in space, with a neighborhood growing with the mesh fineness. This reduces the number of weights, thus postponing the deterioration of the generalization. However, for a MLP with a single hidden layer, this approach will prevent the detection of many non-local correlations in the images, and in effect this fix is like removing the first layer of weights.

6.9.3 The Multi-Scale Technique

We would like to train large, high resolution nets. If one tries to do this directly, by simply starting with a very large network and training by the usual back-propagation methods, not only is the training slow (because of the large size of the network) but the generalization properties of such nets are poor. As described above, a large net with many weights from the input layer to the hidden layer tends to "grandmother" the problem, leading to poor generalization.

The hidden units of a MLP form a set of feature extractors. Considering a complex pattern such as a chinese character, it seems clear that some of the relevant features which distinguish it are large, long range objects requiring little detail while other features are fine scale and require high resolution. Some sort of multi-scale decomposition of the problem therefore suggests itself. The method we will present below builds in long range feature extractors by training on small networks and then uses these as an intelligent starting point on larger, higher resolution networks. The method is somewhat analogous to the multi-grid technique for solving partial differential equations.

Let us now present our multi-scale training algorithm. We begin with the training set, such as the one shown in Figure 6.29, defined at the high resolution (in this case, 32×32). Each exemplar is coarsened by a factor of two in each direction using a simple, grey scale averaging procedure. 2×2 blocks of pixels in which all four pixels were "on" map to an "on" pixel, those in which three of the four were "on" map to a "3/4 on" pixel, and so on. The result is that each 32×32 exemplar is mapped to a 16×16 exemplar in such a way as to preserve the large scale features of the pattern. The procedure is then repeated until a suitably coarse representation of the exemplars is reached. In our case, we stopped after coarsening to 8×8.

At this point, a MLP is trained to solve the coarse mapping problem by one's favorite method (back-propagation, simulated annealing, etc.). In our case, we set up a MLP of 64 inputs (corresponding to 8×8), 32 hidden units, and 26 output units. This was then trained on the set of 320 coarsened exemplars using the simple back propagation method with a momentum term [Rumelhart:86a], Chapter 8. Satisfactory convergence was achieved after approximately 50 cycles through the training set.

We now wish to boost back to a high resolution MLP, using the results of the coarse net. We use a simple interpolating procedure which works well. We leave the number of hidden units unchanged. Each weight from the input

Figure 6.30: An Example Flowchart for the Multi-scale Training Procedure. This was the procedure used in this text, but the averaging and boosting can be continued through an indefinite number of stages.

layer to the hidden layer is split or "un-averaged" into four weights (each now attached to their own pixel), with each 1/4 the size of the original. The thresholds are left untouched during this boosting phase. This procedure gives a higher resolution MLP with an intelligent starting point for additional training at the the finer scale. In fact, before any training at all is done with the 16×16 MLP (boosted from 8×8) it recalls the 16×16 exemplars quite well. This is a measure of how much information was lost when coarsening from 16×16 to 8×8. The boost and train process is repeated to get to the desired 32×32 MLP. The entire multi-scale training process is illustrated in Figure 6.30.

6.9.4 Results

Here we give some details of our results and compare with the standard approach. As mentioned in the previous section, a 32×32 MLP (1024 inputs, 32 hidden units, 26 output units) was trained on the set of Figure 6.29 using the multi-scale method. Outputs are never exactly 0 or 1, so we defined a "successful" recognition to occur when the output value of the desired letter was greater than 0.9, and all other outputs were less than 0.1. The training on the 8×8 grid used back-propagation with a momentum term and went through the exemplars sequentially. The weights are changed so as to reduce the error function for the current character. The result is that the system does not reach an absolute minimum. Rather, at long times the weight

Figure 6.31: The Learning Curve for our Multi-scale Training Procedure Applied to 320 Hand-written Characters. The first part of the curve is the training on the 8×8 net, the second is the training on the 16×16 net, and the last is the training on the full, 32×32 net. The curve is plotted as a function of CPU time and not sweeps through the presentation set, so as to exhibit the speed of training on the smaller networks.

values oscillate with a period equal to the time of one sweep through all the exemplars. This is not a serious problem as the oscillations are very small in practice. Figure 6.31 shows the training curve for this problem. The first part of the curve is the training of the 8×8 network; even though the grid is a bit coarse, almost all of the characters can be memorized. Proceeding to the next grid by scaling the mesh size by a factor of two and using the 16×16 exemplars, we obtained the second part of the learning curve in Figure 6.31. The 8×8 net got 315/320 correct. After 12 additional sweeps on the 16×16 net, a perfect score of 320/320 is achieved. The third part of Figure 6.31 shows the result of the final boost to 32×32. In just two cycles on the 32×32 net, a perfect score of 320/320 was achieved and the training was stopped. It is useful to compare these results with a direct use of back-propagation on the 32×32 mesh without using the multi-scale procedure. Figure 6.32 shows the corresponding learning curve, with the result from Figure 6.31 drawn in for comparison. The learning via the multi-scale method takes much less computer time. In addition, the internal structure of the resultant network is much different and we will now turn to this question.

How do these two networks compare for the real task of recognizing exemplars not belonging to the training set? We used as a generalization test set 156 more hand-written characters. Though there are no ambiguities

Figure 6.32: A Comparison of Multi-scale Training with the Usual, Direct Back-propagation Procedure. The curve labeled "Multiscale" is the same as Figure 6.31, only re-scaled by a factor of two. The curve labeled "Brute Force" is from directly training a 32 x 32 network, from a random start, on the learning set. The direct approach does not quite learn all of the exemplars, and takes much more CPU time.

for humans in this test set, the networks did make mistakes. The network from the direct method made errors 14% of the time, and the multi-scale network made errors 9% of the time. We feel the improved performance of the multi-scale net is due to the difference in quality of the feature extractors in the two cases. In a two-layer MLP, we can think of each hidden-layer neuron as a feature extractor which looks for a certain characteristic shape in the input; the function of the output layer is then to perform the higher-level operation of classifying the input based on which features it contains. By looking at the weights connecting a hidden-layer neuron to the inputs, we can determine what feature that neuron is looking for.

For example, Figure 6.33 shows the input weights of two neurons in the 8 x 8 net. The neuron of (a) seems to be looking for a stroke extending downward and to the right from the center of the input field. This is a feature common to letters like A, K, R, and X. The feature extractor of (b) seems to be a "NOT S" recognizer and, among other things, discriminates between "S" and "Z".

Even at the coarsest scale, the feature extractors usually look for blobs rather than correlating a scattered pattern of pixels. This is encouraging since it matches the behavior we would expect from a "good" character recognizer. The multiscale process accentuates this locality, since a single

Figure 6.33: Two Feature Extractors for the Trained 8×8 net. This figure shows the connection weights between one hidden-layer neuron and all the input-layer neurons. Black boxes depict positive weights, while white depicts negative weights; the size of the box shows the magnitude. The position of each weight in the 8×8 grid corresponds to the position of the input pixel. We can view these pictures as maps of the features which each hidden-layer neuron is looking for. In (a), the neuron is looking for a stroke extending down and to the right from the center of the input field; this neuron fires upon input of the letter "A" for example. In (b), the neuron is looking for something in the lower center of the picture, but it also has a strong "NOT S" component. Among other things, this neuron discriminates between an "S" and a "Z". The outputs of several such feature extractors are combined by the output layer to classify the original input.

Figure 6.34: The Same Feature Extractor as in Figure 6.33(b), after the Boost to 16 × 16. There is an obvious correspondence between each connection in Figure 6.33(b) and 2 × 2 clumps of connections here. This is due to the multi-scale procedure, and leads to spatially smooth feature extractors.

pixel grows to a local clump of four pixels at each rescaling. This effect can be seen in Figure 6.34, which shows the feature extractor of Figure 6.33(b) after scaling to 16 × 16 and further training. Four-pixel clumps are quite obvious in the 16 × 16 network. The feature extractors obtained by direct training on large nets are much more scattered (less smooth) in nature.

6.9.5 Comments and Variants on the Method

Before closing, we would like to make some additional comments on the multi-scale method and suggest some possible extensions.

In a pattern recognition problem such as character recognition, the two-dimensional spatial structure of the problem is important. The multi-scale method preserves this structure so that "reasonable" feature extractors are produced. An obvious extension to the present work is to increase the number of hidden units as one boosts the MLP to higher resolution. This corresponds to adding completely new feature extractors. We did not do this in the present case since 32 hidden units were sufficient—the problem of recognizing upper-case Roman characters is too easy. For a more challenging problem such as chinese characters, adding hidden units will probably be necessary. We should mention that incrementally adding hidden units is easy to do and works well—we have used it to achieve perfect convergence of a back-propagation network for the problem of tic-tac-toe.

When boosting, the weights are scaled down by a factor of four and so it is important to also scale down the learning rate (in the back-propagation algorithm) by a factor of four.

We defined our "blocking" or coarsening procedure to be a simple, grey scale averaging of 2×2 blocks. There are many other possibilities, well known in the field of real-space renormalization in physics. Other interesting blocking procedures include: using a scale factor, λ, different from two; using majority rule averaging; simple decimation; etc.

Multi-scale methods work well in cases where spatial locality or smoothness is relevant (otherwise, the interpolation approximation is bad). Another way of thinking about this is that we are decomposing the problem onto a set of spatially local basis functions such as gaussians. In other problems, a different set of basis functions may be more appropriate and hence give better performance.

The multi-scale method uses results from a small net to help in the training of a large network. The different sized networks are related by the re-scaling or dilation operator. A variant of this general approach would be to use the translation operator to produce a pattern matcher for the game of Go. The idea is that at least *some* of the complexity of Go is concerned with local strategies. Instead of training a MLP to learn this on the full 19×19 board of Go, do the training on a "mini-Go" board of 5×5 or 7×7. The appropriate way to relate these networks to the full sized one is not through dilations, but via the translations: the same local strategies are valid everywhere on the board.

6.9.6 Credits

Steve Otto had the original idea for the MultiScale training technique. Otto and Ed Felten and Olivier Martin developed the method. Jim Hutchinson contributed by supplying the original back-propagation program.

6.10 An Adaptive Multiscale Scheme for Real-Time Motion Field Estimation

When moving objects in a scene are projected onto an image plane (for example onto our retina), the real velocity field is transformed into a two-dimensional field, known as the motion field.

By taking more images at different times and calculating the motion

field, we can extract useful parameters like the time to collision, useful for obstacle avoidance. If we know the motion of a camera (or our ego-motion), we can reconstruct the entire three-dimensional structure of the environment (if the camera translates, near objects will have a larger motion field with respect to distant ones). The depth measurements can be used as starting constraints for a surface reconstruction algorithm like the one described in Section 9.10.

In particular situations, the apparent motion of the brightness pattern, known as the optical flow, provides a sufficiently accurate estimate of the motion field. Although the adaptive scheme that we propose is applicable to different methods, the discussion will be based on the scheme proposed by Horn and Schunck [Horn:81a]. They use the assumptions that the image brightness of a given point remains constant over time, and that the optical flow varies smoothly almost everywhere. Satisfaction of these two constraints is formulated as the problem of minimizing a quadratic energy functional (see also [Poggio:85a]). The appropriate Euler-Lagrange equations are then discretized on a single or multiple grids and solved using, for example, the Gauss-Seidel "relaxation" method [Horn:81a], [Terzopoulos:86a]). The resulting system of equations (two for every pixel in the image) is:

$$(I_x\, u_{i,j} + I_y\, v_{i,j} + I_t)\, I_x = \frac{\alpha^2}{\Delta x^2}\,(\bar{u}_{i,j} - u_{i,j})$$

$$(I_x\, u_{i,j} + I_y\, v_{i,j} + I_t)\, I_y = \frac{\alpha^2}{\Delta x^2}\,(\bar{v}_{i,j} - v_{i,j})$$

where $u_{i,j} = \frac{dx}{dt}$ and $v_{i,j} = \frac{dy}{dt}$ are the optical flow variables to be determined, I_x, I_y, I_t are the partial derivatives of the image brightness with respect to space and time, $\bar{u}$ and $\bar{v}$ are local averages, Δx is the spatial discretization step, and α controls the smoothness of the estimated optical flow.

6.10.1 Errors in Computing the Motion Field

Now, we need to estimate the partial derivatives in the above equations with *discretized formulas* starting from brightness values that are *quantized* (say integers from 0 to n) and noisy. Given these derivative estimation problems, the optimal step for the discretization grid depends on local properties of the image. Use of a single discretization step, produces large errors on some images. Use of a *homogeneous* multiscale approach, where a set of grids at

Figure 6.35: Measured velocity for superposition of sinusoidal patterns as a function of the ratio of short to long wavelength components. Dashed line: $\Delta x = 2$, continuous line: $\Delta x = 1$.

different resolutions is used, may in some cases produce a good estimation on an intermediate grid and a bad one on the final and finest grid. Enkelmann and Glazer [Enkelmann:88a], [Glazer:84a] encountered similar problems.

These difficulties can be illustrated with the following one-dimensional example. Let's suppose that the intensity pattern observed is a superposition of two sinusoids of different wavelengths:

$$I(x,t) \propto (1 + R + \sin\left(\frac{2\pi}{6}(x - 2t)\right) + R\sin\left(\frac{2\pi}{3}(x - 2t)\right)$$

where R is the ratio of short to long wavelength components. Using the *brightness constancy* assumption ($dI/dt = 0$ or $v\,I_x + I_t = 0$, see [Horn:81a]) the measured velocity $\tilde{v}$ is given by:

$$\tilde{v} = -\tilde{I}_t/\tilde{I}_x$$

where $\tilde{I}_x$ and $\tilde{I}_t$ are the three-point approximations of the spatial and temporal brightness derivatives[2].

Now, if we calculate the estimated velocity on two different grids, with spatial step Δx equal to one and two, as a function of the parameter, R, we obtain the result illustrated in Figure 6.35.

While on the coarser grid, the correct velocity is obtained (in this case), on the finer one, the measured velocity depends on the value of R. In

[2]That is $\tilde{I}_x = \frac{1}{2\Delta x}(I(x + \Delta x) - I(x - \Delta x))$ and analogously for $\tilde{I}_t$.

particular, if R is greater than 0.5, we obtain a velocity in the opposite direction!

We propose a method for "tuning" the discretization grid to a measure of the reliability of the optical flow derived at a given scale. This measure is based on a *local estimate of the errors* due to noise and discretization, and is described in [Battiti:89g].

6.10.2 Adaptive Multiscale Scheme on a Multicomputer

First, a Gaussian pyramid [Burt:84a] is computed from the given images. This consists of a hierarchy of images obtained filtering the original ones with Gaussian filters of progressively larger size.

Then, the optical flow field is computed at the coarsest scale using relaxation, and the estimated error is calculated for every pixel. If this quantity is less than a given threshold T_{err}, the current value of the flow is interpolated to the finer resolutions without further processing. This is done by setting an *inhibition flag* contained in the grid points of the pyramidal structure, so that these points do not participate in the relaxation process. On the contrary, if the error is larger than T_{err}, the approximation is relaxed on a finer scale and the entire process is repeated until the finest scale is reached.

In this way, we obtain a *local inhomogeneous* approach where areas of the images, characterized by different spatial frequencies or by different motion amplitudes, are processed at the appropriate resolutions, avoiding corruption of good estimates by inconsistent information from a different scale (the effect shown in the previous example). The optimal grid structure for a given image, is translated into a pattern of active and *inhibited* grid points in the pyramid, as illustrated in Figure 6.36.

The motivation for *freezing* the motion field as soon as the error is below threshold, is that the estimation of the error may *itself* become incorrect at finer scales and, therefore, useless in the decision process. It is important to point out that single scale or homogeneous approaches cannot adequately solve the above problem. Intuitively, what happens in the adaptive multiscale approach is that the velocity is *frozen* as soon as the spatial and temporal differences at a given scale are big enough to avoid quantization errors, but small enough to avoid errors in the use of discretized formulas. The only assumption made in this scheme is that the largest motion in the scene can be reliably computed at one of the used resolutions. If the images contain motion discontinuities, *line processes* (indicating the presence of these discontinuities) are necessary to prevent smoothing where it is not

Figure 6.36: (left) adaptive grid and activity pattern in the multi-resolution pyramid, (right) two mapping strategies

Figure 6.37: Efficiency and solution times

desired (see [Battiti:89d] and the contained references).

Large grain size multicomputers, with a mapping based on *domain decomposition* and limited coarsening, have been used to implement the adaptive algorithm, as described in Section 6.8. The efficiency and solution times for an implementation with Transputers (details in [Battiti:90b]) are show in Figure 6.37.

Real-time computation with high efficiency is within the reach of available digital technology!

On a board with four Transputers, and using the *Express* communication routines from Parasoft, the solution time for 129×129 images is of the order of one second.

Figure 6.38: Results for ray-traced "plaid" image and for natural scene

6.10.3 Results

Results of the algorithm show that the adaptive method is capable of effectively reducing the solution error. In the last figures, we show two test images (showing a "plaid" pattern and a natural scene), with the obtained optical flow.

For the "plaid" image, we show the r.m.s. error obtained with the adaptive (upper line) and homogeneous (lower line) scheme and the resulting fields.

For the natural image, we show the average computed velocity (in a region centered on the pine cone) as a function of the correct velocity for different number of layers. Increasing the number of resolution grids increases the range of velocities that are detected correctly by the algorithm.

6.10.4 Credits and C^3P References

The software implementation is based on the multiscale vison environment developed by Roberto Battiti and described in Section 9.10. Christof Koch and Edoardo Amaldi collaborated in the project.

C3P References: [Battiti:89g]

6.11 Collective Stereopsis

The collective stereopsis algorithm described in [Marr:76a] was historically one of the first "cooperative" algorithms based on relaxation proposed for

Figure 6.39: Results of Collective Stereopsis

early vision.

The purpose is that of measuring the difference in retinal position (*disparity*) of features of a scene observed with two eyes (or video-cameras). A fiber of "neurons" (one for each disparity value) is placed at each pixel position. Each neuron inhibits neurons of different disparities at the same location (because the disparity is unique) and excites neurons of the same disparity at near location (because disparity tends to vary smoothly). After convergence the activation pattern corresponds to the disparity field.

The parallel implementation is based on domain decomposition and the results are illustrated if Figure 6.39.

They show the initial state of disparity computation and the evolution in time of the different layers of disparity neurons. Detail are described in [Battiti:88a].

6.12 Navigation

*** Contribution needed by G. C. Fox

Chapter 7

Independent Parallelism

7.1 Problem Structure and Mappings

*** Contribution needed from G. C. Fox

7.2 Dynamically Triangulated Random Surfaces

7.2.1 Introduction

In this section, we describe some large-scale parallel simulations of dynamically triangulated random surfaces [Baillie:89h], [Baillie:90c], [Baillie:90d], [Baillie:90e]. Dynamically triangulated random surfaces have been suggested as a possible discretization for string theory in high energy physics and fluid surfaces or membranes in biology. As physicists, we shall focus on the former.

String theories describe the interaction of one-dimensional string-like objects in an analogous fashion to the way particle theories describe the interaction of zero-dimensional point-like particles. String theory has its genesis in the dual models that were put forward in the 1960's to describe the behavior of the hadronic spectrum then being observed. The dual model amplitudes could be derived from the quantum theory of a string-like object [Nambu:70a], [Nielsen:70a], [Susskind:70a]. It was later discovered that these so-called bosonic strings could apparently only live in 26 dimensions [Lovelace:68a] if they were to be consistent quantum-mechanically. They also had tachyonic (negative mass-squared) ground states, which is normally the sign of an instability. Later, fermionic degrees of freedom were added to the theory yielding the supersymmetric Neveu-Schwarz-Ramond

[Neveu:71a] (NSR) string. This has a critical dimension of 10, rather than 26, but still suffers from the tachyonic ground state. Around 1973, it became clear that QCD provided a plausible candidate for a model of the hadronic spectrum, and the interest in string models of hadronic interactions waned. However, about this time it was also postulated by numerous groups that strings [Scherk:74a] might provide a model for gravity because of the prescence of higher spin excitations in a natural manner. A further piece of the puzzle fell into place in 1977 when [Gliozzi:77a] found a way to remove the tachyon from the NSR string. The present explosion of work on string theory began with the work of Green and Schwarz [Green:84a], who found that only a small number of string theories could be made tachyon free in 10 dimensions, and predicted the occurrence of one such that had not yet been constructed. This appeared soon after in the form of the heterotic string [Gross:85a], which is a sort of composite of the bosonic and supersymmetric models.

After these discoveries, the physics community leaped on string models as a way of constructing a unified theory of gravity [Schwarz:85a]. Means were found to compactify the unwanted extra dimensions and produce four-dimensional theories that were plausible grand unified models, that is, models which include both the Standard Model and gravity. Unfortunately, it now seems that much of the predictive power that came from the constraints on the 10-dimensional theories is lost in the compactification, so interest in string models for constructing grand unified theories has begun to fade. However, considered as purely mathematical entities, they have led and are leading, to great advances in complex geometry and conformal field theory. Many of the techniques that have been used in string theory can also be directly translated to the field of real surfaces and membranes, and it is from this viewpoint that we want to discuss the subject.

7.2.2 Discretized Strings

As a point particle in space moves through time, it traces out a line; similarly, as the string which looks like a line in space moves through time, it sweeps out a two-dimensional surface called the worldsheet. Thus, there are two ways in which to discretize the string: either the worldsheet is discretized or the (d-dimensional) space-time in which the string is embedded is discretized. We shall consider the former, which is referred to as the *intrinsic* approach; the latter is reviewed in [Ambjorn:89a]. Such discretized surface models fall into three categories: regular surfaces, fixed random sur-

faces and dynamical random surfaces. In the first, the surface is composed of plaquettes in a d-dimensional regular hypercubic lattice; in the second, the surface is randomly triangulated once at the beginning of the simulation; and in the third the random triangulation becomes dynamical (*i.e.*, is changed during the simulation). It is these dynamically triangulated random surfaces we wish to simulate. Such a simulation is, in effect, that of a fluid surface. This is because the varying triangulation means that there is no fixed reference frame, which is precisely what one would expect of a fluid where the molecules at two different points could interchange and still leave the surface intact. In string theory, this is called reparametrization invariance. If, instead, one used a regular surface, one would be simulating a tethered or crystalline surface on which there is considerable literature (see [Ambjorn:89b] for a survey of the work in the field). In this case, the molecules of the surface are frozen in a fixed array. There have also been simulations of fixed random surfaces, see for example [Baig:89a]. One other reason for studying random surface models is to understand integration over geometrical objects and discover whether the nonperturbative discretization procedures, which work so well for local field theories like QCD, can be applied successfully.

The partition function describing the quantum mechanics of a surface was first formulated by Polyakov [Polyakov:81a]. For a bosonic string embedded in d-dimensions, it is written as

$$Z = \int DX\, Dg \exp\left(-T \int d^2x \sqrt{g}\, g^{ab}\, \partial_a X^\mu\, \partial_b X_\mu\right), \tag{7.1}$$

where $\mu = 1, 2, \ldots, d$ labels the dimensions of the embedding space, $a, b = 1, 2$ are the coordinates on the worldsheet, and T is the string tension. The integration is over both the fields X_μ and the metric on the worldsheet g^{ab}. X_μ gives the embedding of the two-dimensional worldsheet swept out by the string in the d-dimensional space in which it lives. If we integrate over the metric, we obtain an area action for the worldsheet,

$$Z \simeq \exp\left(-T \int d^2x \sqrt{|h|}\right), \quad h_{ab} = \partial_z X^\mu\, \partial_b X_\mu, \tag{7.2}$$

which is a direct generalization of the length action for a particle, *i.e.*,

$$Z \simeq \exp\left(-\text{length of world}\right). \tag{7.3}$$

We can, thus, see that the action in equation 7.1 is the natural area action that one might expect for a surface whose dynamics were determined by the surface tension.

The first discretized model of this partition function was suggested independently by three groups: [Ambjorn:85a], [David:85a], and [Kazakov:85a].

$$Z = \sum_{t \epsilon T} \rho(t) \int \prod_{i}^{N} DX_i \exp\left(-\frac{1}{2}\sum_{\langle ij \rangle}(X_i - X_j)^2\right), \tag{7.4}$$

where the outer sum runs over some set T of allowed triangulations of the surface, weighted by their importance factors $\rho(t)$, and is supposed to represent the effect of the metric integration in the path integral. The inner sum in the exponential is over the edges $\langle ij \rangle$ of the triangulation, or mesh, and working with a fixed number of nodes N corresponds to working in a microcanonical ensemble of fixed intrinsic area. The model is that of a dynamically triangulated surface because one is instructed to perform the sum over different triangulations, so both the fields, X_μ on the mesh and the mesh itself, are dynamical objects.

A considerable amount of effort has been devoted to simulating this pure area action, both in microcanonical form with a fixed number of nodes [Billoire:86a], [Boulatov:86a], [Jurkiewicz:86a] and in grand canonical form, where the number of nodes is allowed to change in a manner which satisfies detailed balance [Ambjorn:87b], [David:87a], [Jurkiewicz:86b] (this allows measurements to be made of how the partition function varies with the number of nodes N, which determines an exponent called the string susceptibility). The results are rather disappointing, in that the surfaces appear to be in a very crumpled state, as can be seen from measuring the gyration radius $X2$, which gives a figure for the "mean size" of the surface. Its discretized form is

$$X2 = \frac{1}{9N(N-1)}\left\langle \sum_{ij}^{N}(X_i - X_j)^2 \right\rangle, \tag{7.5}$$

where the sum now runs over all pairs of nodes ij. $X2$ is observed to grow only logarithmically with N. This means that the Hausdorff dimension, d_H, which measures how the surface grows upon the addition of intrinsic area and is defined by

$$X2 \simeq N^{\frac{2}{d_H}}, \tag{7.6}$$

is infinite. Analytical work [Durhuus:84a] shows that the string tension fails to scale in the continuum limit so that, heuristically speaking, it becomes so strong that it collapses the surface into something like a branched polymer.

Figure 7.1: A figure caption is needed here

Thus, the pure area action does not provide a good continuum limit. It was observed in [Ambjorn:85a] and [Espriu:87a] that another way to understand the pathological behavior of the simulations, was to note that spike-like configurations in the surface were not suppressed by the area action, allowing it to degenerate into a spiky, crumpled object. An example of such a configuration is shown in Figure 7.1.

To overcome this difficulty, one uses the fact that adding to the pure area action a term in the extrinsic curvature squared (as originally suggested by Polyakov [Polyakov:86a] and Kleinert [Kleinert:86a] for string models of hadron interactions) smooths out the surface. In three dimensions, the extrinsic curvature of a two-dimensional surface is given by

$$K(x) = \frac{1}{r_1(x)} + \frac{1}{r_2(x)}, \tag{7.7}$$

where the r's are the principle radii of curvature at a point x on the surface. Two discretized forms of this extrinsic curvature term are possible, namely

$$K^2 = 3 \sum_i \frac{1}{\Omega_i} \left(\sum_{j(i)} (X_i - X_j) \right)^2, \tag{7.8}$$

where the inner sum is over the neighbors j of a node i and Ω_i is the sum of the areas of the surrounding triangles as shown in Figure 7.2, and

$$K^2 = \sum_{\langle ij \rangle} (1 - \hat{n}_i^\mu \, \hat{n}_{j\mu}), \tag{7.9}$$

Figure 7.2: A figure caption is needed for this figure

where one takes the dot product of the unit normals $\hat{n}_i^\mu$ of triangles which share a common edge $\langle ij \rangle$. For sufficiently large values of the K^2 coupling, the worldsheet is smooth, as shown in Figure 7.1. Analytical work [Ambjorn:87a], [Ambjorn:89b] strongly suggests, however, that a continuum limit will only be found in the limit of infinite extrinsic curvature coupling.

It came as something of a surprise, therefore, when a simulation by Catterall [Catterall:89a] revealed that the discretization (Equation 7.8) gave a third order phase transition to the smooth phase and the discretization (Equation 7.9) a *second* order phase transition, the latter of which offering the possibility of defining a continuum limit at a *finite* value of the extrinsic curvature coupling (because of the divergence of correlation length). Further work by Baillie, Johnston and Williams [Baillie:89h] confirmed the existence of this "crumpling transition". Typical results, for a surface consisting of 288 nodes, are shown in the series of Figure 7.1. The extrinsic curvature coupling, λ, is increased from 0 (the crumpled phase) up to 1.5 (the smooth phase). We estimate that the crumpling transition is around $\lambda = 0.75$.

To summarize, a dynamically triangulated random surface with a pure area action does not offer a good discretization of a bosonic string or of a fluid surface. The addition of a term in the extrinsic curvature may make this possible if the continuum limit at the second order phase transition is a string theory.

7.2.3 Computational Aspects

In order to give the reader a feel for how one actually simulates a dynamically triangulated random surface, we briefly explain our computer program, "string", which does this—more details can be found in [Baillie:90e]. As we explained previously, in order to incorporate the metric fluctuations, we randomly triangulate the worldsheet of the string or random surface to obtain a mesh and make it dynamical by allowing flips in the mesh that do not change the topology. The incorporation of the flips into the simulation makes vectorization difficult, so running on traditional supercomputers like the CRAY is not worthwhile. Similarly, the irregular nature of the dynamically triangulated random surface inhibits efficient implementation on SIMD computers like the Distributed Array Processor and the Connection Machine. Thus, in order to get a large amount of CPU power behind our random surface simulations, we are forced to run on *MIMD parallel* computers. Here, we have a choice of two main architectures: distributed memory hypercubes, or shared memory computers. We initially made use of the former, as several machines of this type were available to us—all running the *same* software environment, namely, ParaSoft's Express System [ParaSoft:88a]. Having the same software on different parallel computers makes porting the code from one to another very easy. In fact, we ran our strings simulation program on the NCUBE hypercube (for a total of 1800 hours on 512 processors), the Symult Series 2010 (900 hours on 64 processors), and the Meiko Computing Surface (200 hours on 32 procesors). Since this simulation fits easily into the memory of a single node of any of these hypercubes, we ran multiple simulations in parallel—giving, of course, linear speedup. Each node was loaded with a separate simulation (using a different random number generator seed), starting from a single mesh that has been equilibrated elsewhere, say on a Sun workstation. After allowing a suitable length of time for the meshes to decorrelate, data can be collected from each node, treating them as separate experiments. More recently, we have also run "string" on the GP1000 Butterfly (1000 hours on 14 processors) and TC2000 Butterfly II (500 hours on 14 processors) shared memory computers—again with each processor performing a unique simulation. Parallelism is thereby obtained by "decomposing" the space of Monte Carlo configurations.

The reason that we run "multiple independent Monte Carlo" simulations, rather than distributing the mesh over the processors of the parallel computer, is that this domain decomposition would be difficult for such an irregular problem. This is because, with a distributed mesh, each processor

wanting to change its part of the mesh would have to first check that the affected pieces were not simultaneously being changed by another processor. If they were, detailed balance would be violated and the Metropolis algorithm could no longer be used. Similar parallelization difficulties arise in other irregular Monte Carlo problems, such as gas and liquid systems.

The mesh is set up as a linked list in the programming language C, using software developed at Caltech for doing computational fluid dynamics on unstructured triangular meshes called DIME (Distributed Irregular Mesh Environment) [Williams:88a], [Williams:90b]. The logical structure is that of a set of triangular elements corresponding to the faces of the triangulation of the worldsheet, connected via nodes corresponding to the nodes of the triangulation of the worldsheet. The data required in the simulation (of random surfaces or computational fluid dynamics) is then stored in either the nodes or the elements. We simulate a worldsheet with a fixed number of nodes, N, which corresponds to the partition function of Equation 7.1 evaluated at a fixed area. We also fix the topology of the mesh to be spherical. (The results for other topologies, such as a torus, are identical). The Monte Carlo procedure sweeps through the mesh moving the X's which live at the nodes, and doing a Metropolis accept/reject. It then sweeps through the mesh a second time doing the flips, and again performs a Metropolis accept/reject at each attempt. In Figure 7.3 is illustrated the local change to the mesh called a flip. For any edge, there is a triangle on each side, forming the quadrilateral ABCD, and the flip consists of changing the diagonal AC to BD. Both types of Monte Carlo updates to the mesh can be implemented by identifying which elements and nodes would be affected by the change, then

1. saving the data from these affected elements and nodes;
2. making the proposed change;
3. recalculating the element data;
4. recalculating the node data;
5. calculating the change in the action δS as the difference of the sum of the new action contributions from the affected nodes, and the sum of the old action contributions;
6. given δS deciding whether to accept or reject the change using the standard Metropolis algorithm—if the change is rejected, we replace

Figure 7.3: A figure caption is needed here

Figure 7.4: A figure caption is needed here

the element and node data with the saved values; if accepted, we need do nothing since the change has already been made.

For the move, the affected elements are the neighbors of node i, and the affected nodes are the node i itself and its node neighbors, as shown in Figure 7.4. For the flip, the affected elements are the two sharing the edge, and the affected nodes are the four neighbor nodes of these elements, as shown in Figure 7.5.

7.2.4 Performance of String Program

Due to its irregular nature, "string" is an extremely good benchmark of the *scalar* performance of a computer. Hence, we timed it on several machines

Figure 7.5: A figure caption is needed here

we had access to, yielding the numbers in Table 7.1. Note that we timed *one* processor of the parallel machines. We see immediately that the Sun 4/60, known as the SPARC station 1, is the highest performance desktop Sun currently available. Moreover, this machine (running with TI 8847 floating point processor at clock rate of 20 MHz) is as fast as the new Motorola 88000 processor (at 16 MHz) which is used in the TC2000 Butterfly. Turning to the hypercubes, we see that the Meiko is twice as fast as the (scalar) Symult, which in turn is twice as fast as the NCUBE, per processor, for the "string" program. We have also run on the Weitek vector processors of the Mark III and Symult. The vector processors are faster than the scalar processors, but since "string" is entirely scalar, it does not run very efficiently on the vector processors and, hence, is still slower than on the Sun 4/60. The Mark III is as fast as the Symult, despite having one-third the clock rate, because is has a high-performance cache between its vector processor and memory. We have also timed the new IBM RISC 6000 computer, and Cimarron Boozer of SKY Computers has timed the Intel i860. As a final comparison, the CRAY X-MP runs "string" about five times faster than the Sun 4/60.

We should emphasize that these performances are for *scalar* codes. A completely different picture emerges for codes which vectorize well, like QCD. QCD, with dynamical fermions, runs on the CRAY X-MP at around 100 Mflops and pure gauge QCD runs on one processor of the Mark III at 6 Mflops. In contrast, the Sun 4/60 only achieves about 1 Mflops for pure gauge QCD. This ratio of QCD performance (which we may claim as the "realistic peak" performance of the machines) 100:6:1 compares with 5:0.7:1 for Strings. Thus, these two calculations from one area of physics illustrate

Computer	Floating Point Processor	Clock Rate (M Hz)	"string" (seconds)
Sun 3/160	FPA	16.7	73
Sun 3/50	68881	15	148
Sun 3/60	68881	20	94
Sun 386i	80387	20	23
Sun 4/280	W 1164/65	16	15
Sun 4/260	W 1164/65	16	14
Sun 4/280	TI 8847	16	12
Sun 4/60	SPARC FP	20	12
Sun 4/60	TI 8847	20	11
GP 1000	68882	16	40
TC 2000	88000	16	11
NCUBE	Custom	6.7	90
Symult	68882	25	47
Symult	Weitek 13364	25	15
Meiko	T800	20	23
Mark III	68882	16	63
Mark III	Weitek 3164	8	15
IBM RISC 6000	320	20	2.0
Intel	i860	40	2.1
CRAY	X-MP	105	1.8

Table 7.1: Time taken to execute the "string" program

clearly that the preferred computer architecture depends on the problem.

7.2.5 Conclusion

Large-scale numerical simulations are becoming increasingly important in many areas of science. Lattice QCD calculations have been commonplace in high energy physics for many years. More recently, this technique has been applied to string theories formulated as dynamically triangulated random surfaces. As we have pointed out, such computer simulations of strings are difficult to implement efficiently on all but MIMD computers, due to the inherent irregular nature of random surfaces. Moreover, on most MIMD machines, it is possible to get 100% speedup by doing multiple independent Monte Carlo's, since an entire simulation easily fits within each processor.

7.3 Numerical Study of High-T_c Spin Systems

Although the mechanism of high temperature superconductivity is not yet established, an enormous amount of experimental work has been completed on these materials and as a result, a "magnetic explanation has probably gained the largest number of adherents. In this picture, high temperature superconductivity normally results from the effects of dynamical-hole doping on the magnetic properties of CuO_2 planes, perhaps through the formation of bound hole pairs. In the undoped materials ("precursor insulators"), these CuO_2 planes are magnetic insulators and appear to be well described by the two dimensional spin-1/2 Heisenberg antiferromagnet,

$$H = \sum_{\langle ij \rangle} \vec{S}_i \cdot \vec{S}_j \ , \tag{7.10}$$

where each spin represents a d-electron on a CuO_2 site. Since many aspects of the two-dimensional Heisenberg antiferromagnet were obscure before the discovery of high-T_c, this model has been the subject of intense numerical study, and comparisons with experiments on the precursor insulators have generally been successful. (A review of this subject for the C^3P group is currently in preparation [Barnes:90a].) If the proposed "magnetic" origin of high temperature superconductivity is correct, one may only need to incorporate dynamical holes in the Heisenberg antiferromagnet to construct a model that exhibits high temperature superconductivity. Unfortunately, such models (for example the "t-J" model) are dynamical many-fermion systems and exhibit the "minus sign problem" which makes them very difficult

to simulate on large lattices. The lack of appropriate algorithms for many-fermion systems accounts in part for the relatively slow pace of theoretical work on these models.

In our work, we carried out numerical simulations of the low lying states of one- and two-dimensional Heisenberg antiferromagnets; the problems we studied on the hypercube which relate to high-T_c systems were the determination of low lying energies and ground state matrix elements of the two-dimensional spin-1/2 Heisenberg antiferromagnet and, in particular, the response of the ground state to anisotropic couplings in the generalized model

$$H = \sum_{\langle ij \rangle} S_{zi} S_{zj} + g(S_{xi} S_{xj} + S_{yi} S_{yj}) \ . \tag{7.11}$$

Until recently, the possible existence of infinite range spin antialignment "staggered magnetization" in the ground state of the two-dimensional Heisenberg antiferromagnet, which would imply spontaneous breaking of rotational symmetry, was considered an open question. Since the precursor insulators, such as $La_2 Cu O_4$, are observed to have a nonzero staggered magnetization, one might hope to observe it in the Heisenberg model as well. (It has actually been proven to be zero in the isotropic model above zero temperature, so this is a very delicate kind of long range order.) Assuming that such long range order exists, one might expect to see various kinds of singular behavior in response to anisotropies, which would choose a preferred direction for symmetry breaking in the ground state. In our numerical simulations, we measured the ground state energy per spin, e_0, the energy gap to the first spin excitation, E_{gap}, and the $\hat{z}$ component of the staggered magnetization, N_z, as a function of the anisotropy parameter, g, on L×L square lattices and attempted to extrapolate to the bulk limit. We did indeed find evidence of singular behavior at the isotropic point $g = 1$, specifically that $de_0(g)/dg$ is probably discontinuous there (Figure 7.6), $E_{gap}(g)$ decreases to zero at $g = 1$ (Figure 7.7) and remains zero for $g > 1$, and $N_z(g)$ decreases to a nonzero limit as g approaches one, is zero for $g > 1$, and is undefined at $g = 1$ ([Barnes:89c], [Barnes:89a]). Finite lattice results which led to this conclusion are shown in Figure 7.8. (Perturbative and spin-wave predictions also appear in these figures; details are discussed in the publications we have cited.) These results are consistent with a "spin flop" transition, in which the long range spin order is oriented along the energetically most favorable direction, which changes discontinuously from $\hat{z}$ to planar as g passes through the isotropic point. The qualitative behavor of the energy gap can be understood as a consequence of Goldstone's theorem, given these types of

Figure 7.6: Ground state energy per spin.

Figure 7.7: Spin excitation energy gap.

spontaneous symmetry breaking. Our results also provided interesting tests of spin wave theory, which has been applied to the study of many antiferromagnetic systems, including the two-dimensional Heisenberg model, but is of questionable accuracy, especially for small spin. In this spin-1/2 case, we found that finite-size and anisotropic effects were qualitatively described surprisingly well by spin wave theory, but that numerical results were sometimes rather inaccurate; for example, the energy gap due to a small easy-axis anisotropy was in error by about a factor of two.

In additional work, we developed hypercube programs to study static holes in the Heisenberg model, as a first step towards more general Monte Carlo investigations of the behavior of holes in antiferromagnets. We have not yet completed this work, however, due to a problem which eventually

Figure 7.8: Ground state staggered magnetization N_z.

proved to be an operating system bug. Our collaboration is now continuing this study on the Intel RX hypercube at Oak Ridge National Laboratory.

For these studies, we used the "DRGW" discrete guided random walk Monte Carlo algorithm of Barnes and Daniell [Barnes:88c], and incorporated algorithm improvements which lowered the statistical errors [Barnes:89b]. This algorithm solves the Euclidean time Schrödinger equation stochastically by running random walks in the configuration space of the system and accumulating a weight factor, which implicitly contains energies and matrix elements. Since the algorithm only requires a single configuration, our memory requirements were very small so we simply placed a copy of the program on each node; no internode communication was necessary. A previously developed DGRW spin system Fortran program written by T. Barnes was rewritten in C and adapted to the hypercube by D. Kotchan, and an independent DGRW code was written by E. S. Swanson for debugging purposes. Our collaboration for this work eventually grew to include K. J. Cappon (who also wrote a DGRW Monte Carlo code) and E. Dagotto (UCSB/ITP) and A. Moreo (UCSB), who wrote Lanczos programs to give essentially exact results on the 4×4 lattice. This provided an independent check of the accuracy of our Monte Carlo results.

As a "note added in proof", in addition to providing resources that led to these physics results, access to the hypercube and the support of the C^3P group were very helpful in the PhD programs of D. Kotchan and E. S. Swanson, and their experience has encouraged several other graduate-level theorists at the University of Toronto to pursue studies in computational physics, both in high temperature superconductivity (K. J. Cap-

pon and W. MacReady) and Monte Carlo studies of quark model physics (G. Grondin).

7.4 Statistical Gravitational Lensing on the Mark III Hypercube

We used the Caltech/JPL Mark III to simulate gravitational lenses. These are galaxies which bend the light of a background quasar to produce multiple images of it. Astronomers are very interested in these objects, and have discovered more than 10 of them to date. Several exhibit symptoms of lensing by more than one galaxy. This spurred us to simulate models of this class of lens. Our model systems were composed of two galaxy-like lensing potentials in different positions and redshifts. We studied about 100 cases at a resolution of 1536^2, taking about three weeks of running time on a 32-node Mark III. The algorithm we used is based on ray tracing. The problem is very irregular; this led us to use a scattered block decomposition. We achieved the performance needed for our purposes, but did not gain large speedups. The feature of the machine that was essential for our calculation was its large memory, because of the need for high resolution. Two of the cases we studied are illustrated in Figure 7.9 and Figure 7.10: Areas on the source plane that produce one, three, five, or seven images, and the respective image regions on the image plane can be seen. An interesting example of an extended source is also shown in each case. A detailed exposition of our results and a description of our algorithm for a concurrent machine are contained in [Kochanek:88a] and [Apostolakis:88d], respectively.

7.4.1 Credits

This work was done by John Apostolakis and Chris Kochanek.

7.5 Parallel Random Number Generators

Many important algorithms in science and engineering are of the Monte Carlo type. This means that they employ pseudo-random number generators to simulate physical systems which are inherently probabilistic or statistical in nature. At other times, Monte Carlo is used to getting a fast approximation to what is actually a large, deterministic computation. Examples of

Figure 7.9: Part A shows the areas of the source plane that produce different numbers of images. Part B is a map of the areas of the image plane with negative amplification, i.e., flipped images, and positive amplification. Part C is a similar plot of the image plane, separating the areas by the total number of images of the same source. An example extended source is shown in A, whose images can be seen in Part B and Part C.

Figure 7.10: Part A shows the areas of the source plane that produce different numbers of images. Part B is a map of the areas of the image plane with negative amplification, i.e., flipped images, and positive amplification. Part C is a similar plot of the image plane, separating the areas by the total number of images of the same source. An example extended source is shown in A, whose images can be seen in Part B and Part C.

Figure 7.11: The figure and caption should be taken from Figure 12-1 of [Fox:88a].

this are Lattice Gauge computations (Section 4.2) and Simulated Annealing methods (Section 11.2.5. Section 11.3).

Even for a sequential algorithm, the question of correlations between members of the pseudo-random number sequence is non-trivial. In the parallel case, at least for the popular linear congruential algorithm, it is easy for the parallel algorithm to exactly mimic what a sequential algorithm would do. This means that the parallel case can be reduced to the well-understood sequential case.

The fundamental idea is for the processors of an N processor concurrent computer to each compute only the Nth number of the sequential random number sequence. The parallel sequences are staggered and interleaved so that the parallel computer exactly reproduces the sequential sequence. Figure 7.11 illustrates what happens in the parallel case versus the sequential case for a four processor concurrent computer.

Chapter 12 of [Fox:88a] has an extensive discussion of the parallel algorithm. This reference also has a discussion of what to do to achieve exact matching between parallel and sequential computations in more complex applications.

Chapter 8

Full Matrix Algorithms and Their Applications

8.1 History and Problem

*** Contribution needed from G. C. Fox

8.2 Full and Banded Matrix Algorithms

Concurrent matrix algorithms were among the first to be studied on the hypercubes at Caltech [Fox:82a], and have also been intensively studied at other institutions, notably Yale [Ipsen:87b], [Johnsson:87b], [Johnsson:89a] [Saad:85a], and Oak Ridge National Laboratory [Geist:86a], [Geist:89a], [Romine:87a], [Romine:90a]. The motivation for this interest is the fact that matrix algorithms play a prominent role in many scientific and engineering computations.

Early work at Caltech in this area (1983–1987) focused on specific algorithms, such as matrix multiplication, matrix-vector products, and LU decomposition. A major issue in determining the optimal algorithm for these problems is choosing a decomposition which has good load balance and low communication overhead. Many matrix algorithms proceed in a series of steps in which rows and/or columns are successively made inactive. The scattered decomposition described in Section 8.2.2 is usually used to balance the load in such cases. The block decomposition, also described in Section 8.2.2, generally minimizes the amount of data communicated, but results in sending several short messages rather than a few longer messages.

Thus, a block decomposition is optimal for a multiprocessor with low message latency, or startup cost, such as the Caltech/JPL Mark II hypercube. For machines with high message latency, such as the Intel iPSC/1, a row decomposition may be preferable. The best decomposition, therefore, depends crucially on the characteristics of the concurrent hardware.

In recent years (1988–1990), interest has centered on the development of libraries of concurrent linear algebra routines. As discussed in Section 8.2.6, two approaches have been followed at Caltech. One approach by Fox, *et al.* has led to a library of routines that are optimal for low latency, homogeneous hypercubes, such as the Caltech/JPL Mark II hypercubes. In contrast, Van de Velde has developed a library of routines that are generally sub-optimal, but which may be ported to a wider range of multiprocessor architectures, and are suitable for incorporation into programs with dynamically changing data distributions.

8.2.1 Matrix Decomposition

The data decomposition (or distribution) is a major factor in determining the efficiency of a concurrent matrix algorithm, so before detailing the research into concurrent linear algebra done at Caltech, we shall first introduce some basic decomposition strategies.

The processors of a concurrent computer can be uniquely labeled by $0, 1, \ldots, N_p - 1$, where N_p is the number of processors. A vector of length M may be decomposed over the processors by assigning the vector entry with global index m (where $0 \leq m < M$) to processor p, where it is stored as the ith entry in a local array. Thus, the decomposition of a vector can be regarded as a mapping of the global index, m, to an index pair, (p, i), specifying the processor number and local index.

For matrix problems, the processors are usually arranged as a P by Q grid. Thus, the grid consists of P rows of processors and Q columns of processors, and $N_p = PQ$. Each processor can be uniquely identified by its position, (p, q), on the processor grid. The decomposition of an $M \times N$ matrix can be regarded as the Cartesian product of two vector decompositions, μ and ν. The mapping μ decomposes the M rows of the matrix over the P processor rows, and ν decomposes the N columns of the matrix over the Q processor columns. Thus, if $\mu(m) = (p, i)$ and $\nu(n) = (q, j)$ then the matrix entry with global index (m, n) is assigned to the processor at position (p, q) on the processor grid, where it is stored in a local array with index (i, j).

Two common decompositions are the *linear* and *scattered* decomposi-

Figure 8.1: These eight figures show different ways of decomposing a 10×10 matrix. Each cell represents a matrix entry, and is labeled by the position, (p, q), in the processor grid of the processor to which it is assigned. To emphasize the pattern of decomposition, the matrix entries assigned to the processor in the first row and column of the processor grid are shown shaded. Figures (a) and (b) show linear and scattered row-oriented decompositions, respectively, for four processors arranged as a 4×1 grid ($P = 4$, $Q = 1$). In figures (c) and (d), the corresponding column-oriented decompositions are shown ($P = 1$, $Q = 4$). Figures (e)–(h) show linear and scattered block-oriented decompositions for 16 processors arranged as a 4×4 grid ($P = Q = 4$).

tions. The linear decomposition, λ, assigns contiguous entries in the global vector to the processors in blocks,

$$\lambda(m) = (\, p,\ m - pL - \min(p, R)\,), \tag{8.1}$$

where

$$p = \max\left(\left\lfloor \frac{m}{L+1} \right\rfloor, \left\lfloor \frac{m-R}{L} \right\rfloor \right) \tag{8.2}$$

and $L = \lfloor M/P \rfloor$ and $R = M \bmod P$. The scattered decomposition, σ, assigns consecutive entries in the global vector to different processors,

$$\sigma(m) = (\, m \bmod P, \lfloor m/P \rfloor\,) \tag{8.3}$$

Figure 8.1 shows examples of these two types of decomposition for a 10×10 matrix.

The mapping of processors onto the processor grid is determined by the programming methodology, which in turn depends closely on the concurrent

hardware. For machines such as the NCUBE-1 hypercube, it is advantageous to exploit any locality properties in the algorithm in order to reduce communication costs. In such cases, processors may be mapped onto the processor grid by a binary Gray code scheme [Fox:88a], which ensures that adjacent processors on the processor grid are directly connected by a communication channel. For machines such as the Symult 2010, for which the time to send a message between any two processors is almost independent of their separation in the hardware topology, locality of communication is not an issue, and the processors can be mapped arbitrarily onto the processor grid.

8.2.2 Basic Matrix Arithmetic

One of the first linear algebra algorithms implemented on the Caltech/JPL Mark II hypercube was the multiplication of two full matrices, A and B, to form the product, $C = AB$ [Fox:85b]. The algorithm uses a block-oriented, linear decomposition, which is optimal for machines with low message latency when the subblocks are (as nearly as possible) square. Let us denote by $A_{(p,q)}$ the subblock of A in the processor at position (p, q) of the processor grid, with a similar designation applying to the subblocks of B and C. Then, if the processor grid is square, *i.e.*, $P = Q = \sqrt{N_p}$, the matrix multiplication algorithm in block form is,

$$C_{(p,q)} = \sum_{r=0}^{P-1} A_{(p,r)}\, B_{(r,q)} \quad \text{for } p, q = 0, 1, \ldots, P-1 \tag{8.4}$$

The case in which $P \neq Q$ involves some extra bookkeeping, but does not change the concurrent algorithm in any essential way.

On the Mark II hypercube, communication cost increases with processor separation, so processors are mapped onto the processor grid using a binary Gray code scheme. Two types of communication are required at each stage of the algorithm, and both exploit the hypercube topology to minimize communication costs. Matrix subblocks are communicated to the processor above in the processor grid, and subblocks are broadcast along processor rows by a communication pipeline (see Figure 8.2).

The matrix multiplication algorithm has been modified for use on the Caltech/JPL Mark IIIfp hypercube [Hipes:89b]. The Mark II hypercube is a homogeneous machine in the sense that there is only one level in the memory hierarchy, *i.e.*, the local memory of each processor. However, each

Figure 8.2: A schematic representation of a pipeline broadcast for an eight-processor computer. White squares represent processors not involved in communication, and such processors are available to perform calculations. Shaded squares represent processors involved in communication, with the degree of shading indicating how much of the data have arrived at any given step. In the first six steps, those processor not yet involved in the broadcast can continue to perform calculations. Similarly, in steps 11 to 16, processors that are no longer involved in communicating can perform useful work since they now have all the data necessary to perform the next stage of the algorithm.

processor of the Mark IIIfp hypercube has a Weitek floating-point processor with a 64 kbyte data cache. To take full advantage of the high processing speed of the Weitek, data transfer between local memory and the Weitek data cache must be minimized. Since there are two levels in the memory hierarchy of each processor (local memory and cache), the Mark IIIfp is an inhomogeneous hypercube. The main computational task in each stage of the concurrent algorithm is to multiply the subblocks in each processor, and for large problems not all of the data will fit into the cache. The multiplication is, therefore, done in inner product form on the Weitek by further subdividing the subblocks in each processor. This intraprocessor subblocking allows the multiplication in each processor to be done in a number of stages, during each of which only the data needed for that stage is explicitly loaded into the cache.

A concurrent algorithm to perform the matrix-vector product $\mathbf{y} = A\mathbf{x}$ has also been implemented on the Caltech/JPL Mark II hypercube [Fox:88a]. Again, a block-oriented, linear decomposition is used for the matrix A. The vector $\mathbf{x}$ is decomposed linearly over the processors' columns, so that all the processors in the same processor column contain the same portion of $\mathbf{x}$. Similarly, at the end of the algorithm, the vector $\mathbf{y}$ is decomposed over the processor rows, so that all the processors in the same processor row contain the same portion of $\mathbf{y}$. In block form the matrix-vector product is,

$$\mathbf{y}_{(p)} = \sum_{q=0}^{Q-1} A_{(p,q)}\,\mathbf{x}_{(q)} \quad \text{for } p = 0, 1, \ldots, P-1 \tag{8.5}$$

As in the matrix multiplication algorithm, the concurrent matrix-vector product algorithm is optimal for low latency, homogeneous hypercubes if the subblocks of A are square.

8.2.3 Systems of Linear Equations

Factorization of Full Matrices

LU factorization of full matrices, and the closely-related Gaussian elimination algorithm, is widely used in the solution of linear systems of equations of the form $A\mathbf{x} = \mathbf{b}$. LU factorization expresses the coefficient matrix, A, as the product of a lower triangular matrix, L, and an upper triangular matrix, U. After factorization, the original system of equations can be written as a pair of triangular systems,

$$L\mathbf{y} = \mathbf{b} \quad \text{and} \quad U\mathbf{x} = \mathbf{y} \tag{8.6}$$

The first of the systems can be solved by forward reduction, and then back substitution can be used to solve the second system to give $\mathbf{x}$. If A is an $M \times M$ matrix, LU factorization proceeds in $M-1$ steps, in the kth of which column k of L and row k of U are found,

$$\begin{array}{lcll} \ell_{k,k} & = & 1 & \\ \ell_{k+i,k} & = & a_{k+i,k}/a_{k,k} & \text{for } 1 \le i < M \\ u_{k,k+j} & = & a_{k,k+j} & \text{for } 0 \le j < M \end{array}$$

and the entries of A in a "window" extending from column $k+1$ to $M-1$ and row $k+1$ to $M-1$ are updated,

$$a_{k+i,k+j} = a_{k+i,k+j} - \ell_{k+i,k}\, u_{k,k+j} \quad \text{for } 1 \le i,j < M \tag{8.7}$$

Partial pivoting is usually performed to improve numerical stability. This involves reordering the rows or columns of A.

In the absence of pivoting, the row and column oriented decompositions involve almost the same amounts of communication and computation. However, the row-oriented approach is generally preferred as it is more convenient for the back substitution phase [Chu:87a], [Geist:86a], although column-based algorithms have been proposed [Li:87a], [Moler:86a]. A block-oriented decomposition minimizes the amount of data communicated, and is the best approach on hypercubes with low message latency. However, since the block decomposition generally involves sending shorter messages, it is not suitable for machines with high message latency. In all cases, pipelining is the most efficient way of broadcasting rows and columns of the matrix since it minimizes the idle time that a processor must wait when participating in a broadcast, and effectively overlaps communication and calculation.

Load balance is an important issue in LU factorization. If a linear decomposition is used, the computation will be imbalanced and processors will become idle once they no longer contain matrix entries in the computational window. A scattered decomposition is much more effective in keeping all the processors busy, as shown in Figure 8.3. The load imbalance is least when a scattered block-oriented decomposition is used.

At Caltech, Van de Velde has investigated LU factorization of full matrices for a number of different pivoting strategies, and for various types of matrix decomposition on the Intel iPSC/2 hypercube and the Symult 2010 [Velde:87a]. One observation based on this work was that if a linear decomposition is used, then in many cases pivoting results in a faster algorithm than with no pivoting, since the exchange of rows effectively re-arranges

Figure 8.3: The shaded area in these two figures shows the computational window at the start of step three of the LU factorization algorithm. In (a) we see that by this stage the processors in the first row and column of the processor grid have become idle if a linear block decomposition is used. In contrast, in (b) we see that all processors continue to be involved in the computation if a scattered block decomposition is used.

the decomposition, resulting in better load balance. Van de Velde also introduces a clever enhancement to the standard concurrent partial pivoting procedure. To illustrate this, consider partial pivoting over rows. Usually, only the processors in a single processor column are involved in the search for the pivot candidate, and the other processors are idle at this time. In Van de Velde's multirow pivoting scheme, in each processor column a search for a pivot is conducted concurrently within a randomly selected column of the matrix. This incurs no extra cost compared with the standard pivoting procedure, but improves the numerical stability. A similar multicolumn pivoting scheme can be used when pivoting over columns. Van de Velde concludes from his extensive experimentation with LU factorization schemes that a scattered decomposition generally results in a more efficient algorithm on the iPSC/2 and Symult 2010, and his work illustrates the importance of decomposition and pivoting strategy in determining load balance, and hence concurrent efficiency.

LU Factorization of Banded Matrices

Aldcroft, *et al.* [Aldcroft:88a] have investigated the solution of linear systems of equations by LU factorization, followed by forward elimination and back substitution, when the coefficient matrix, A, is an $M \times M$ matrix of

Figure 8.4: Schematic representation of step k of LU factorization for an $M \times M$ matrix, A, with bandwidth w. The $m \times m$ computational window is shown as a dark shaded square, and matrix entries in this region are updated at step k. The light shaded part of the band above and to the left of the window has already been factorized, and in an in-place algorithm contains the appropriate columns and rows of L and U. The unshaded part of the band below and to the right of the window has not yet been modified. The shaded region of the matrix B represents of $m \times n_b$ window updated a step k of forward reduction, and in step $M - k - 1$ of back substitution.

bandwidth $w = 2m-1$. The case of multiple right hand sides was considered, so the system may be written as $AX = B$, where X and B are $M \times n_b$ matrices. The LU factorization algorithm for banded matrices is essentially the same as that for full matrices, except that the computational window containing the entries of A updated in each step is different. If no pivoting is performed, the window is of size $m \times m$ and lies along the diagonal, as shown in Figure 8.4. If partial pivoting over rows is performed, then fill-in will occur, and the window may attain a maximum size of $m \times (2m - 1)$. In the work of Aldcroft, *et al.* the size of the window was allowed to vary dynamically. This involved some additional bookkeeping, but is more efficient than working with a fixed window of the maximum size. Additional complications arise from only storing the entries of A within the band in order to reduce memory usage.

As in the full matrix case, good load balance is ensured by using a scattered block decomposition for the matrices. As noted previously, this choice of decomposition also minimizes communication cost on low latency multiprocessors, such as the Caltech/JPL Mark II hypercube used in this work,

but may not be optimal for machines in which the message startup cost is substantial.

A comparison between an analytic performance model and results on the Caltech/JPL Mark II hypercube shows that the concurrent overhead for the LU factorization algorithm falls to zero as $1/\hat{m}$, where $\hat{m} = \lfloor m/\sqrt{N_p} \rfloor$. This is true in both the pivoting and non-pivoting cases. Thus, the LU factorization algorithm scales well to larger machines.

8.2.4 The Gauss-Jordan Method

Hipes has studied the use of the Gauss-Jordan (GJ) algorithm as a means of solving systems of linear equations [Hipes:88c]. On a sequential computer, LU factorization followed by forward reduction and back substitution is preferable over GJ for solving linear systems since the former has a lower operation count. Another apparent drawback of GJ is that it has generally been believed that the right hand sides must be available *a priori*, which in applications requiring the solution for multiple right hand sides is a handicap. Hipes' work has shown this not to be the case, and that a well-written, parallel GJ solver is significantly more efficient than using LU factorization with triangular solvers on hypercubes.

As noted by Gerasoulis, *et al.* [Gerasoulis:88a], GJ does not require the solution of triangular systems. Since the solution of such systems features an outer loop of fixed length and two inner loops of decreasing length, whereas GJ has two outer fixed length loops and only one inner loop of decreasing length. GJ is, therefore, intrinsically more parallel than the LU solver, and its better load balance compensates for its higher operation count. Hipes has pointed out that the multipliers generated in the GJ algorithm can be saved where zeros are produced in the coefficient matrix. The entries in the coefficient matrix are, therefore, overwritten by the GJ multipliers, and we shall call this the GJ factorization (although we are not actually expressing the original matrix A as the product of two matrices). It is now apparent that the right hand side matrix does *not* have to be known in advance, since a solution can be obtained using the previously computed multipliers. Another factor, noted by Hipes, favoring the use of the GJ solver on a multiprocessor is the larger grain size maintained throughout the GJ factorization and solution phases, and the lower communication cost in the GJ solution phase.

Hipes has implemented his GJ solver on the Caltech/JPL Mark III and NCUBE-1 hypercubes, and compared the performance with the usual LU

solver [Hipes:89d]. In the GJ factorization, a scattered column decomposition is used, similar to that shown in Figure 8.1(d). This ensures good load balance as columns become eliminated in the course of the algorithm. In the LU factorization, both rows and columns are eliminated so a scattered block decomposition is used. On both machines, it was found that the GJ approach is faster for sufficiently many right hand sides.

8.2.5 Other Matrix Algorithms

Hipes has also compared the Gaussian-Jordan (GJ) and Gaussian Elimination (GE) algorithms for finding the inverse of a matrix [Hipes:88a]. This work was motivated by an application program that integrates a special system of crdinary differential equations that arise in chemical dynamics simulations [Hipes:87a], [Kuppermann:86a]. The sequential GJ and GE algorithms have the same operation count for matrix inversion. However, Hipes found that parallel GJ inversion has a more homogeneous load distribution, and requires fewer communication calls than GE inversion, and so should result in a more efficient parallel algorithm. Hipes has compared the two methods on the Caltech/JPL Mark II hypercube, and as expected found that GJ inversion algorithm to be the fastest.

Fox and Furmanski have also investigated matrix algorithms at Caltech [Furmanski:88b]. Among the parallel algorithms they discuss is the power method for finding the largest eigenvalue, and corresponding eigenvector, and a matrix A. This starts with an initial guess, $\mathbf{x}_0$, at the eigenvector, and then generates subsequent estimates using

$$\mathbf{y}_k = A\mathbf{x}_k, \quad \text{and} \quad \mathbf{x}_{k+1} = \frac{\mathbf{y}_k}{|\mathbf{y}_k|} \quad \text{for} \quad k = 0, 1, \ldots \tag{8.8}$$

As k becomes large, $|\mathbf{y}_k|$ tends to the eigenvalue with the largest absolute value (except for a possible sign change), and $\mathbf{x}_k$ tends to the corresponding eigenvector. Since the main component of the algorithm is matrix-vector multiplication, it can be done as discussed in Section 8.2.2.

A more challenging algorithm to parallelize is the tridiagonalization of a symmetric matrix by Householder's method, which involves the application of a series of rotations to the original matrix. Although the basic operations involved in each rotation are straightforward (matrix-vector multiplication, scalar products, and so on), special care must be taken to balance the load. This is particularly difficult since the symmetry of the matrix A means that the basic structure being processed is triangular, and this is decomposed into

a set of local triangular matrices in the individual processors. Load balance is optimized by scattering the rows over the processors, and the algorithm requires vectors to be broadcast and transposed.

8.2.6 Concurrent Linear Algebra Libraries

Since matrix algorithms play such an important role in scientific computing, it is desirable to develop a library of linear algebra routines for concurrent multiprocessors. Ideally, these routines should be optimal and general-purpose, *i.e.*, portable to a wide variety of multiprocessors. Unfortunately, these two objectives are antagonistic, and an algorithm that is optimal on one machine will often not be optimal on another machine. Even among hypercubes it is apparent that the optimal decomposition, and hence the optimal algorithm, depends on the message latency, with a block decomposition being best for low latency machines, and a row decomposition often being best for machines with high latency. Another factor to be considered is that often a matrix algorithm is only part of a larger application code. Thus, the data decomposition before and after the matrix computation may not be optimal for the matrix algorithm itself. We are faced with the choice of either transforming the decomposition before and after the matrix computation so that the optimal matrix algorithm can be used, or leaving the decomposition as it is and using a sub-optimal matrix algorithm. To summarize, the main issues that must be addressed are:

- Optimal or general-purpose routines,
- algorithms with data transformation or sub-optimal algorithms with no data transformation.

Two approaches to designing linear algebra libraries have been followed at Caltech. Fox, Furmanski, and Walker choose optimality as the most important concern in developing a set of linear algebra routines for low latency, homogeneous hypercubes, such as the Caltech/JPL Mark II hypercube. These routines feature the use of the scattered decomposition to ensure load balance, and to minimize communication costs. Transformations between decompositions are performed using the *comutil* library of global communication routines [Angus:90a], [Fox:88h], [Furmanski:88b]. This approach was mainly dictated by historical factors, rather than being a considered design decision—the hypercubes used most at Caltech up to 1987 were homogeneous and had low latency.

A different, and probably more useful approach, has been taken at Caltech by Van de Velde [Velde:89b] who opted for general-purpose library routines. The decomposition currently in use is passed to a routine through its argument list, so in general the decomposition is not changed and a sub-optimal algorithm is used. The main advantage of this approach is that it is decomposition-independent and allows portability of code between a wide variety of multiprocessors. Also, the sub-optimality of a routine must be weighed against the possibly large cost of transforming the data decomposition, so sub-optimality does not necessarily result in a slower algorithm if the time to change the decomposition is taken into account.

Occasionally, it may be advantageous to change the decomposition, and most changes of this type are what Van de Velde calls *orthogonal*. In an orthogonal redistribution of the data, each pair of processors exchanges the same amount of data. Van de Velde has shown [Velde:88d] that any orthogonal redistribution can be performed by the following sequence of operations;

Local permutation – Global transposition – Local permutation

A local permutation merely involves re-indexing the local data within individual processors. If we have P processors and P data items in each processor then the global transposition, τ, takes the item with local index i in processor p and sends it to processor i where it is stored with local index p. Thus,

$$\tau((p, i)) = (i, p) \tag{8.9}$$

Van de Velde's *transpose* routine is actually a generalization of the hypercube-specific *index* routine in the *comutil* library.

Van de Velde has implemented his linear algebra library on the Intel iPSC/2 and the Symult 2010, and has used it in investigations of concurrent LU and QR factorization algorithms [Velde:87a], [Velde:89b], and in studies of invariant manifolds of dynamical systems [Lorenz:89a], [Velde:90b].

8.2.7 Credits and References

Work on the concurrent Gauss-Jordan algorithm was mostly done by Paul Hipes. Eric Van de Velde developed the linear algebra library discussed in Section 8.2.6, and collaborated with Jens Lorenz in their work on invariant manifolds. Many of the other current algorithms were devised by Geoffrey Fox. Wojtek Furmanski and David Walker worked on routines

for transforming decompositions. The implementation of the banded LU solver on the Caltech/JPL Mark II hypercube was done by Tom Aldcroft, Arturo Cisneros, and David Walker.

8.3 Quantum Mechanical Reactive Scattering Using a High Performance Parallel Computer

8.3.1 Introduction

There is considerable current interest in performing accurate quantum mechanical three-dimensional reactive scattering cross section calculations. Accurate solutions have, until recently, proved to be difficult and computationally expensive to obtain, in large part due to the lack of sufficiently powerful computers. Prior to the advent of supercomputers, one could only solve the equations of motion for model systems or for sufficiently light atom-diatom systems at low energy [Schatz:75a], [Schatz:76a], [Schatz:76b]. As a result of the current development of efficient methodologies and increased access to supercomputers, there has been a remarkable surge of activity in this field. The use of symmetrized hyperspherical coordinates [Kuppermann:75a] and of the local hyperspherical surface function formalism [Hipes:87a], [Kuppermann:86a], [Ling:75a] has proven to be a successful approach to solve the three-dimensional Schrödinger equation [Cuccaro:89a], [Cuccaro:89b], [Hipes:87a], [Kuppermann:86a]. However, even for modest reactive scattering calculations, the memory and CPU demands are so great that CRAY-type supercomputers will soon be limiting progress.

In this section, we show how quantum mechanical reactive scattering calculations can be structured so as to use MIMD-type parallel computer architectures efficiently. We present a concurrent algorithm for calculating local hyperspherical surface functions (LHSF) and use a parallelized version [Hipes:88b] of Johnson's logarithmic derivative method [Johnson:73a], [Johnson:77a], [Johnson:79a], modified to include the improvements suggested by Manolopoulos [Manolopoulos:86a], for integrating the resulting coupled channel reactive scattering equations. We compare the results of scattering calculations on the Caltech/JPL Mark IIIfp 64-processor hypercube for the $H + H_2$ system $J = 0, 1, 2$ partial waves on the LSTH [Liu:73a], [Siegbahn:78a], [Truhlar:78a], [Truhlar:79a] potential energy surface with those of calculations done on a CRAY X-MP/48 and a CRAY-2. Both accuracy and performance are discussed, and speed estimates are made for

the Mark IIIfp 128-processor hypercube soon to become available and compared with those of the San Diego Supercomputer Center CRAY Y-MP/864 machine which has recently been put into operation.

8.3.2 Methodology

The detailed formulation of reactive scattering based on hyperspherical coordinates and local variational hyperspherical surface functions (LHSF) is discussed elsewhere [Kuppermann:86a], [Hipes:87a], and [Cuccaro:89a]. We present a very brief review to facilitate the explanation of the parallel algorithms.

For a triatomic system, we label the three atoms A_α, A_β and A_γ. Let (λ, ν, κ) be any cyclic permutation of the indices (α, β, γ). We define the λ coordinates, the mass-scaled [Delves:59a], [Delves:62a] internuclear vector $\mathbf{r}_\lambda$ from A_ν to A_κ, and the mass-scaled position vector $\mathbf{R}_\lambda$ of A_λ with respect to the center of mass of $A_\nu A_\kappa$ diatom. The symmetrized hyperspherical coordinates [Kuppermann:75a] are the hyper-radius $\rho = (R_\lambda^2 + r_\lambda^2)^{1/2}$, and a set of five angles ω_λ, γ_λ, θ_λ, ϕ_λ and ψ_λ, denoted collectively as ζ_λ. The first two of these are in the range 0 to π and are respectively $2 \arctan \frac{r_\lambda}{R_\lambda}$ and the angle between $\mathbf{R}_\lambda$ and $\mathbf{r}_\lambda$. The angles θ_λ, ϕ_λ are the polar angles of $\mathbf{R}_\lambda$ in a space-fixed frame and ψ_λ is the tumbling angle of the $\mathbf{R}_\lambda$, $\mathbf{r}_\lambda$ half-plane around its edge $\mathbf{R}_\lambda$. The hamiltonian $\hat{H}_\lambda$ is the sum of a radial kinetic energy operator term in ρ, and the surface hamiltonian $\hat{h}_\lambda$, which contains all differential operators in ζ_λ and the electronically adiabatic potential $V(\rho, \omega_\lambda, \gamma_\lambda)$. $\hat{h}_\lambda$ depends on ρ parametrically and is therefore the "frozen" hyperradius part of $\hat{H}_\lambda$.

The scattering wave function $\Psi^{JM\Pi\Gamma}$ is labelled by the total angular momentum J, its projection M on the laboratory-fixed Z axis, the inversion parity Π with respect to the center of mass of the system and the irreducible representation Γ of the permutation group of the system (P_3 for $H + H_2$) to which the electronuclear wave function, excluding the nuclear spin part [Lepetit:90a], [Lepetit:90b], belongs. It can be expanded in terms of the LHSF $\Phi^{JM\Pi\Gamma}$, defined below, and calculated at the values $\bar{\rho}_q$ of ρ:

$$\Psi_i^{JM\Pi\Gamma}(\rho, \zeta_\lambda) = \sum_n b_{ni}^{\Pi\Gamma}(\rho; \bar{\rho}_q)\, \Phi_n^{JM\Pi\Gamma}(\zeta_\lambda; \bar{\rho}_q) \qquad (8.10)$$

The index i is introduced to permit consideration of a set of many linearly independent solutions of the Schrödinger equation corresponding to distinct

initial conditions which are needed to obtain the appropriate scattering matrices.

The LHSF $\Phi_n^{JM\Pi\Gamma}(\zeta_\lambda;\bar{\rho}_q)$ and associated energies $\epsilon_n^{J\Pi\Gamma}(\bar{\rho}_q)$ are respectively the eigenfunctions and eigenvalues of the surface hamiltonian $\hat{h}_\lambda$. They are obtained using a variational approach [Cuccaro:89a]. The variational basis set consists of products of Wigner rotation matrices $D^J_{M\Omega}(\phi_\lambda,\theta_\lambda,\psi_\lambda)$, associated Legendre functions of γ_λ and functions of ω_λ which depend parametically on $\bar{\rho}_q$ and are obtained from the numerical solution of one-dimensional eigenvalue-eigenfunction differential equations in ω_λ involving a potential related to $V(\bar{\rho},\omega_\lambda,\gamma_\lambda)$.

The variational method leads to an eigenvalue problem with coefficient and overlap matrices $h^{J\Pi\Gamma}(\bar{\rho}_q)$ and $s^{J\Pi\Gamma}(\bar{\rho}_q)$ and whose elements are five-dimensional integrals involving the variational basis functions.

The coefficients $b_{ni}^{J\Pi\Gamma}(\rho;\bar{\rho}_q)$ defined by Equation 8.10 satisfy a coupled set of second order differential equations involving an interaction matrix $\mathcal{I}^{J\Pi\Gamma}(\rho;\bar{\rho}_q)$ whose elements are defined by

$$\begin{aligned}[\mathcal{I}^{J\Pi\Gamma}(\rho;\bar{\rho}_q)]_n^{n'} &= \left\langle \Phi_n^{JM\Pi\Gamma}(\zeta_\lambda;\bar{\rho}_q)\right| V(\rho,\omega_\lambda,\gamma_\lambda) \\ &- (\bar{\rho}_q/\rho)^2 V(\bar{\rho}_q,\omega_\lambda,\gamma_\lambda)\left|\Phi_{n'}^{JM\Pi\Gamma}(\zeta_\lambda;\bar{\rho}_q)\right\rangle \quad (8.11)\end{aligned}$$

The configuration space ρ,ζ_λ is divided in a set of Q hyperspherical shells $\rho_q \le \rho \le \rho_{q+1}\,(q=1,2,\ldots,Q)$ within each of which we choose a value $\bar{\rho}_q$ used in expansion 8.10.

When changing from the LHSF set at $\bar{\rho}_q$ to the one at $\bar{\rho}_{q+1}$, neither $\Psi_i^{JM\Pi\Gamma}$ nor its derivative with respect to ρ should change. This imposes continuity conditions on the $b_{ni}^{J\Pi\Gamma}$ and their ρ-derivatives at $\rho=\rho_{q+1}$, involving the overlap matrix $\mathcal{O}^{J\Pi\Gamma}(\bar{\rho}_{q+1},\bar{\rho}_q)$ between the LHSF evaluated at $\bar{\rho}_q$ and $\bar{\rho}_{q+1}$

$$[\mathcal{O}^{J\Pi\Gamma}(\bar{\rho}_{q+1},\bar{\rho}_q)]_n^{n'} = \left\langle \Phi_n^{JM\Pi\Gamma}(\zeta_\lambda;\bar{\rho}_{q+1})\right|\left.\Phi_{n'}^{JM\Pi\Gamma}(\zeta_\lambda;\bar{\rho}_q)\right\rangle \qquad (8.12)$$

The five-dimensional integrals required to evaluate the elements of $h^{J\Pi\Gamma}$, $s^{J\Pi\Gamma}$, $\mathcal{I}^{J\Pi\Gamma}$ and $\mathcal{O}^{J\Pi\Gamma}$ are performed analytically over ϕ_λ, θ_λ and ψ_λ and by two-dimensional numerical quadratures over γ_λ and ω_λ. These quadratures account for 90% of the total time needed to calculate the LHSF $\Phi_n^{JM\Pi\Gamma}$ and the matrices $\mathcal{I}^{J\Pi\Gamma}$ and $\mathcal{O}^{J\Pi\Gamma}$.

The system of second-order ordinary differential equations in the $b_{ni}^{J\Pi\Gamma}$ is integrated as an initial value problem from small values of ρ to large values

using Manolopoulos' logarithmic derivative propagator [Manolopoulos:86a]. Matrix inversions account for more than 90% of the time used by this propagator. All aspects of the physics can be extracted from the solutions at large ρ by a constant ρ projection [Hipes:87a], [Hood:86a], and [Kuppermann:86a].

8.3.3 Parallel Algorithm

The computer used for this work is a 64-processor Mark IIIfp hypercube. The Crystalline Operating System (CrOS)-channel-addressed synchronous communication provides the library routines to handle communications between nodes [Fox:85h], [Fox:85d], and [Fox:88a]. The programs are written in C programming language except for the time-consuming two-dimensional quadratures and matrix inversions, which are optimized in assembly language.

The hypercube was configured as a two-dimensional array of processors. The mapping is done using binary Gray codes [Gilbert:58a], [Fox:88a], and [Salmon:84b] which gives the Cartesian coordinates in processor space and communication channel tags for a processor's nearest neighbors.

We mapped the matrices into processor space by local decomposition. Let N_r and N_c be the number of processors in the rows and columns of the hypercube configuration, respectively. Element $A(i,j)$ of an $M \times M$ matrix is placed in processor row $P_r = int\left(\frac{i \times N_r}{M}\right)$ and column $P_c = int\left(\frac{j \times N_c}{M}\right)$, where $int\ x$ means the integer part of x.

The parallel code implemented on the hypercube consists of five major steps. Step one constructs, for each value of $\bar{\rho}_q$, a primitive basis set composed of the product of Wigner rotation matrices, associated Legendre functions, and the numerical one-dimensional functions in ω_λ mentioned in Section 8.3.2 and obtained by solving the corresponding one-dimensional eigenvalue-eigenvector differential equation using a finite difference method. This requires that a subset of the eigenvalues and eigenvectors of a tridiagonal matrix be found.

A bisection method [Fox:84g], [Ipsen:87a], and [Ipsen:87c] which accomplishes the eigenvalue computation using the TRIDIB routine from EISPACK [Smith:76a] was ported to the Mark IIIfp. This implementation of the bisection method allows computation of any number of consecutive eigenvalues specified by their indices. Eigenvectors are obtained using the EISPACK inverse iteration routine TINVIT with modified Gram-Schmidt orthogonalization. Each processor solves independent tridiagonal eigenproblems since the number of eigenvalues desired from each tridiagonal system is small.

but there are a large number of distinct tridiagonal systems. To achieve load balancing, we distributed subsets of the primitive functions among the processors in such a way that no processor computes greater than one eigenvalue and eigenvector more than any other. These large grain tasks are most easily implemented on MIMD machines; SIMD (Single Instruction Multiple Data) machines would require more extensive modifications and would be less efficient because of the sequential nature of effective eigenvalue iteration procedures. The one-dimensional bases obtained are then broadcast to all the other nodes.

In step two, a large number of two-dimensional quadratures involving the primitive basis functions which are needed for the variational procedure are evaluated. These quadratures are highly parallel procedures requiring no communication overhead once each processor has the necessary subset of functions. Each processor calculates a subset of integrals independently.

Step three assembles these integrals into the real symmetric dense matrices $s^{J\Pi\Gamma}(\bar{\rho}_q)$ and $h^{J\Pi\Gamma}(\bar{\rho}_q)$ which are distributed over processor space. The entire spectrum of eigenvalues and eigenvectors for the associated variational problem is sought. With the parallel implementation of the Householder method [Fox:84h], [Patterson:86a], this generalized eigensystem is tridiagonalized and the resulting single tridiagonal matrix is solved in each processor completely with the QR algorithm [Wilkinson:71a]. The QR implementation is purely sequential since each processor obtains the entire solution to the eigensystem. However, only different subsets of the solution are kept in different processors for the evaluation of the interaction and overlap matrices in step four. This part of the algorithm is not time-consuming and the straightforward sequential approach was chosen. It has the further effect that the resulting solutions are fully distributed, so no communication is required.

Step four evaluates the two-dimensional quadratures needed for the interaction $\mathcal{I}^{J\Pi\Gamma}(\rho;\bar{\rho}_q)$ and overlap $\mathcal{O}^{J\Pi\Gamma}(\bar{\rho}_{q+1};\bar{\rho}_q)$ matrices. The same type of algorithms are used as were used in step two. By far, the most expensive part of the sequential version of the surface function calculation is the calculation of the large number of two-dimensional numerical integrals required by steps two and four. These steps are, however, highly parallel and well suited for the hypercube.

Step five uses Manolopoulos' [Manolopoulos:86a] algorithm to integrate the coupled linear ordinary differential equations. The parallel implementation of this algorithm is discussed elsewhere [Hipes:88b]. The algorithm is dominated by parallel Gauss-Jordan matrix inversion and is I/O inten-

sive, requiring the input of one interaction matrix per integration step. To reduce the I/O overhead, a second source of parallelism is exploited. The entire interaction matrix (at all ρ) and overlap matrix (at all $\bar{\rho}_q$) data sets are loaded across the processors, and many collision energies are calculated simultaneously. This strategy works because the same set of data is used for each collision energy, and because enough main memory is available. Calculation of scattering matrices from the final logarithmic derivative matrices is not computationally intensive, and is done sequentially.

The program steps were all run on the Weitek coprocessor, which only supports 32-bit arithmetic. Experimentation has shown that this precision is sufficient for the work reported below. The 64-bit arithmetic hardware needed for larger calculations was installed after the present calculations were completed.

8.3.4 Results and Discussion

Accuracy:

Calculations were performed for the $H + H_2$ system on the LSTH surface [Liu:73a], [Siegbahn:78a], [Truhlar:78a], [Truhlar:79a] for partial waves with total angular momentum $J = 0, 1, 2$ and energies up to 1.6 eV. Flux is conserved to better than 1% for $J = 0$, 2.3% for $J = 1$ and 3.6% for $J = 2$ for all open channels over the entire energy range considered.

To illustrate the accuracy of the 32-bit arithmetic calculations, the scattering results from the Mark IIIfp with 64 processors are shown in Figure 8.5 for $J = 0$, in which some transition probabilities as a function of the total collision energy, E are plotted. The differences between these results, and those obtained using a CRAY X-MP/48 and a CRAY-2, do not exceed 0.004 in absolute value over the energy range investigated.

Timing and Parallel Efficiency: In Table 8.1 and Table 8.2, we present the timing data on the 64-processor Mark IIIfp, a CRAY X-MP/48 and a CRAY 2, for both the surface function code (including calculation of the overlap $\mathcal{O}^{J\Pi\Gamma}$ and interaction $\mathcal{I}^{J\Pi\Gamma}$ matrices) and the logarithmic derivative propagation code. For the surface function code, the speeds on the first two machines are about the same. The CRAY 2 is 1.43 times faster than the Mark IIIfp and 1.51 times faster than the CRAY X-MP/48 for this code. The reason is that this program is dominated by matrix-vector multiplications which are done in optimized assembly code in all three machines. For this particular operation, the CRAY 2 is 2.03 times faster than

Figure 8.5: Probabilities as a function of total energy E (lower abscissa) and initial relative translational energy E_{00} (upper abscissa) for the $J = 0(0,0,0) \rightarrow (0,0,0)\ A_1$ symmetry transition in $H + H_2$ collisions on the LSTH potential energy surface. The symbol (v, j, Ω) labels an asymptotic state of the $H + H_2$ system in which v, j, and Ω are the quantum numbers of the initial or final H_2 states. The vertical arrows on the upper abscissa denote the energies at which the corresponding $H_2\,(v, j)$ states open up. The length of those arrows decreases as v spans the values 0, 1 and 2, and the numbers 0, 5, and 10 associated with the arrows define a labelling for the value of j. The number of LHSF used was 36 and the number of primitives used to calculate these surface functions was 80.

J	Mark IIIfp 64 processors		CRAY X-MP/48		CRAY 2	
	Time (hr)	Speed (Mflops)	Time (hr)	Speed (Mflops)	Time (hr)	Speed (Mflops)
0	0.71	100	0.74	96	0.49	145
1	2.88	112	3.04	106	2.01	160
2	5.60	124	5.94	117	3.96	176

Table 8.1: Performance of the surface function code. This code calculates the surface functions at the 51 values of $\bar{\rho}$ from 2.0 bohr to 12.0 bohr in steps of 0.2 bohr, the corresponding overlap matrices between consecutive values of $\bar{\rho}$ and the propagation matrices in ρ steps of 0.1 bohr. The number of primitives used for each J and described in the remaining figure captions, permits us to generate enough LHSF to achieve the accuracy described in the text.

the CRAY X-MP/48 whereas, for more memory-intensive operations, the CRAY 2 is slower than the CRAY X-MP/48 [Pfeiffer:88a]. A slightly larger primitive basis set is required on the Mark IIIfp in order to obtain surface function energies of an accuracy equivalent to that obtained with the CRAY machines. This is due to the lower accuracy of the 32-bit arithmetic of the former with respect to the 64-bit arithmetic of the latter.

The efficiency (ε) of the parallel LHSF code was determined using the definition $\varepsilon = \frac{T_1}{(N \times T_N)}$ where T_1 and T_N are respectively the implementation times using a single processor and N processors. The single processor times are obtained from runs performed after removing the overhead of the parallel code, *i.e.*, after removing the communication calls and some logical statements. Perfect efficiency ($\varepsilon = 1.0$) implies that the N-processor hypercube is N times faster than a single processor. In Figure 8.6, efficiencies for the surface function code (including the calculation of the overlap and interaction matrices) as a function of the size of the primitive basis set are plotted for 2, 4, 8, 16, 32 and 64 processor configurations of the hypercube. The global dimensions of the matrices used are chosen to be integer multiples of the number of processor rows and columns in order to insure load balancing among the processors. Because of the limited size of a single processor memory, the efficiency determination is limited to 32 primitives. As shown in Figure 8.6, the efficiencies increase monotonically and approach

	Mark IIIfp		CRAY X-MP/48	CRAY 2
	64 Processor Global Configuration	8 Clusters of 8 Processors		
Total time (hrs)	4.8	3.4	1.5	2.9
Total for 1 energy (min)	2.2	1.6	0.7	1.3
Efficiency	0.52	0.81	...	...
Speed (Mflops)	34.4	48.5	110	55.4

Table 8.2: Performance of the logarithmic derivative code. Based on a calculation using 245 surface functions and 131 energies, and a logarithmic derivative integration step of 0.01 bohr.

unity asymptotically as the size of the calculation increases. Converged results require large enough primitive basis sets so that the efficiency of the surface function code is estimated to be about 0.95 or greater.

The data for the logarithmic derivative code given in Table 8.2 for a 245 channel (*i.e.*, LHSF) example show that the Mark IIIfp has a speed about 62% to that of the CRAY 2, but only about 31% of that of the CRAY X-MP/48. This code is dominated by matrix inversions, which are done in optimized assembly code in all three machines. The reason for the slowness of the hypercube with respect to the CRAYs is that the efficiency of the parallel logarithmic derivative code is 0.52. This relatively low value is due to the fact that matrix inversions require a significant amount of inter-processor communication. Figure 8.7 displays efficiencies of the logarithmic derivative code as a function of the number of channels propagated for different processor configurations, as done previously for the Mark III [Hipes:88b], [Messina:90a] hypercubes. The data can be fit well by an operations count formula developed previously for the matrix inversion part of the code [Hipes:88a]; this formula can be used to extrapolate the data to larger numbers of processors or larger numbers of channels. It can be seen that for an 8-processor configuration, the code runs with an efficiency of 0.81. This observation suggested that we divide the Mark IIIfp into eight clusters of eight processors each, and perform calculations for different energies in different clusters. The corresponding timing information is also given

Figure 8.6: Efficiency of the surface function code (including the calculation of the overlap and interaction matrices) as a function of the global matrix dimension (*i.e.*, the size of the primitive basis set) for 2, 4, 8, 16, 32, and 64 processors. The solid curves are straight line segments connecting the data points for a fixed number of processors and are provided as an aid to examine the trends.

in Table 8.2. As can be seen from the last row of this table, the speed of the logarithmic derivative code using this configuration of the 64-processor Mark IIIfp is 48.5 Mflops, which is about 44% of that of the CRAY X-MP/48 and 88% of that of the CRAY 2. As the number of channels increases, the number of processors per cluster may be made larger in order to increase the amount of memory available in each cluster. The corresponding efficiency should continue to be adequate due to the larger matrix dimensions involved.

In the near future, the number of processors of the Mark IIIfp will be increased to 128 and the I/O system will be replaced by high performance CIO (concurrent I/O) hardware. The new Weitek coprocessors, installed since the present calculations were done, perform 64-bit floating point arithmetic at about the same nominal peak speed as the 32-bit boards. From the data in the present paper, it is possible to predict with good reliability the performance of this upgraded version of the Mark IIIfp. A CRAY Y-MP/864 has recently been installed at the San Diego Supercomputer Center. Initial speed measurements show that it is about two times faster than the CRAY X-MP/48 for the surface function code and 1.7 times faster for the logarithmic derivative code. In Table 8.3, we summarize the available or predicted speed information for the present codes for the current 64-processor and near future 128-processor Mark IIIfp, as well as the CRAY X-MP/48, CRAY 2 and

Figure 8.7: Efficiency of logarithmic derivative code as a function of the global matrix dimension (*i.e.*, the number of channels or LHSF) for 8, 16, 32, and 64 processors. The solid curves are straight line segments connecting the data points for a fixed number of processors, and are provided as an aid to examine the trends.

CRAY Y-MP/864 supercomputers. It can be seen that Mark IIIfp machines are competitive with all of the currently available CRAYs (operating as single processor machines). The results described in this paper demonstrate the feasibility of performing reactive scattering calculations with high efficiency in parallel fashion. As the number of processors continues to increase, such parallel calculations in systems of greater complexity will become practical in the not too distant future.

8.3.5 Acknowledgements

The work described in this paper was supported in part by DOE grant DE-AS03-83ER and Air Force Astronautics Laboratory contract F04611-86-K-0067. The calculations were performed on the 64-processor Mark IIIfp Caltech/JPL hypercube, the CRAY X-MP/48 and CRAY Y-MP/864 at the NSF San Diego Supercomputing Center, and the CRAY 2 at the Air Force Weapons Laboratory. We thank those institutions for their help. We also thank Dr. B. Lepetit and Professor Geoffrey Fox for useful discussions.

	Mark IIIfp		CRAY X-MP/48	CRAY 2	CRAY Y-MP/864
	64 Processors	128 Processors			
Surface function code for $J = 2$ (Mflops)	124	240	117	176	232
Logarithmic derivative code (Mflops)	48.5	127	110	55.4	187
Total main memory of computer (64 bit Mwords)	32	64	8	256	64

Table 8.3: Overall speed of reactive scattering codes on several machines.

8.4 Parallel Computational Electromagnetics

We are conducting research in parallel computational electromagnetics by applying the computational power of the hypercube parallel computing systems to the solution of large-scale electromagnetic scattering and radiation problems. A number of parallel analysis codes have been developed. Two are parallel implementations of standard production level EM analysis codes and the remaining are largely or entirely new. Included in the parallel implementations of existing codes is the widely used Numerical Electromagnetics Code (NEC-2) developed at Lawrence Livermore National Laboratory. Other codes include an integral equation formulation Patch code, a time domain finite difference code, a two-dimensional scalar finite elements code, and infinite and finite frequency selective surfaces codes. Currently, we are developing both a three-dimensional Frequency Domain Finite Elements code and a two-dimensional Coupled Approach code. In the Coupled Approach, one uses finite elements to represent the interior of a scattering object, and the boundary integrals for the exterior. Along with the analysis tools, we are developing an Electromagnetic Interactive Analysis Workstation as an integrated environment to aid in design and analysis. The workstation provides a general user interface for specification of an object to be analyzed and graphical representations of the results. The workstation is implemented on an Apollo DN4500 Color Graphics Workstation, and has access to a set of

hypercubes include eight-processor to 64-processor Mark IIIfp's and soon an eight-processor Intel iPSC/860 Hypercube [Calalo:89b].

One of the areas of current emphasis is the development of a fully three-dimensional finite element analysis tool. We will briefly describe this effort here. The finite element method is being used to compute solutions to open region electromagnetic scattering problems where the domain may be irregularly shaped and contain differing material properties. Such a scattering object may be composed of dielectric and conducting materials with possibly anisotropic and inhomogeneous dielectric properties. The domain is discretized by a mesh of polygonal (2-D) and polyhedral (3-D) elements with nodal points at the corners. The finite element solution which determines the field quantities at these nodal points is stated using the Helmholtz equation. It is derived from Maxwell's equations describing the incident and scattered field for a particular wave number, k. The two-dimensional equation for the out-of-plane magnetic field, H_z, is given by

$$\nabla \left(\frac{1}{\epsilon} \nabla H_z \right) + k^2 \mu H_z = 0 \tag{8.13}$$

where ϵ is the relative permittivity and μ is the relative magnetic permeability. The equation for the electric field is similarly stated interchanging ϵ and μ.

The open region problem is solved in a finite domain by imposing an artificial boundary condition for a circular boundary. For the two-dimensional case, we are applying the approach of Bayliss and Turkel [Bayliss:80a]. The cylindrical artificial boundary condition on scattered field, H_z^s (where $H_z^s = H_z - H_{\text{inc}}$), is given by

$$\frac{\partial H_z^s}{\partial \rho} = A(\rho) H_z^s + B(\rho) \frac{\partial^2 H_z^s}{\partial \varphi^2} \tag{8.14}$$

where ρ is the radius of artificial boundary, φ is the angular coordinate, and A and B are operators which are dependent on ρ.

The differential equation 8.13 can be converted to an integral equation by multiplying by a test function which has certain continuity properties. If the total field is expressed in terms of the incident and scattered fields, then we may substitute equation 8.14 to arrive at our weak form equation

$$\iint_{\Gamma} \left(\frac{1}{\epsilon} \nabla T \cdot \nabla H_z^s - k^2 \mu T H_z^s \right) dx dy - \oint_{\partial \Gamma} \frac{1}{\epsilon} T \left(A H_z^s + B \frac{\partial^2 H_z^s}{\partial \varphi^2} \right) dl = F \tag{8.15}$$

Figure 8.8: Domain decomposition of the finite element mesh into subdomains each of which are assigned to different hypercube processors.

where F is the excitation, which depends on the incident field.

$$F = \int\int_{\Gamma} \left(T\nabla(\frac{1}{\epsilon}\nabla H_z^{\text{inc}}) + k^2 \mu T H_z^{\text{inc}} \right) dxdy \tag{8.16}$$

Substituting the field and test function representations in terms of nodal basis functions into equation 8.15 forms a set of linear equations for the coefficients of the basis functions. The matrix which results from this finite element approximation is sparse with non-zero elements clustered about the diagonal.

The solution technique for the finite element problem is based on a domain decomposition. This decomposition technique divides the physical problem space among the processors of the hypercube. While elements are the exclusive responsibility of hypercube processors, the nodal points on the boundaries of the subdomains are shared between processors. Because shared nodal points require that there be communication between hypercube processors, it is important for processing efficiency to minimize the number of these shared nodal points.

The tedious process of specifying the finite element model to describe the geometry of the scattering object is greatly simplified by invoking the graphical editor, PATRAN-Plus, within the Hypercube Electromagnetics Interactive Analysis Workstation. The graphical input is used to generate the finite element mesh. Currently, we have implemented isoparametric three-node triangular, six-node triangular, and nine-node quadrilateral elements for the two-dimensional case and linear four-node tetrahedral elements for

Figure 8.9: Finite element mesh for a dielectric cylinder partitioned among eight hypercube processors.

the three-dimensional case.

Once the finite element mesh has been generated, the elements are allocated to hypercube processors with the aid of a partitioning tool which we have developed. In order to achieve good balance of load, each of the hypercube processors should receive approximately the same number of elements (which reflects the computation load) and the same number of subdomain edges (which reflects the communication requirement). The Recursive Inertial Partitioning (RIP) algorithm chooses the best bisection axis of the mesh based on calculated moments of inertia. Figure 8.9 illustrates one possible partitioning for a dielectric cylinder.

The finite element problem can be solved using several different strategies: iterative solution, direct solution, or a hybrid of the two. We will employ all of these techniques in our finite elements testbed. We use a preconditioned bi-conjugate gradients approach for iterative solutions and a Crout solver for the direct solution [Peterson:85c] [Peterson:86a]. In progress is the hybrid solver which will use Gaussian elimination locally within hypercube processors, and then bi-conjugate gradients to resolve the remaining degrees of freedom [Nour-Omid:87b].

The output from the finite elements code is displayed graphically at the Electromagnetics Interactive Analysis Workstation. In Figure 8.10 are plotted the real (on the left) and the imaginary (on the right) components of the total scalar field for a conducting cylinder of $ka = 50$. The absorbing boundary is placed at $kr = 62$. Figure 8.11 shows the plane wave propagation indicated by vectors in a rectangular box (no scatterer). The box

Figure 8.10: Results from 2D Scalar Finite Element Code

Figure 8.11: Test Case for 3D Code—No Scatter

is modeled using linear tetrahedral elements. Figure 8.12 shows the plane wave propagation (no scatterer) in a spherical domain, again using tetrahedral linear elements. The half slices show the internal fields. In the upper left is the x-component of the field, the lower left is the z-component, and on the right is the y-component with the fields shown as contours on the surface.

The speedups over the problem running on one processor are plotted for hypercube configurations ranging from one to 32 processors in Figure 8.13. The problem for this set of runs is a two-dimensional dielectric cylinder model consisting of 9,313 nodes.

The setup and solve portions of the total execution time demonstrate 87% and 81% efficiencies, respectively. The output portion where the results

Figure 8.12: Test Case for 3D—Planewave in Spherical Domain—No Scatter

Figure 8.13: Finite Element Execution Speedup vs. Hypercube Size

obtained by each processor are sent back to the workstation run at about 50% efficiency. The input routine exhibits no speedup and greatly reduced the overall efficiency, 63% of the code. Clearly, this is an area on which we now must focus. We have recently implemented the partitioning code on parallel. We now are also reducing the size of the input file by compressing the contents of the mesh data file and removing formatted reads and writes. Ultimately, we will want to have the actual mesh generation and partitioning done almost entirely within the parallel system.

We are currently exploring a number of accuracy issues with regard to the finite elements problem. Such issues include gridding density, element types, placement of artificial boundaries, and specification of basis functions. We are investigating outgoing wave boundary conditions; currently, we are using a modified Sommerfield radiation condition in three dimensions. In addition, we are exploring a number of higher order element types for three dimensions. Central to our investigations is the objective of developing analysis techniques for massive three-dimensional problems.

We have demonstrated that the parallel processing environments offered by the current coarse-grain MIMD architectures, such as the Mark IIIfp Hypercube, are very well-suited to the solution of large-scale electromagnetic scattering and radiation problems. We have developed a number of parallel EM analysis codes that currently run in production mode. These codes are being embedded in a Hypercube Electromagnetic Interactive Analysis Workstation. The workstation environment facilitates the specification of the model geometry and material properties, and the input of run parameters. The workstation also provides an ideal environment for graphically viewing the resulting currents and near- and far-fields. We are continuing to explore a number of issues to fully exploit the capabilities of this large-memory, high-performance computing environment. We are also investigating improved matrix solvers for both dense and sparse matrices, and have implemented out-of-core solving techniques which will prevent us from becoming memory limited. By establishing testbeds, such as the finite element one described here, we will continue to explore issues that will maintain computational accuracy while reducing the overall computation time for EM scattering and radiation analysis problems.

Chapter 9

Loosely Synchronous Problems

9.1 Problem Structure

*** Contribution needed from G. C. Fox

9.2 Thermal Convection in the Earth's Mantle

*** Contribution needed from G. Lyzenga

9.3 Tectonics of the Crust

*** Contribution needed from G. Lyzenga

9.4 Application of Hypercubes to Micromechanical Simulations

Geomorphology is the study of the small-scale surface evolution of the earth under the forces due to such agents as wind, water, gravity and ice. Understanding and prediction in geomorphology are dependent critically upon the ability to model the processes that shape the landscape. Because these processes in general are too complicated to describe on large scales in detail, it is necessary to adopt a system of hierarchical models in which the behavior of small systems is summarized by a set of rules in the next larger system;

in essence, these rules constitute a simplified algorithm for the physical processes in the smaller system that cannot be treated fully at larger scales. A significant fraction of the processes in geomorphology involve entrainment, transport and deposition of particulate matter. Where the intergrain forces become comparable to or greater than the forces due to the transporting agents, consideration of the properties of a granular material, a system of grains which collide with the slide against neighboring grains, is warranted. A micromechanical description of granular materials has proved difficult, except in energetic flow regimes [Haff:83a], [Jenkins:83a]. Thus, researchers have turned to dynamical and computer simulations at the level of individual grains in order to elucidate some of the basic mechanical properties of granular materials ([Cundall:79a], [Walton:83a] pioneered this simulation technique). In this article, we discuss the role that hypercube concurrent processing has played and is expected to play both in grain-level dynamical simulations and in relating these simulations to modeling the formation and evolution of landforms.

As an example of this approach to geomorphology, we shall consider efforts to model transport of sand by the wind based upon the grain-to-grain dynamics. Sand is transported by the wind primarily in saltation and in reptation [Bagnold:41a]. Saltating grains are propelled along the surface in short hops by the wind. Each collision between a saltating sand grain and the surface results in a loss of energy which is compensated, on the average, by energy acquired from the wind. Reptating grains are ejected from the sand surface by saltating grain-sand bed impacts; they generally come to rest shortly after returning to the sand surface.

Computer simulations of saltating grain impacts upon a loose grain bed were performed on an early version of the hypercube [Werner:88a], [Werner:88b]. Collisions between a single impacting grain and a box of 384 circular grains were simulated. The grains interact through stiff, inelastic compressional contact forces plus a Coulomb friction force. The equations of motion for the particles are integrated forward in time using a predictor-corrector technique. At each step in time, the program checks for contacts between particles and, where contacts exist, computes the contact forces. Dynamical simulations of granular materials are computationally intensive, because the time scale of the interaction between grains (tens of microseconds) is much smaller than the time scale of the simulation (order one second).

The simulation was decomposed on a hypercube by assigning the processors to regions of space lying on a rectangular grid. The computation time is

a combination of calculation time in each processor due to contact searches and to force computations, and of communication time in sending information concerning grain positions and velocities to neighboring processors for interparticle force calculations on processor boundaries. Because the force computation is complicated, the comunication time was found to be a negligible fraction of the total computation time for granular materials in which enduring intergrain contacts are dominant. The boundaries between processors are changed incrementally throughout the calculation in order to balance the computational load among the processors. The optimal decomposition has enough particles per processor to diminish the relative importance of statistical fluctuations in the load, and a system of boundaries which conforms as much as possible to the geometry of the problem. For grain-bed impacts, efficiencies between 0.89 and 0.97 were achieved [Werner:88a].

The results of the grain-bed impact simulations have facilitated treatment of two larger scale problems. A simulation of steady-state saltation in which calculation of saltating grain trajectories and modifications to the wind velocity profile, due to acceleration of saltating grains, were combined with a grain-bed impact distribution function derived from experiments and simulations. This simulation yielded such characteristics of saltation as flux and erosive potential [Werner:90a]. A simulation of the rearrangement of surface grains in reptation led to the formation of self-organized small-scale bedforms, which resemble wind ripples in both size and shape [Werner:90b]. Larger, more complicated ripple formation simulations and a simulation of sand dune formation, using a similar approach which is under development, are problems that will require the combination of processing power and memory not available on present supercomputers. Ripple and dune simulations are expected to run efficiently with a spatial decomposition on a hypercube.

Water is an important agent for the transport of sediment. Unlike windblown sand transport, underwater sand transport requires simultaneous simulation of the grains and the fluid because water and sand are similar in density. We are developing a grain/fluid mixture simulation code for a hypercube in which the fluid is modeled by a gas composed of elastic hard circles (spheres in three dimensions). The simulation steps the gas forward at discrete time intervals, allowing the gas particles to collide (with another gas particle or with a macroscopic grain) only once per step. The fluid velocity and the fluid force on each grain are computed by averaging. Since a typical void between macroscopic grains will be occupied by up to 1000 gas particles, the requisite computational speed and memory capacity can be found only in the hypercube architecture. Communication is expected to

be minimal and load balancing can be accomplished for a sufficiently large system. It is expected that larger scale simulations of erosion and deposition by water [Ahnert:87a] will benefit from the findings of the fluid/grain mixture simulations. Also, these large-scale landscape evolution simulations are, themselves, suitable for a hypercube.

Computer simulation is assuming an increasing role in geomorphology. We suggest that the development and availability of hypercube concurrent processors will have considerable influence upon the future of computing in geomorphology.

9.5 Plasma Particle-in-Cell Simulation of an Electron Beam Plasma Instability

9.5.1 Introduction

Plasmas—gases of electrically charged particles—are one of the most complex fluids encountered in nature. Because of the long range nature of the electric and magnetic interactions between the plasma electrons and ions composing them, plasmas exhibit a wide variety of collective forms of motions, e.g., coherent motions of large number of electrons, ions, or both. This leads to an extremely rich physics in plasmas. Plasma particle-in-cell (PIC) simulation codes have proven to be a powerful tool for the study of complex nonlinear plasma problems in many areas of plasma physics research such as space and astrophysical plasmas, magnetic and inertial confinement, free electron lasers, electron and ion beam propagation and particle accelerators. In PIC codes, the orbits of thousands to millions of interacting plasma electrons and ions are followed in the time as the particles move in electromagnetic fields calculated self-consistently from the charge and current densities created by these same plasma particles.

We have developed an algorithm, called the General Concurrent Particle-in-Cell Algorithm (GCPIC) for implementing PIC codes efficiently on MIMD parallel computers [Liewer:89c]. This algorithm was first used to implement a well-benchmarked [Decyk:88a] one-dimensional electrostatic PIC code. The benchmark problem, used to benchmark the Mark IIIfp, was a simulation of an electron beam plasma instability ([Decyk:88a], [Liewer:89c]). More recently, dynamic load balancing has been implemented in a one-dimensional electromagnetic GCPIC code [Liewer:90a] and a two-dimensional electrostatic code has been implemented using the GCPIC algorithm [Ferraro:90b].

9.5.2 GCPIC Algorithm

In plasma PIC codes, the orbits of the many interacting plasma electrons and ions are followed as an initial value problem as the particles move in self-consistently calculated electromagnetic fields. The fields are found by solving Maxwell's equations, or a subset, with the plasma currents and charge density as source terms; the electromagnetic fields determine the forces on the particles. In a PIC code, the particles can be anywhere in the simulation domain, but the field equations are solved on a discrete grid. At each time step in a PIC code, there are two stages to the computation. In the first stage, the position and velocities of the particles are updated by calculating the forces on the particles from interpolation of the field values at the grid points; the new charge and current densities at the grid points are then calculated by interpolation from the new positions and velocities of the particles. In the second stage, the updated fields are found by solving the field equations on the grid using the new charge and current densities. Generally, the first stage accounts for most of the computation time because there are many more particles than grid points.

The General Concurrent PIC (GCPIC) algorithm [Liewer:89c] is designed to make the most computationally intensive portion of a PIC code, updating the particles and the resulting charge and current densities, run efficiently on a parallel processor. The time used to make these updates is generally on the order of 90% of the total time for a sequential code, with the remaining time divided between the electromagnetic field solution and the diagnostic computations.

To implement a PIC code in parallel using the GCPIC algorithm, the physical domain of the particle simulation is partitioned into sub-domains, equal in number to the number of processors, such that all sub-domains have roughly equal numbers of particles. For problems with non-uniform particle densities, these sub-domains will be of unequal physical size. Each processor is assigned a sub-domain and is responsible for storing the particles and the electromagnetic field quantities for its sub-domain and for performing the particle computations for its particles. For a one-dimensional code on a hypercube, nearest neighbor sub-domains are assigned to nearest neighbor processors. When particles move to new sub-domains, they are passed to the appropriate processor. As long as the number of particles per sub-domain is approximately equal, the processors' computational loads will be balanced. Dynamic load balancing is accomplished by repartitioning the simulations domain into sub-domains with roughly equal particle numbers when the

processor loads become sufficiently unbalanced. The computation of the new partitions, done in a simple way using a crude approximation to the plasma density profile, adds very little overhead to the parallel code.

The decomposition used for dividing the particles is termed the *primary decomposition.* Because the primary decomposition is not generally the optimum one for the field solution on the grid, a *secondary decomposition* is used to divide the field computation. The secondary decomposition remains fixed. At each time step, grid data must be transferred between the two decompositions [Ferraro:90b], [Liewer:89c].

The GCPIC algorithm has led to a very efficient parallel implementation of the benchmarked one-dimensional electrostatic PIC code [Liewer:89c]. In this electrostatic code, only forces from self-consistent (and external) electric fields are included; neither an external nor a plasma-generated magnetic field is included.

9.5.3 Electron Beam Plasma Instability

The problem used to benchmark the one-dimensional electrostatic GCPIC code on the Mark IIIfp was a simulation of an instability in a plasma due to the presence of an electron beam. The six color pictures in Figure 9.1 show results from this simulation from the Mark IIIfp. Plotted is electron phase space—position versus velocity of the electrons—at six times during the simulation. The horizontal axis is the velocity and the vertical axis is the position of the electrons. Initially, the background plasma electrons (magenta dots) have a Gaussian distribution of velocities about zero. The width of the distribution in velocity is a measure of the temperature of the electrons. The beam electrons (yellow dots) streams through the background plasma at five times the thermal velocity. The beam density was 10% of the density of the background electrons. Initially, these have a Gaussian distribution about the beam velocity. Both beam and background electrons are distributed uniformly in x. This initial configuration is unstable to an electrostatic plasma wave which grows by tapping the free energy of the electron beam. At early times, the unstable waves grow exponentially. The influence of this electrostatic wave on the electron phase space is shown in the subsequent plots. The beam electrons lose energy to the wave. The wave acts to try to "thermalize" the electron's velocity distribution in the way collisions would act in a classical fluid. At some point, the amplitude of the wave's electrostatic potential is enough to "trap" some of the beam and background electrons, leading the visible swirls in the phase space plots. This trapping

		Mark III		Mark IIIfp	
Processors	Particles	Push Time μ secs	Push Efficiency	Push Time μ secs	Push Efficiency
1	11,264	241.5	...	49.0	...
2	22,528	121.2	100%	24.7	99%
4	45,056	60.6	100%	12.3	99%
8	90,112	30.5	100%	6.3	99%
16	180,224	15.6	97%	3.1	98%
32	360,448	7.8	97%	1.6	98%

Table 9.1: Hypercube Push Efficiency for Increasing Problem Size

causes the wave to stop growing. In the end, the beam and background electrons are mixed and the final distribution is "hotter" kinetic energy from the electron beam which has gone into heating both the background and beam electrons.

9.5.4 Performance Results for One-Dimensional Electrostatic Code

Timing results for the benchmark problem, using the one-dimensional code without dynamic load balancing, are given in the tables. In Table 9.1, results for the *push time* are given for various hypercube dimensions for the Mark III and Mark IIIfp hypercubes. Here, we define the push time as the time per particle per time step to update the particle positions and velocities (including the interpolation to find the forces at the particle positions) and to *deposit* (interpolate) the particles' contributions to the charge and/or current densities onto the grid. Table 9.1 shows the efficiency of the push for runs in which the number of particles *increased linearly* with the number of processors used, so that the number of particles per processor was constant (*fixed grain size*). The *efficiency* is defined to be $\epsilon = \frac{t(1)}{Nt(N)}$, where $t(N)$ is the run time on N processors. In the ideal situation, a code's run time on N will be $1/N$ of its run time on one processor, and the efficiency is 100%. In practice, communication between nodes and unequal processors loads leads to a decrease in the efficiency.

The Mark III Hypercube consists of up to 64 independent processors,

each with four megabytes of dynamic random access memory and 128 kilobytes of static RAM. Each processor consists of two Motorola MC68020 CPU's with a MC68882 Co-Processor. The newer Mark IIIfp Hypercubes have, in addition, a Weitek floating point processor on each node. In Table 9.1, push times are given for both the Mark III processor (Motorola MC68882) and the Mark IIIfp processor (Weitek). For the Weitek runs, the entire parallel code was downloaded into the Weitek processors. The push time for the one-dimensional electrostatic code has been benchmarked on many computers [Decyk:88a]. Some of the times are given in Table 9.2; times for other computers can be found in [Decyk:88a]. For the Mark III and Mark IIIfp runs, 720,896 particles were used (11,264 per processor); for the other runs in Table 9.2, 11,264 particles were used. In all cases, the push time is the time per particle per time step to make the particle updates. It can be seen that for the push portion of the code, the 64-processor Mark IIIfp is nearly twice the speed of a one-processor CRAY X-MP and 2.6 times the speed of a CRAY 2.

We have also compared the total run time for the benchmark code for a case with 720,896 particles and 1,024 grid points run for 1,000 time steps. The total run time on the 64-node Mark IIIfp was 1062 secs and the total run time on a one-processor CRAY 2 was 1714 secs. For this case, the 64-node Mark IIIfp was 1.6 times faster than the CRAY 2 for the entire code. For the Mark IIIfp run, about 10% of the total run time was spent in the initialization of the particles, which is done sequentially.

Benchmark times for the two-dimensional GCPIC code can be found in [Ferraro:90b].

9.5.5 One-Dimensional Electromagnetic Code

The parallel one-dimensional electrostatic code was modified to include the effects of external and self-consistent magnetic fields. This one-dimensional electromagnetic code, with kinetic electrons and ions, has been used to study electron dynamics in oblique collisionless shock waves such as in the earth's bow shock. Forces on the particles are found from the fields at the grid points by interpolation. For this code, with variation in the $x-$direction only, the orbit equations for the ith particle are

$$\begin{aligned} \frac{d\,x_i}{dt} &= v_{x,i} \\ \frac{d\,\mathbf{v}_i}{dt} &= \frac{q_i}{m_i}\left(\mathbf{E} + \frac{\mathbf{v}_i \times \mathbf{B}}{c}\right). \end{aligned} \tag{9.1}$$

Computer	Push Time μ secs
Mark IIIfp (64 processor)	0.8
CRAY X-MP/48 (1 processor)	
Vectorized	1.5
Scalar	4.1
CRAY 2 (1 processor)	
Vectorized	2.1
Scalar	10.1
IBM 3090 VF	
Vectorized	2.9
Scalar	6.0
Mark III (64 processor)	3.9
Alliant FX/8	12.6
VAX 11/750, F.P.A.	200.9

Table 9.2: Comparison of Push Times on Various Computers

Motion is followed in the x direction only, but all three velocity components must be calculated in order to calculate the $\mathbf{v} \times \mathbf{b}$ force. The longitudinal (along x) electric field is found by solving Poisson's equation

$$\nabla \cdot \mathbf{E} = 4\pi\rho_q(x, t). \tag{9.2}$$

The transverse (to x) electromagnetic fields, E_y, E_z, B_y, and B_z, are found by solving

$$\begin{aligned} \frac{\partial \mathbf{b}}{\partial t} &= -\frac{1}{c}\nabla \times \mathbf{E} \\ \frac{\partial \mathbf{E}}{\partial t} &= c\nabla \times \mathbf{B} - 4\pi\mathbf{j}. \end{aligned} \tag{9.3}$$

The plasma current density $\mathbf{j}(x, t)$ and charge density $\rho_q(x, t)$ are found at the grid points by interpolation from the particle positions. Only the transverse (y and z) components of the plasma current are needed. These coupled particle and field equations are solved in time as an initial value problem. As in the electrostatic code, the fields are solved by Fourier transforming the charge and current densities and solving the equation in k space, and advancing the Fourier components in time. External fields and currents can also be included. At each time step, the fields are transformed back to configuration space to calculate the forces needed to advance the particles to the next time step. The hypercube FFT routine, described in Section 12.4 was used in the one-dimensional codes. Extending the existing parallel electrostatic code to include the electromagnetic effects required no change in the parallel decomposition of the code.

9.5.6 Dynamic Load Balancing

In the GCPIC electrostatic code, the partitioning of the grid was static. The grid was partitioned so that the computational load of the processors was initially balanced. As simulations progress and particles move among processors, the spatial distribution of the particles can change, leading to load imbalance. This can severely degrade the parallel efficiency of the push stage of the computation. To avoid this, dynamic load balancing has been implemented in the one-dimensional electromagnetic code [Liewer:90a].

To implement dynamic load balancing, the grid is repartitioned into new sub-domains with roughly equal numbers of particles as the simulation progresses. The repartitioning is not done at every time step. The load imbalance is monitored at a user-specified interval. When the imbalance becomes

sufficiently large, the grid is repartitioned and the particles moved to the appropriate processors, as necessary. The load was judged sufficiently imbalanced to warrant load balancing when the number of particles per processor deviated from the ideal value n_{ideal} (= number of particles/number of processors) by $2\sqrt{n_{ideal}}$, *e.g.*, twice the statistical fluctuation level.

The dynamic load balancing is performed during the push stage of the computation. Specifically, the new grid partitions are computed after the particle positions have been updated, but before the particles are moved to new processors to avoid an unnecessary moving of particles. If the loads are sufficiently balanced, the subroutine computing the new grid partitions is not called. The subroutine, which moves the particles to appropriate processors, is called in either case.

To accurately represent the physics, a particle can not move more than one grid cell per time step. As a result, in the static one-dimensional code, the routine which moves particles to new processors only had to move particles to nearest neighbor processors. To implement dynamic load balancing, this subroutine had to be modified to allow particles to be moved to processors any number of steps away. Moving the particles to new processors after grid repartitioning can add significant overhead; however, this is incurred only at time steps when load balancing occurs.

The new grid partitions are computed by a very simple method which adds very little overhead to the parallel code. Each processor constructs an approximation to the plasma density profile, $\bar{n}(x)$, and uses this to compute the grid partitioning to load balance. To construct the approximate density profile, each processor sends the locations of its current sub-domain boundaries and its current number of particles to all other processors. From this information, each processor can compute the average plasma density in each processor and from this can create the approximate to density profile (with as many points as processors). This approximate profile is used to compute the grid partitioning which approximately divides the particles equally among the processors. This is done by determining the set of sub-domain boundaries x_{left} and x_{right} such that

$$n_{ideal} \approx \int_{x_{left}}^{x_{right}} \bar{n}(x)\, dx. \tag{9.4}$$

Linear interpolation of the approximate profile is used in the numerical integration. The actual plasma density profile could also be used in the integration to determine the partitions. No additional computation would be necessary to obtain the local (within a processor) $n(x_{grid})$ because it is

already computed for the field solution stage, but it would require more communication to make the density profile global. Other methods of calculating new sub-domain boundaries, such as sorting particles, require a much large amount of communication and computational overhead.

9.5.7 Credits and References

The GCPIC algorithm was developed and implemented by Paulett C. Liewer, Jet Propulsion Propulsion Laboratories, Caltech and Viktor K. Decyk, Physics Department, University of California, Los Angeles. E. W. Leaver, Jet Propulsion Laboratory, Caltech contributed to implementation of dynamic load balancing.

The research was sponsored by Sandia National Laboratory, Albuquerque, DOE, and Caltech President's Fund Grant No. PF-317.

9.6 LU Factorization of Sparse, Unsymmetric Jacobian Matrices on Multicomputers: Experience, Strategies, Performance

Efficient sparse linear algebra *cannot* be achieved as a straightforward extension of the dense case, even for concurrent implementations. This paper details a new, general-purpose unsymmetric sparse LU factorization code built on the philosophy of Harwell's *MA28*, with variations. We apply this code in the framework of Jacobian-matrix factorizations, arising from Newton iterations in the solution of nonlinear systems of equations. Serious attention has been paid to the data-structure requirements, complexity issues and communication features of the algorithm. Key results include reduced communication pivoting for both the "analyze" A-mode and repeated B-mode factorizations, and effective general-purpose data distributions useful incrementally to trade-off process-column load balance in factorization against triangular solve performance. Future planned efforts are cited in conclusion.

9.6.1 Introduction

The topic of this paper is the implementation and concurrent performance of sparse, unsymmetric LU factorization for medium-grain multicomputers. Our target hardware is distributed-memory, message-passing concurrent computers such as the Symult s2010 and Intel iPSC/2 systems. For both of

these systems, efficient cut-through *wormhole* routing technology provides pair-wise communication performance essentially independent of the spatial location of the computers in the ensemble [Athas:88a]. The Symult s2010 is a two-dimensional, mesh-connected concurrent computer; all examples in this paper were run on this variety of hardware. Message-passing performance, portability and related issues relevant to this work are detailed in [Skjellum:90a].

Questions of linear-algebra performance are pervasive throughout scientific and engineering computation. The need for high-quality, high-performance linear algebra algorithms (and libraries) for multicomputer systems therefore requires no attempt at justification. The motivation for the work described here has a specific origin, however. Our main higher-level research goal is the concurrent dynamic simulation of systems modelled by ordinary differential and algebraic equations; specifically, dynamic flowsheet simulation of chemical plants (*e.g.*, coupled distillation columns) [Skjellum:90c]. Efficient sequential integration algorithms solve staticized nonlinear equations at each time point via modified Newton iteration (*cf.*, [Brenan:89a], Chapter 5). Consequently, a sequence of structurally identical linear systems must be solved; the matrices are finite-difference approximations to Jacobians of the staticized system of ordinary differential-algebraic equations. These Jacobians are large, sparse and unsymmetric for our application area. In general, they possess both band and significant off-band structure. Generic structures are depicted in Figure 9.2. This work should also bear relevance to electric power network/grid dynamic simulation where sparse, unsymmetric Jacobians also arise, and also elsewhere.

9.6.2 Design Overview

We solve the problem $Ax = b$ where A is large, and includes many zero entries. We assume that A is unsymmetric both in sparsity pattern and in numerical values. In general, the matrix A will be computed in a distributed fashion, so we will inherit a distribution of the coefficients of A (*cf.*, Figure 9.3, Figure 9.4). Following the style of Harwell's *MA28* code for unsymmetric sparse matrices, we use a two-phase approach to this solution. There is a first LU factorization called A-mode or "analyze," which builds data structures dynamically, and uses a user-defined pivoting function. The repeated B-mode factorization uses the existing data structures statically to factor a new, similarly structured matrix, with the previous pivoting pattern. B-mode monitors stability with a simple growth factor estimate. In

Figure 9.1: Time history of electron phase space in a plasma PIC simulation of an electron beam plasma instability on the Mark III hypercube. The horizontal axis is the electron velocity and the vertical axis is the position. Initially, a small population of beam electrons (green dots) stream through the background plasma electrons (magenta dots). An electrostatic wave grows, tapping the energy in the electron beam. The vortices in phase space at late times result from electrons becoming trapped in the potential of the wave. See Section 9.5 for further description.

Figure 9.2: Example Jacobian Matrix Structures. In chemical-engineering process flowsheets, Jacobians with main band structure, and lower-triangular structure (feedforwards), upper-triangular structure (feedbacks), and borders (global or artificially restructured feedforwards and/or feedbacks) are common.

Figure 9.3: Process Grid Data Distribution of $Ax = b$. Representation of a concurrent matrix, and distributed-replicated concurrent vectors on a 4×4 logical process grid. The solution of $Ax = b$ first appears in x, a column-distributed vector, and then is normally "transposed" via a global *combine* to the row-distributed vector y.

practice, A-mode is repeated whenever instability is detected. The two key contributions of this sparse concurrent solver are: reduced communication pivoting, and new data distributions for better overall performance.

Following Van de Velde [Velde:90a], we consider the LU factorization of a real matrix A, $A \in \Re^{N \times N}$. It is well known (*e.g.*, [Golub:89a], pp. 117-118), that for any such matrix A, an LU factorization of the form

$$\mathsf{P}_R A \mathsf{P}_C^T = \hat{L}\hat{U}$$

exists, where $\mathsf{P}_R, \mathsf{P}_C$ are square, (orthogonal) permutation matrices, and $\hat{L}, \hat{U}$ are the unit lower-triangular, and upper-triangular factors, respectively. Whereas the pivot sequence is stored (two N-length integer vectors), the permutation matrices are not stored or computed with explicitly. Rearranging, based on the orthogonality of the permutation matrices, $A = \mathsf{P}_R^T \hat{L}\hat{U}\mathsf{P}_C$. We factor A with implicit pivoting (no rows or columns are exchanged explicitly as a result of pivoting). Therefore, we do not store $\hat{L}, \hat{U}$ directly, but instead: $L = \mathsf{P}_R^T \hat{L}\mathsf{P}_C$, $U = \mathsf{P}_R^T \hat{U}\mathsf{P}_C$. Consequently, $\hat{L}\mathsf{P}_R L \mathsf{P}_C^T$, $\hat{U} = \mathsf{P}_R U \mathsf{P}_C^T$, and $A = L(\mathsf{P}_C^T \mathsf{P}_R)U$. The "unravelling" of the permutation matrices is accomplished readily (without implication of additional interprocess communication) during the triangular solves.

For the sparse case, performance is more difficult to quantify than for the dense case, but, for example, banded matrices with bandwidth β can

$$\begin{pmatrix} A^{0,0} & A^{0,1} & A^{0,2} & A^{0,3} \\ A^{1,0} & A^{1,1} & A^{1,2} & A^{1,3} \\ A^{2,0} & A^{2,1} & A^{2,2} & A^{2,3} \\ A^{3,0} & A^{3,1} & A^{3,2} & A^{3,3} \end{pmatrix}_{\mathcal{G}} = \left(\begin{array}{cc|cc|cc|ccc} a_{0,1} & a_{0,5} & a_{0,2} & a_{0,6} & a_{0,3} & a_{0,7} & a_{0,0} & a_{0,4} & a_{0,8} \\ a_{1,1} & a_{1,5} & a_{1,2} & a_{1,6} & a_{1,3} & a_{1,7} & a_{1,0} & a_{1,4} & a_{1,8} \\ \hline a_{2,1} & a_{2,5} & a_{2,2} & a_{2,6} & a_{2,3} & a_{2,7} & a_{2,0} & a_{2,4} & a_{2,8} \\ a_{3,1} & a_{3,5} & a_{3,2} & a_{3,6} & a_{3,3} & a_{3,7} & a_{3,0} & a_{3,4} & a_{3,8} \\ \hline a_{4,1} & a_{4,5} & a_{4,2} & a_{4,6} & a_{4,3} & a_{4,7} & a_{4,0} & a_{4,4} & a_{4,8} \\ a_{5,1} & a_{5,5} & a_{5,2} & a_{5,6} & a_{5,3} & a_{5,7} & a_{5,0} & a_{5,4} & a_{5,8} \\ a_{6,1} & a_{6,5} & a_{6,2} & a_{6,6} & a_{6,3} & a_{6,7} & a_{6,0} & a_{6,4} & a_{6,8} \\ a_{7,1} & a_{7,5} & a_{7,2} & a_{7,6} & a_{7,3} & a_{7,7} & a_{7,0} & a_{7,4} & a_{7,8} \\ \hline a_{8,1} & a_{8,5} & a_{8,2} & a_{8,6} & a_{8,3} & a_{8,7} & a_{8,0} & a_{8,4} & a_{8,8} \\ a_{9,1} & a_{9,5} & a_{9,2} & a_{9,6} & a_{9,3} & a_{9,7} & a_{9,0} & a_{9,4} & a_{9,8} \\ a_{10,1} & a_{10,5} & a_{10,2} & a_{10,6} & a_{10,3} & a_{10,7} & a_{10,0} & a_{10,4} & a_{10,8} \end{array}\right)$$

Figure 9.4: Example of Process-Grid Data Distribution. An 11×9 array with block-linear rows ($B = 2$) and scattered columns on a 4×4 logical process grid. Local arrays are denoted at left by $A^{p,q}$ where (p,q) is the grid position of the process on $\mathcal{G} \equiv \left(\left\{(\lambda_2, \lambda_2^{-1}, \lambda_2^{\sharp}); P = 4, M = 11\right\}, \left\{(\sigma_1, \sigma_1^{-1}, \sigma_1^{\sharp}); Q = 4, N = 9\right\}\right)$. Subscripts (*i.e.*, $a_{I,J}$) are the global (I,J) indices.

Figure 9.5: Linked-list Entry Structure of Sparse Matrix. A single entry consists of a double-precision value (8 bytes), the local row (i) and column (j) index (2 bytes each), a "Next Column Pointer" indicating the next current column entry (fixed j), and a "Next Row Pointer" indicating the next current row entry (fixed i), at 4 bytes each. Total: 24 bytes per entry.

be factored with $O(\beta^2 N)$ work; we expect sub-cubic complexity in N for reasonably sparse matrices, and strive for sub-quadratic complexity, for very sparse matrices. The triangular solves can be accomplished in work proportional to the number of entries in the respective triangular matrix L or U. The pivoting strategy is treated as a parameter of the algorithm and is not pre-determined. We can consequently treat the pivoting function as an application-dependent function, and sometimes tailor it to special problem structures (*cf.*, Section 7 of [Velde:88a]) for higher performance. As for all sparse solvers, we also seek sub-quadratic memory requirements in N, attained by storing matrix entries in linked-list fashion, as illustrated in Figure 9.5.

For further discussion of LU factorizations and sparse matrices, see [Golub:89a], and [Duff:86a].

9.6.3 Reduced-Communication Pivoting

At each stage of the concurrent LU factorization, the pivot element is chosen by the user-defined pivot function. Then, the pivot row (new row of U) must be broadcast, and pivot column (new column of L) must be computed and broadcast on the logical process grid (*cf.*, Figure 9.3, vertically and horizontally, respectively. Note that these are interchangeable operations. We use this degree-of-freedom to reduce the communication complexity of particular pivoting strategies, while impacting the effort of the LU factorization itself negligibly.

We define two "correctness modes" of pivoting functions. In the first correctness mode "first row fanout," the exit conditions for the pivot function are: all processes must know $\hat{p}$ (the pivot process row), the pivot process row must know $\hat{q}$ (the pivot process column) as well as $\hat{\imath}$, the $\hat{p}$-local matrix row of the pivot, and the pivot process must know in addition the pivot value and $\hat{q}$-local matrix column $\hat{\jmath}$ of the pivot. Partial column pivoting and preset pivoting can be setup to satisfy these correctness conditions as follows. For partial column pivoting, the kth row is eliminated at the kth step of the factorization. From this fact, each process can derive the process row $\hat{p}$ and $\hat{p}$-local matrix row $\hat{\imath}$ using the row data distribution function. Having identified themselves, the pivot-row processes can look for the largest element in local matrix row $\hat{\imath}$ and choose the pivot element globally among themselves via a *combine*. At completion this places $\hat{q}$, $\hat{\jmath}$ and the pivot value in the entire pivot process row. This completes the requirements for the "first row fanout" correctness mode. For preset pivoting, the kth elimination row and column are both stored as $\hat{p}, \hat{\imath}, \hat{q}, \hat{\jmath}$, and each process knows these values *without communication.*[1] Furthermore, the pivot process looks up the pivot value. Hence, preset pivoting satisfies the requirements of this correctness mode also.

For "first row fanout," the universal knowledge of $\hat{p}$ and knowledge of the pivot matrix row $\hat{\imath}$ by the pivot process row, allows the vertical broadcast of this row (new row of U). In addition, we broadcast $\hat{q}$, $\hat{\jmath}$ and the pivot value simultaneously. This extends the correct value of $\hat{q}$ to all processes, as well as $\hat{\jmath}$ and the pivot value to the pivot process column. Hence, the multiplier (L) column may be correctly computed and broadcast. Along with the multiplier column broadcast, we include the pivot value. After this broadcast, all processes have the correct indices $\hat{p}, \hat{\imath}, \hat{q}, \hat{\jmath}$ and the pivot value. This provides all that's required to complete the current elimination step.

[1] Memory unscalabilities can be removed very cheaply; see [Skjellum:90c].

For the second correctness mode "first column fanout," the exit conditions for the pivot function are: all processes must know $\hat{q}$, the entire pivot process column must know $\hat{\jmath}$, the pivot value, and $\hat{p}$. The pivot process in addition knows $\hat{\imath}$. Partial row pivoting can be setup to satisfy these correctness conditions. The arguments are analogous to partial column pivoting and are given in [Skjellum:90c].

For "first column fanout," the entire pivot process column knows the pivot value, and local column of the pivot. Hence, the multiplier column may be computed by dividing the pivot matrix column by the pivot value. This column of L may then be broadcast horizontally, including the pivot value, $\hat{p}$ and $\hat{\imath}$ as additional information. After this step, the entire ensemble has the correct pivot value, and $\hat{p}$; in addition, the pivot process row has the correct $\hat{\imath}$. Hence, the pivot matrix row may be identified and broadcast. This second broadcast completes the needed information in each process for effecting the kth elimination step.

Hence, when using partial row or partial column pivoting, only local combines of the pivot process column (respectively row) are needed. The other processes don't participate in the combine, as they must without this methodology. Preset pivoting implies no pivoting communication, except very occasionally (*e.g.*, 1 in 5000 times) as noted in [Skjellum:90c] to remove memory unscalabilities. This pivoting approach is a direct savings, gained at a negligible additional broadcast overhead. See also [Skjellum:90c].

9.6.4 New Data Distributions

We introduce new closed-form $O(1)$-time, $O(1)$-memory data distributions useful for sparse matrix factorizations and the problems that generate such matrices. We quantify evaluation costs in Table 9.3 .

Every concurrent data structure is associated with a logical process grid at creation (*cf.*, Figure 9.3 and [Skjellum:90a], and [Skjellum:90c]). Vectors are either row- or column-distributed within a two-dimensional process grid. Row-distributed vectors are *replicated* in each process column, and distributed in the process rows. Conversely, column-distributed vectors are replicated in each process row, and distributed in the process columns. Matrices are distributed both in rows and columns, so that a single process owns a subset of matrix rows and columns. This partitioning follows the ideas proposed by Fox *et al.* [Fox:88a] and others. Within the process grid, coefficients of vecto and matrices are distributed according to one of several data distributions. Data distributions are chosen to compromise

Distribution:	$\mu(I,P,M)$	$\mu^{-1}(p,i,P,M)$
One-Parameter (ζ)	$5.5554 \times 10^1 \pm 5 \times 10^{-3}$	$4.0024 \times 10^1 \pm 7 \times 10^{-3}$
Two-Parameter (ξ)	$6.1710 \times 10^1 \pm 1 \times 10^{-2}$	$4.2370 \times 10^1 \pm 8 \times 10^{-3}$
Block-Linear (λ)	$5.4254 \times 10^1 \pm 7 \times 10^{-3}$	$3.5404 \times 10^1 \pm 5 \times 10^{-3}$

Table 9.3: For the data distributions and inverses described here, evaluation time in μs is quoted for the Symult s2010 multicomputer. Cardinality function calls are inexpensive, and fall within lower-order work anyway—their timing is hence omitted. The cheapest distribution function (scatter) costs $\approx 15\mu$s by way of comparison.

between load-balancing requirements and constraints on where information can be calculated in the ensemble.

Definition 1 (Data-Distribution Function)

A data-distribution function μ maps three integers $\mu(I,P,M) \mapsto (p,i)$ where I, $0 \leq I < M$, is the global name of a coefficient, P is the number of processes among which all coefficients are to be partitioned, and M is the total number of coefficients. The pair (p,i) represents the process p ($0 \leq p < P$) and local (process-p) name i of the coefficient ($0 \leq i < \mu^\sharp(p,P,M)$). The inverse distribution function $\mu^{-1}(p,i,P,M) \mapsto I$ transforms the local name i back to the global coefficient name I.

The formal requirements for a data distribution function are as follows. Let $\mathcal{I}^p$ be the set of global coefficient names associated with process p, $0 \leq p < P$, defined implicitly by a data distribution function $\mu(\bullet,P,M)$. The following set properties must hold:

$$\begin{aligned} \mathcal{I}^{p_1} \cap \mathcal{I}^{p_2} &= \emptyset, \ \forall\, p_1 \neq p_2, \quad 0 \leq p_1, p_2 < P \\ \bigcup_{p=0}^{P-1} \mathcal{I}^p &= \{0,\ldots,M-1\} \equiv \mathcal{I}_M \end{aligned}$$

The cardinality of the set $\mathcal{I}^p$, is given by $\mu^\sharp(p,P,M)$.

The linear and scatter data-distribution functions are most often defined. We generalize these functions (by blocking and scattering parameters) to incorporate practically important degrees of freedom. These generalized distribution functions yield optimal static load balance as do the unmodified

functions described in [Velde:90a] for unit block size, but differ in coefficient placement. This distinction is technical, but necessary for efficient implementations.

Definition 2 (Generalized Block-Linear)

The definitions for the generalized block-linear distribution function, inverse, and cardinality function are:

$$\begin{aligned}
\lambda_B(I,P,M) &\mapsto (p,i),\\
p &\equiv P-1-\\
&\qquad \max\left(\left\lfloor\frac{I_B^{\text{rev}}}{l+1}\right\rfloor,\left\lfloor\frac{I_B^{\text{rev}}-r}{l}\right\rfloor\right),\\
i &\equiv I-B\left(pl+\Theta^1\left(p-(P-r)\right)\right),
\end{aligned}$$

while

$$\begin{aligned}
\lambda_B^{-1}(p,i,P,M) &\equiv i+B\left(\left(pl+\Theta^1\left(p-(P-r)\right)\right),\right.\\
\lambda_B^{\sharp}(p,P,M) &\equiv B\left(\left\lfloor\frac{b+p}{P}\right\rfloor-\theta\right)+\\
&\qquad (M \bmod B)\theta,
\end{aligned}$$

where B denotes the coefficient block size,

$$\begin{aligned}
b &= \begin{cases} \frac{M}{B} & \text{if } M \bmod B = 0\\ \left\lfloor\frac{M}{B}\right\rfloor+1 & \text{otherwise,}\end{cases}\\
I_B &= \left\lfloor\frac{I}{B}\right\rfloor,\quad I_B^{\text{rev}} = b-1-I_B,\\
l &= \left\lfloor\frac{b}{P}\right\rfloor,\quad r = b \bmod P\ ,\\
\Theta^k(t) &\equiv \begin{cases} 0 & t\le 0\\ t^k & t>0,\ k>0\\ 1 & t>0,\ k=0\end{cases},\\
\theta &= \left\lfloor\frac{p+1}{P}\right\rfloor\Theta^0(M \bmod B)
\end{aligned}$$

and where $b \geq P$.

For $B = 1$, *a load-balance-equivalent variant of the common linear data-distribution function is recovered. The general block-linear distribution function divides coefficients among the* P *processes* $p = 0, \ldots, P-1$ *so that each* $\mathcal{I}^p$ *is a set of coefficients with contiguous global names, while optimally load-balancing the* b *blocks among the* P *sets. Coefficient boundaries between processes are on multiples of* B. *The maximum possible coefficient imbalance between processes is* B. *If* $B \bmod P \neq 0$, *the last block in process* $P-1$ *will be foreshortened.*

Definition 3 (Parametric Functions)

To allow greater freedom in the distribution of coefficients among processes, we define a new, two-parameter distribution function family, ξ. *The* B *blocking parameter (just introduced in the block-linear function) is mainly suited to the clustering of coefficients that must not be separated by an interprocess boundary (again, see [Skjellum:90c] for a definition of general block-scatter,* σ*). Increasing* B *worsens the static load balance. Adding a second scaling parameter* S *(of no impact on the static load balance) allows the distribution to scatter coefficients to a greater or lesser degree, directly as a function of this one parameter. The two-parameter distribution function, inverse and cardinality function are defined below. The one-parameter distribution function family,* ζ, *occurs as the special case* $B = 1$, *also as noted below:*

$$\xi_{B,S}(I,P,M) \;\mapsto\; (p,i) \;\equiv\; \begin{cases} (p_0,i_0) & \Lambda_0 \geq l_S \\ (p_1,i_1) & \Lambda_0 < l_S \end{cases}$$

where

$$\begin{aligned} l_S &\equiv \left\lfloor \frac{l}{S} \right\rfloor, \quad \Lambda_0 \equiv \left\lfloor \frac{i_0}{BS} \right\rfloor, \\ (p_0,i_0) &\leftarrow \lambda_B(I,P,M), \\ I_{BS} &= p_0 l_S + \Lambda_0, \\ p_1 &\equiv I_{BS} \bmod P\ , \\ i_1 &\equiv BS \left\lfloor \frac{I_{BS}}{P} \right\rfloor + (i_0 \bmod BS), \end{aligned}$$

with

$$\zeta_S(I,P,M) \;\equiv\; \xi_{1,S}(I,P,M),$$

$$\xi^{\sharp}_{B,S}(p,P,M) \equiv \lambda^{\sharp}_{B}(p,P,M),$$
$$\zeta^{\sharp}_{S}(p,P,M) \equiv \lambda^{\sharp}_{1}(p,P,M),$$

and where r, b, etc. *are as defined above. The inverse distribution function* ξ^{-1} *is defined as follows:*

$$\xi^{-1}_{B,S}(p,i,P,M) \mapsto I = \lambda^{-1}_{B}(p^*,i^*,P,M),$$
$$(p^*,i^*) \equiv \begin{cases} (p,i) & \Lambda \geq l_S \\ (p_2,i_2) & \Lambda < l_S \end{cases},$$
$$\Lambda \equiv \left\lfloor \frac{i}{BS} \right\rfloor, \quad I^*_{BS} = p + \Lambda P,$$
$$p_2 \equiv \left\lfloor \frac{I^*_{BS}}{l_S} \right\rfloor,$$
$$i_2 \equiv BS\,(I^*_{BS} \bmod l_S) + (i \bmod BS),$$

with

$$\zeta^{-1}_{S}(p,i,P,M) \equiv \xi^{-1}_{1,S}(p,i,P,M).$$

For $S = 1$, *a block-scatter distribution results, while for* $S \geq S_{crit} \equiv \lfloor l/2 \rfloor + 1$, *the generalized block-linear distribution function is recovered. See also [Skjellum:90c].*

Definition 4 (Data Distributions)

Given a data-distribution function family $(\mu, \mu^{-1}, \mu^{\sharp})$ $((\nu, \nu^{-1}, \nu^{\sharp}))$, *a process list of* P (Q), M (N) *as the number of coefficients, and a row (respectively, column) orientation, a row (column) data distribution* $\mathcal{G}^{row}$ $(\mathcal{G}^{col})$ *is defined as:*

$$\mathcal{G}^{row} \equiv \left\{(\mu, \mu^{-1}, \mu^{\sharp}); P, M\right\},$$

respectively,

$$\mathcal{G}^{col} \equiv \left\{(\nu, \nu^{-1}, \nu^{\sharp}); Q, N\right\}.$$

A two-dimensional data distribution may be identified as consisting of a row and column distribution defined over a two-dimensional process grid of $P \times Q$ *processes, as* $\mathcal{G} \equiv (\mathcal{G}^{row}, \mathcal{G}^{col})$.

Further discussion and detailed comparisons on data-distribution functions are offered in [Skjellum:90c]. Figure 9.4 illustrates the effects of linear and scatter data-distribution functions on a small rectangular array of coefficients.

9.6.5 Performance vs. Scattering

Consider a fixed logical process grid of R processes, with $P \times Q = R$. For the sake of argument, assume partial row pivoting during LU factorization for the retention of numerical stability. Then, for the LU factorization, it is well known that a scatter distribution is "good" for the matrix rows, and optimal were there no off-diagonal pivots chosen. Furthermore, the optimal column distribution is also scatter, because columns are chosen in order for partial row pivoting. Compatibly, a scatter distribution of matrix rows is also "good" for the triangular solves. However, for triangular solves, the best column distribution is linear, because this implies less intercolumn communication, as we detail below. In short, the optimal configurations conflict, and because explicit redistribution is expensive, a static compromise must be chosen. We address this need to compromise through the one-parameter distribution function ζ described in the previous section, offering a variable degree of scattering via the S-parameter. To first order, changing S does not affect the cost of computing the Jacobian (assuming columnwise finite-difference computation), because each process column works independently.

It's important to note that triangular solves derive no benefit from $Q > 1$. The standard column-oriented solve keep one process column active at any given time. For any column distribution, the updated right-hand-side vectors are retransmitted W times (process column-to-process column) during the triangular solve—whenever the active process column changes. There are at least $W_{min} \equiv Q - 1$ such transmissions (linear distribution), and at most $W_{max} \equiv N - 1$ transmissions (scatter distribution). The complexity of this retransmission is $O(WN/P)$, representing quadratic work in N for $W \sim N$.

Calculation complexity for a sparse triangular solve is proportional to the number of elements in the triangular matrix, with a low leading coefficient. Often, there are $O(N^{1.x})$ with $x < 1$ elements in the triangular matrices, including fill. This operation is then $O(N^{1.x}/P)$, which is less than quadratic in N. Consequently, for large W, the retransmission step is likely of greater cost than the original calculation. This retransmission effect constrains the amount of scattering and size of Q in order to have any chance of concurrent speedup in the triangular solves.

Using the one-parameter distribution with $S \geq 1$ implies that $W \approx N/S$, so that the retransmission complexity is $O(N^2/SP)$. Consequently, we can bound the amount of retransmission work by picking S sufficiently large. Clearly, $S = S_{crit}$ is a hard upper bound, because we reach the linear distribution limit at that value of the parameter. We suggest picking

$S \approx 10$ as a first guess, and $S \sim \sqrt{N}$, more optimistically. The former choice basically reduces retransmission effort by an order of magnitude. Both examples in the following section illustrate the effectiveness of choosing S by these heuristics.

The two-parameter ξ distribution can be used on the matrix rows to tradeoff load balance in the factorizations and triangular solves against the amount of (communication) effort needed to compute the Jacobian. In particular, a greater degree of scattering can dramatically increase the time required for a Jacobian computation (depending heavily on the underlying equation structure and problem), but importantly reduce load imbalance during the linear algebra steps. The communication overhead caused by multiple process rows suggests shifting toward smaller P and larger Q (a squatter grid), in which case greater concurrency is attained in the Jacobian computation, and the additional communication previously induced is then somewhat mitigated. The one-parameter distribution used on the matrix columns then proves effective in controlling the cost of the triangular solves by choosing the minimally allowable amount of column scattering.

Let's make explicit the performance objectives we consider when tuning S, and, more generally, when tuning the grid shape $P \times Q = R$. In the modified Newton iteration, for instance, a Jacobian factorization is reused until convergence slows unacceptably. An "LU Factorization + Backsolve" step is followed by η "Forward + Backsolves," with $\eta \sim O(1)$ typically (and varying dynamically throughout the calculation). Assuming an averaged η, say η^* (perhaps as large as five [Brenan:89a]), then our first-level performance goal is a heuristic minimization of

$$T_{LU} + (\eta^* + 1)T_{Back} + \eta^* T_{Forward}$$

over S for fixed P, Q. $\eta^* > 1$ more heavily weights the reduction of triangular solve costs *vs.* B-mode factorization than we might at first have assumed, placing a greater potential gain on the one-parameter distribution for higher overall performance. We generally want heuristically to optimize

$$T_{Jac} + T_{LU} + (\eta^* + 1)T_{Back} + \eta^* T_{Forward}$$

over S, P, Q, R. Then, the possibility of fine-tuning row and column distributions is important, as is the use of non-power-of-two grid shapes.

Distribution: Row	Column	(time in seconds) A-Mode	B-Mode	Back-Solve	Solve
Scatter	S=1	1.140×10^3	1.603×10^2	1.196×10^2	2.426×10^2
	S=10	1.148×10^3	1.696×10^2	3.294×10^1	6.912×10^1
	S=25	1.091×10^3	1.670×10^2	2.713×10^1	5.752×10^1
	S=30	1.095×10^3	1.769×10^2	2.653×10^1	5.631×10^1
	S=40	1.116×10^3	2.157×10^2	2.573×10^1	5.472×10^1
	S=50	1.127×10^3	2.157×10^2	2.764×10^1	5.743×10^1
	S=100	1.279×10^3	4.764×10^2	2.520×10^1	5.367×10^1
	Linear	2.247×10^3	1.161×10^3	2.333×10^1	4.993×10^1

Table 9.4: Order 13040 Band Matrix Performance. The above timing data, for the 16 × 12 grid configuration with scattered rows, indicates the importance of the one-parameter distribution with $S > 1$ for balancing factorization cost *vs.* triangular-solve cost. The random matrices, of order 13040, have an upper bandwidth of 164 and a lower bandwidth of 162. "Best"performance occurs in the range $S \approx 25 \ldots 40$.

9.6.6 Performance

Order 13040 Example

We consider an order 13040 banded matrix with a bandwidth of 326 under partial row pivoting. For this example, we have compiled timing results for a 16 × 12 process grid with random matrices (entries have range 0–10,000) using different values of S on the column distribution (see Table 9.4). We indicate timing for A-mode, B-mode, Backsolves and Forward- and Backsolves together ("Solve" heading). For this example, $S = 30$ saves 76% of the triangular solve cost compared to $S = 1$, or approximately 186 seconds, roughly six seconds above the linear optimal. Simultaneously, we incur about 17 seconds additional cost in B-mode, while saving about 93 seconds in the Backsolve. Assuming $\eta^* = 1$ ($\eta^* = 0$), in the first above-mentioned objective function, we save about 262 (respectively, 76) seconds. Based on this example, and other experiences, we conclude that this is a successful practical technique for improving overall sparse linear algebra performance. The following example further bolsters this conclusion.

Order 2500 Example

Now, we turn to a timing example of an order 2500 sparse, random matrix. The matrix has a random diagonal, plus two-percent random fill of the off-diagonals; entries have a dynamic range of 0–10,000. Normally, data is averaged over random matrices for each grid shape (as noted), and over four repetitive runs for each random matrix. Partial row pivoting was used exclusively. Table 9.5 compiles timings for various grid shapes of row-scatter/column-scatter, and row-scatter / column-($S = 10$) distributions, for as few as nine nodes and as many as 128. Memory limitations set the lower bound on the number of nodes.

This example demonstrates that speedups are possible for this reasonably small sparse example with this general-purpose solver, and that the one-parameter distribution is key to achieving overall better performance even for this random, essentially unstructured example. Without the one-parameter distribution, triangular solver performance is poor, except in grid configurations where the factorization is itself degraded (*e.g.*, 2×16). Furthermore, the choice of $S = 10$ is universally reasonable for the $Q > 1$ grid shapes illustrated here, so the distribution proves easy to tune for this type of matrix. We are able to maintain an almost constant speed for the triangular solves while increasing speed for both the A-mode and B-mode factorizations. We presume, based on experience, that triangular solve times are comparable to the sequential solution times—further study is needed in this area to see if and how performance can be improved. The consistent A-mode to B-mode ratio of approximately two is attributed primarily to reduced communication costs in B-mode, realized through the elimination of essentially all *combine* operations in B-mode.

While triangular-solve performance exemplifies sequentialism in the algorithm, it should be noted that we do achieve significant overall performance improvements between six nodes and 96 (16×6 grid) nodes, and that the repeatedly used B-mode factorization remains dominant compared to the triangular solves even for 128 nodes. Consequently, efforts aimed further to increase performance of the B-mode factorization (at the expense of additional A-mode work) are interesting to consider. For the factorizations, we also expect that we are achieving non-trivial speedups relative to one node, but we are unable to quantify this at present because of the memory limitations alluded to above.

Shape	Distribution: Row	Column	(time in seconds) A-Mode	B-Mode	Back-Solve	Solve	Avgs
6×1	Scatter	Scatter	4.859×10^2	2.145×10^2	3.025×10^0	6.696×10^0	3
3×3		Scatter	3.567×10^2	1.783×10^2	1.997×10^1	4.115×10^1	1
3×4		Scatter	3.101×10^2	1.303×10^2	2.149×10^1	4.452×10^1	1
4×3		Scatter	2.778×10^2	1.526×10^2	1.728×10^1	3.537×10^1	1
2×16		Scatter	4.500×10^2	3.350×10^2	3.175×10^0	1.101×10^1	1
12×1		Scatter	2.636×10^2	1.206×10^2	4.0188×10^0	8.340×10^0	3
16×1		Scatter	2.085×10^2	1.000×10^2	4.856×10^0	9.8744×10^0	3
8×2		Scatter	2.013×10^2	9.41×10^1	1.127×10^1	2.295×10^1	3
		$S = 10$	1.997×10^2	9.63×10^1	4.508×10^0	9.399×10^0	3
4×4		Scatter	2.371×10^2	1.056×10^2	1.225×10^1	3.549×10^1	3
		$S = 10$	2.329×10^2	1.104×10^2	4.192×10^0	9.406×10^0	3
4×6		Scatter	1.456×10^2	7.72×10^1	1.723×10^1	3.528×10^1	3
		$S = 10$	1.684×10^2	8.85×10^1	4.206×10^0	9.303×10^0	3
12×2		Scatter	1.490×10^2	6.95×10^1	9.08×10^0	1.851×10^1	3
		$S = 10$	1.425×10^2	6.54×10^1	4.557×10^0	9.439×10^0	3
12×3		Scatter	1.0429×10^2	5.39×10^1	9.34×10^0	1.898×10^1	3
		$S = 10$	1.0382×10^2	5.42×10^1	4.539×10^0	9.390×10^0	3
8×8		Scatter	1.154×10^2	6.16×10^1	1.1082×10^1	2.2906×10^1	3
		$S = 10$	1.145×10^2	6.64×10^1	4.4600×10^0	9.651×10^0	3
12×6		Scatter	6.470×10^1	3.527×10^1	9.410×10^0	1.9141×10^1	3
		$S = 10$	6.265×10^1	3.417×10^1	4.555×10^0	9.495×10^0	3
16×6		Scatter	5.014×10^1	2.744×10^1	9.085×10^0	1.8327×10^1	3
		$S = 10$	4.984×10^1	2.905×10^1	5.2811×10^0	1.0740×10^1	3
16×8		Scatter	7.046×10^1	3.879×10^1	8.9535×10^0	1.8243×10^1	3
		$S = 10$	6.70×10^1	3.854×10^1	5.239×10^0	1.0816×10^1	3

Table 9.5: Order 2500 Matrix Performance. Performance as a function of grid shape and size, and S-parameter. "Best" performance is for the 16×6 grid with $S = 10$.

9.6.7 Future Work, Conclusions

There are several classes of future work to be considered. First, we need to take the A-mode "analyze" phase to its logical completion, by including pivot-order sorting of the L/U pointer structures to improve performance for systems that should demonstrate sub-quadratic sequential complexity. This will require minor modifications to B-mode (that already takes advantage of column-traversing elimination), to reduce testing for inactive rows as the elimination progresses. We already realize optimal computation work in the triangular solves, and we mitigate the effect of $Q > 1$ quadratic communication work using the one-parameter distribution.

Second, we need to exploit "timelike" concurrency in linear algebra—multiple pivots. This has been addressed by Alaghband for shared-memory implementations of *MA28* with $O(N)$-complexity heuristics [Alaghband:89a]. These efforts must be reconsidered in the multicomputer setting and effective variations must be devised. This approach should prove an important source of additional speedup for many chemical engineering applications, because of the tendency towards extreme sparsity, with mainly band and/or block-diagonal structure.

Third, we could exploit new communication strategies and data redistribution. Within a process grid, we could incrementally redistribute L/U by utilizing the inherent broadcasts of L columns and U rows to improve load balance in the triangular solves at the expense of slightly more factorization computational overhead and significantly more memory overhead (nearly a factor of two). Memory overhead could be reduced at the expense of further communication if explicit pivoting were used concommitantly.

Fourth, we can develop adaptive broadcast algorithms that track the known load imbalance in the B-mode factorization, and shift greater communication emphasis to nodes with less computational work remaining. For example, the pivot column is naturally a "hot spot because the multiplier column (L column) must be computed before broadcast to the awaiting process columns. Allowing the non-pivot columns to handle the majority of the communication could be beneficial, even though this implies additional overall communication. Similarly, we might likewise apply this to the pivot row broadcast, and especially for the pivot process, because it must participate in two broadcast operations.

We could utilize two process grids. When rows (columns) of $U(L)$ are broadcast, extra broadcasts to a secondary process grid could reasonably be included. The secondary process grid could work on redistribution L/U to

an efficient process grid shape and size for triangular solves while the factorization continues on the primary grid. This overlapping of communication and computation could also be used to reduce the cost of transposing the solution vector from column-distributed to row-distributed, which normally follows the triangular solves.

The sparse solver supports arbitrary user-defined pivoting strategies. We have considered but not fully explored issues of fill-reduction *vs.* minimum time; in particular we have implemented a Markowitz-count fill-reduction strategy [Duff:86a]. Study of the usefulness of partial column pivoting and other strategies is also needed. We will report on this in the future.

Reduced-communication pivoting and parametric distributions can be applied immediately to concurrent dense solvers with definite improvements in performance. While triangular solves remain lower-order work in the dense case, and may sensibly admit less tuning in S, the reduction of pivot communication is certain to improve performance. A new dense solver exploiting these ideas is under construction at present.

In closing, we suggest that the algorithms generating the sequences of sparse matrices must themselves be reconsidered in the concurrent setting. Changes that introduce multiple right-hand sides could help to amortize linear algebra cost over multiple timelike steps of the higher-level algorithm. Because of inevitable load imbalance, idle processor time is essentially free—algorithms that find ways to use this time by asking for more speculative (partial) solutions appear of merit toward higher performance.

9.6.8 Acknowledgements

We wish to acknowledge the dense concurrent linear algebra library provided by Eric Van de Velde, as well as a prototype sparse concurrent linear algebra library, both of which were useful springboards for this work.

The first author acknowledges partial support under DOE grants DE-FG03-85ER25009 and DE-AC03-85ER40050. The second author (presently at the University of California, Santa Cruz) received support for his 1989 Caltech Summer Undergraduate Research Fellowship (SURF) under the same grants, and wishes to thank the Caltech SURF program for the opportunity to pursue the research discussed in part here.

The software implementation of this research was accomplished using machine resources made available by the Caltech Computer Science sub-Micron System Architectures Project and the Caltech Concurrent Supercomputer Facilities (CCSF).

9.7 Concurrent DASSL Applied to Dynamic Distillation Column Simulation

The accurate, high-speed solution of systems of ordinary differential-algebraic equations (DAE's) of low index is of great importance in chemical, electrical and other engineering disciplines. Petzold's Fortran-based *DASSL* is the most widely used sequential code for solving DAE's. We have devised and implemented a completely new C code, *Concurrent DASSL*, specifically for multicomputers and patterned on *DASSL*. In this work, we address the issues of data distribution and the performance of the overall algorithm, rather than just that of individual steps. *Concurrent DASSL* is designed as an open, application-independent environment below which linear algebra algorithms may be added in addition to standard support for dense and sparse algorithms. The user may furthermore attach explicit data interconversions between the main computational steps, or choose compromise distributions. A "problem formulator" (simulation layer) must be constructed above *Concurrent DASSL*, for any specific problem domain. We indicate performance for a particular chemical engineering application, a sequence of coupled distillation columns. Future efforts are cited in conclusion.

9.7.1 Introduction

In this paper, we discuss the design of a general-purpose integration system for ordinary differential-algebraic equations of low index, following up on our more preliminary discussion in [Skjellum:89a]. The new solver, *Concurrent DASSL*, is a parallel, C-language implementation of the algorithm codified in Petzold's *DASSL*, a widely used Fortran-based solver for DAE's [Petzold:83a], [Brenan:89a], and based on a loosely synchronous model of communicating sequential processes [Hoare:78a]. *Concurrent DASSL* retains the same numerical properties as the sequential algorithm, but introduces important new degrees of freedom compared to it. We identify the main computational steps in the integration process; for each of these steps, we specify algorithms that have correctness independent of data distribution.

We cover the computational aspects of the major computational steps, and their data distribution preferences for highest performance. We indicate the properties of the concurrent sparse linear algebra as it relates to the rest of the calculation. We describe the *proto-Cdyn* simulation layer, a distillation-simulation-oriented *Concurrent DASSL* driver which, despite specificity, exposes important requirements for concurrent solution of ordi-

nary DAE's; the ideas behind a template formulation for simulation are, for example, expressed.

We indicate formulation issues and specific features of the chemical engineering problem—dynamic distillation simulation. We indicate results for an example in this area, which demonstrates the feasibility of this method, but the need for additional future work, both on the sparse linear algebra, and on modifying the *DASSL* algorithm to reveal more concurrency, thereby amortizing the cost of linear algebra over more time steps in the algorithm.

9.7.2 Mathematical Formulation

We address the following initial-value problem consisting of combinations of N linear and nonlinear coupled, ordinary differential-algebraic equations over the interval $t \in [T_0, T_1]$:

$$\mathbf{IVP}(\mathbf{F}, \mathbf{u}, \mathbf{Z}_0, [T_0, T_1]; N, P):$$

$$\begin{aligned} \mathbf{F}(\mathbf{Z}, \dot{\mathbf{Z}}, \mathbf{u}; t) &= \mathbf{0}, \qquad t \in [T_0, T_1], \\ \mathbf{Z}(t = T_0) &\equiv \mathbf{Z}_0, \quad \dot{\mathbf{Z}}(t = T_0) \equiv \dot{\mathbf{Z}}_0, \end{aligned} \tag{9.5}$$

with unknown state vector $\mathbf{Z}(t) \in \Re^N$, known external inputs $\mathbf{u}(t) \in \Re^P$, where $\mathbf{F}(\bullet; t) \mapsto \Re^N$ and $\mathbf{Z}_0, \dot{\mathbf{Z}}_0 \in \Re^N$ are the given initial-value, derivative vectors, respectively. We will refer to Equation 9.5's deviation from $\mathbf{0}$ as the residuals or residual vector. Evaluating the residuals means computing $\mathbf{F}(\mathbf{Z}, \dot{\mathbf{Z}}, \mathbf{u}; t)$ ("model evaluation") for specified arguments $\mathbf{Z}$, $\dot{\mathbf{Z}}$, $\mathbf{u}$ and t.

DASSL's integration algorithm can be used to solve systems fully implicit in $\mathbf{Z}$ and $\dot{\mathbf{Z}}$ and of index zero or one, and specially structured forms of index two (and higher) [Brenan:89a, Chapter 5], where the index is the minimum number of times that part or all of Equation 9.5 must be differentiated with respect to t in order to express $\dot{\mathbf{Z}}$ as a continuous function of $\mathbf{Z}$ and t [Brenan:89a, page 17].

By substituting a finite-difference approximation $\mathcal{D}_i\mathbf{Z}$ for $\dot{\mathbf{Z}}$, we obtain:

$$\mathbf{F}_{\mathcal{D}}(\mathbf{Z}_i; \tau_i) \equiv \mathbf{F}(\mathbf{Z}_i, \mathcal{D}_i\mathbf{Z}_i, \mathbf{u}_i; t = \tau_i) = \mathbf{0}, \tag{9.6}$$

a set of (in general) nonlinear *staticized* equations. A sequence of Equation 9.6's will have to be solved, one at each discrete time $t = \tau_i$, $i = 1, 2, \ldots, M$[2], in the numerical approximation scheme; neither M nor the τ_i's

[2] and more at trial timepoints which are discarded by the integration algorithm.

need be pre-determined. In *DASSL*, the variable step-size integration algorithm picks the τ_i's as the integration progresses, based on its assessment of the local error. The discretization operator for $\dot{\mathbf{Z}}$, $\mathcal{D}$, varies during the numerical integration process and hence is subscripted as $\mathcal{D}_i$.

The usual way to solve an instance of the staticized equations, Equation 9.6, is via the familiar Newton-Raphson iterative method (yielding $\mathbf{Z}_i \equiv \mathbf{Z}_i^{\infty}$):

$$\mathbf{Z}_i^{k+1} = \mathbf{Z}_i^k - c \left\{ \nabla_{\mathbf{Z}} \mathbf{F}_{\mathcal{D}} \left(\mathbf{Z}_i^{m_k}; \tau_i \right) \right\}^{-1} \mathbf{F}_{\mathcal{D}} \left(\mathbf{Z}_i^k; \tau_i \right), \quad k = 0, 1, \ldots \tag{9.7}$$

given an initial, sufficiently good approximation $\mathbf{Z}_i^0$. The classical method is recovered for $m_k = k$ and $c = 1$, whereas a modified (damped) Newton-Raphson method results for $m_k < k$ (respectively, $c < 1$). In the original *DASSL* algorithm and in *Concurrent DASSL*, the Jacobian $\nabla_{\mathbf{Z}} \mathbf{F}_{\mathcal{D}}(\mathbf{Z})$ is computed by finite differences rather than analytically; this departure leads in another sense to a modified Newton-Raphson method even though $m_k = k$ and $c = 1$ might always be satisfied. For termination, a limit $k \leq k^*$ is imposed; a further stopping criterion of the form $\|\mathbf{Z}_i^{k+1} - \mathbf{Z}_i^k\| < \epsilon$ is also incorporated (see Brenan *et al.* [Brenan:89a, pages 121–124]).

Following Brenan *et al.*, the approximation $\mathcal{D}_i \mathbf{Z}$ is replaced by a BDF-generated linear approximation, $\alpha \mathbf{Z} + \beta$, and the Jacobian

$$\nabla_{\mathbf{Z}} \mathbf{F}(\mathbf{Z}, \alpha \mathbf{Z} + \beta, \mathbf{u}; t) = \frac{\partial \mathbf{F}}{\partial \mathbf{Z}} + \alpha \frac{\partial \mathbf{F}}{\partial \dot{\mathbf{Z}}}. \tag{9.8}$$

From this approximation, we define $\mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i)$ in the intuitive way. We then consider Taylor's Theorem with remainder, from which we can easily express a forward finite-difference approximation for each Jacobian column (assuming sufficient smoothness of $\mathbf{F}_{\alpha,\beta}$) with a scaled difference of two residual vectors:

$$\mathbf{F}_{\alpha,\beta}(\mathbf{Z} + \delta_j; \tau_i) - \mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i) = \left\{ \nabla_{\mathbf{Z}} \mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i) \right\} \delta_j + O\left(\|\delta_j\|_2^2 \right) \tag{9.9}$$

By picking δ_j proportional to $\mathbf{e}_j$, the jth unit vector in the natural basis for $\Re^N$, namely $\delta_j = d_j \mathbf{e}_j$, Equation 9.9 yields a first-order-accurate approximation in d_j of the jth column of the Jacobian matrix:

$$\begin{aligned} \frac{\mathbf{F}_{\alpha,\beta}(\mathbf{Z} + \delta_j; \tau_i) - \mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i)}{d_j} &= \left\{ \nabla_{\mathbf{Z}} \mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i) \right\} \mathbf{e}_j + O(d_j), \\ j &= 1, \ldots, N \end{aligned} \tag{9.10}$$

Each of these N Jacobian-column computations is independent and trivially parallelizable. It's well known, however, that for special structures such as banded and block n-diagonal matrices, and even for general sparse matrices, a single residual can be used to generate multiple Jacobian columns [Brenan:89a], [Duff:86a]. We discuss these issues as part of the concurrent formulation section below.

The solution of the Jacobian linear system of equations is required for each k-iteration, either through a direct (*e.g.*, LU-factorization) or iterative (*e.g.*, preconditioned-conjugate-gradient) method. The most advantageous solution approach depends on N as well as special mathematical properties and/or structure of the Jacobian matrix $\nabla_{\mathbf{Z}} \mathbf{F}_{\mathcal{D}}$. Together, the inner (linear equation solution) and outer (Newton-Raphson iteration) loops solve a single time point; the overall algorithm generates a sequence of solution points $\mathbf{Z}_i$, $i = 0, 1, \ldots, M$.

In the present work, we restrict our attention to direct, sparse linear algebra as described in [Skjellum:90d], although future versions of *Concurrent DASSL* will support the iterative linear algebra approaches by Ashby, Lee, Brown, Hindmarsh *et al.* [Ashby:90a], [Brown:91a]. For the sparse LU factorization, the factors are stored and reused in the modified Newton scenario. Then, repeated use of the old Jacobian implies just a forward and back-solve step using the triangular factors L and U. Practically, we can use the Jacobian for up to about five steps [Brenan:89a]. The useful lifetime of a single Jacobian evidently depends somewhat strongly on details of the integration procedure [Brenan:89a].

9.7.3 *proto-Cdyn* – Simulation Layer

To use the *Concurrent DASSL* system on other than toy problems, a simulation layer must be constructed above it. The purpose of this layer is to accept a problem specification from within a specific problem domain, and formulate that specification for concurrent solution as a set of differential-algebraic equations, including any needed data. On one hand, such a layer could explicitly construct the subset of equations needed for each processor, generate the appropriate code representing the residual functions, and create a set of node programs for effecting the simulation. This is the most flexible approach, allowing the user to specify arbitrary nonlinear DAE's. It has the disadvantage of requiring a lot of compiling and linking for each run in which the problem is changed in any significant respect (including but not limited to data distribution), although with sophisticated tactics, para-

metric variations within equations could be permitted without re-compiling from scratch, and incremental linking could be supported.

We utilize a template-based approach here, as we do in the Waveform-Relaxation paradigm for concurrent dynamic simulation [Skjellum:88a]. This is akin to the *ASCEND II* methodology utilized by Kuru and many others [Kuru:81a]. It is a compromise approach from the perspective of flexibility; interesting physical prototype subsystems are encapsulated into compiled code as templates. A template is a conceptual building block with states, non-states, parameters, inputs and outputs (see below). A general network made from instantiations of templates can be constructed at runtime without changing any executable code. User input specifies the number and type of each template, their interconnection pattern, and the initial value of systemic states and extraneous (non-state) variables, plus the value of adjustable parameters and more elaborate data, such as physical properties. The addition of templates requires new subroutines for the evaluation of the residuals of their associated DAE's, and also for interfacing to the remainder of the system (*e.g.*, parsing of user input, interconnectivity issues). With suitable automated tools, this addition process can be made straightforward to the user.

Importantly, the use of a template-based methodology does not imply a degradation in the numerical quality of the model equations or solution method used. We are not obliged to tear equations based on templates or groups of templates as is done in sequential-modular simulators [Westerberg:79a], [Cook:80a], where "sequential" refers in this sense to the stepwise updating of equation subsets, without connection to the number of computers assigned to the problem solution.

Ideally, the simulation layer could be made universal. That is, a generic layer of high flexibility and structural elegance would be created once and for all (and without predilection for a specific computational engine). Thereafter, appropriate templates would be added to articulate the simulator for a given problem domain. This is certainly possible with high-quality simulators such as *ASCEND II* and *Chemsim* (a recent Fortran-based simulator driving *DASSL* and *MA28* [Andersen:88a], [Petzold:83a], and [Duff:77a]). Even so, we have chosen to restrict our efforts to a more modest simulation layer, called *proto-Cdyn*, which can create arbitrary networks of coupled distillation columns. This restricted effort has required significant effort, and already allows us to explore many of the important issues of concurrent dynamic simulation. General-purpose simulators are for future consideration. They must address significant questions of user-interface in addition

to concurrency-formulation issues.

In the next paragraphs, we describe the important features of *proto-Cdyn*. In doing so, we indicate important issues for any *Concurrent DASSL* driver.

Template Structure

A template is a prototype for a sequence of DAE's which can be used repeatedly in different instantiations. Normally, but not always, the template corresponds to some subsystem of a physical-model description of a system, like a tank or distillation tray. The key characteristics of a template are: the number of integration states it incorporates (typically fixed), the number of non-state variables it incorporates (typically fixed), its input and output connections to other templates, and external sources (forcing functions) and sinks. State variables participate in the overall *DASSL* integration process. Non-states are defined as variables which, given the states of a template alone, may be computed uniquely. They are essentially local tear variables. It is up to the template designer whether or not to use such local tear variables: They impact the numerical quality of the solution, in principle. Alternative formulations, where all variables of a template are treated as states, can be posed, and comparisons made. Because of the superlinear growth of linear algebra complexity, the introduction of extra integration states must be justified on the basis of numerical accuracy. Otherwise, they artificially slow down the problem solution, perhaps significantly. Non-states are extremely convenient, and practically useful; they appear in all the dynamic simulators we have come across.

The template state and non-state structure implies a two-phase residual computation. First, given a state $\mathbf{Z}$, the non-states of each template are updated on a template-by-template basis. Then, given its states and non-states, inputs from other templates and external inputs, each template's residuals may be computed. In the sequential implementation, this poses no particular nuisances, other than two evaluation loops over all templates. However, in concurrent evaluation, a communication phase intervenes between non-state updates and residual updates. This communication phase transmits all states and non-states appearing as outputs of templates to their corresponding inputs at other templates. This transmission mechanism is considered further below under concurrent formulation.

Problem Preformulation

In general, the "optimal" ordering for the equations of a dynamic simulation will in general be too difficult to establish[3], because of the NP-hard issues involved in structure selection. However, many important heuristics can be applied, such as those that precedence order the nonlinear equations, and those that permute the Jacobian structure to a more nearly triangular or banded form [Duff:86a]. For the *proto-Cdyn* simulator, we skirt these issues entirely, because it proves easy to arrange a network of columns to produce a "good structure"—a main block tri-diagonal Jacobian structure with off-block-diagonal structure for the intercolumn connections, simply by taking the distillation columns with their states in tray-by-tray, top-down (or bottom-up) order.

Given a set of DAE's, and an ordering for the equations and states (*i.e.*, rows and columns of the Jacobian, respectively), we need to partition these equations between the multicomputer nodes, according to a two-dimensional process grid of shape $P \times Q = R$. The partitioning of the equations forms, in main part, the so-called "concurrent database." This grid structure is illustrated in [Skjellum:90d, Figure 2.]. In *proto-Cdyn*, we utilize a single process grid for the entire *Concurrent DASSL* calculation. That is, we do not currently exploit the *Concurrent DASSL* feature which allows explicit transformations between the main calculational phases (see below). In each process column, the entire set of equations is to be reproduced, so that any process column can compute not only the entire residual vector for a prediction calculation, but also, any column of the Jacobian matrix.

A mapping between the global equations and local equations must be created. In the general case, it will be difficult to generate a closed-form expression for either the global-to-local mapping or its inverse (that also require $< O(N)$ storage). At most, we will have on a hand a partial (or weak) inverse in each process, so that the corresponding global index of each local index will be available. Furthermore, in each node, a partial global-to-local list of indices associated with the given node will be stored in global sort order. Then, by binary search, a weak global-to-local mapping will be possible in each process. That is, each process will be able to identify if a global index resides within it, and the corresponding local index. A strong mapping for row (column) indices will require communication between all

[3]Optimality *per se* hinges on what our objective is. If, for instance, we want minimum time for LU factorization, still the objective of minimum fill-in does not guarantee minimum time in a concurrent setting.

the processes in a process row (respectively, column). In the foregoing, we make the tacit assumption that is is an unreasonable practice to use storage proportional to the entire problem size N in each node, except if this unscalability can be removed cheaply when necessary for large problems.

The *proto-Cdyn* simulator works with templates of specific structure—each template is a form of a distillation tray and generates the same number of integration states. It therefore skirts the need for weak distributions. Consequently, the entire row mapping procedure can be accomplished using the closed-form general two-parameter distribution function family ξ described in [Skjellum:90d], where the block size B is chosen as the number of integration states per template. The column mapping procedure is accomplished with the one-parameter distribution function family ζ also described in [Skjellum:90d]. The effects of row and column degree-of-scattering are described in [Skjellum:90d] with attention to linear algebra performance.

9.7.4 Concurrent Formulation

Overview

Next, we turn to Equation 9.5's (that is, **IVP**'s) concurrent numerical solution via the *DASSL* algorithm. We cover the major computational steps in abstract, and we also describe the generic aspects of *proto-Cdyn* in this connection. In the subsequent section, we discuss issues peculiar to the distillation simulation.

Broadly, the concurrent solution of **IVP** consists of three block operations: startup, dynamic simulation, and a cleanup phase. Significant concurrency is apparent only in the dynamic simulation phase. We will assume that the simulation interval requested generates enough work so that the startup and cleanup phases prove insignificant by comparison and consequently pose no serious Amdahl's-law bottleneck. Given this assumption, we can restrict our attention to a single step of **IVP** as illustrated schematically in Figure 9.6.

In the startup phase, a sequential host program interprets the user specification for the simulation. From this it generates the concurrent database: the templates and their mutual interconnections, data needed by particular templates, and a distribution of this information among the processes that are to participate. The processes are themselves spawned and fed their respective databases. Once they receive their input information, the processes re-build the data structures for interfacing with *Concurrent DASSL*,

Figure 9.6: Major Computational Blocks of a Single Integration Step. A single step in the integration begins with a number of BDF-related computations, including the solution "prediction" step. Then, "correction" is achieved through Newton iteration steps, each involving a Jacobian computation, and linear-system solution (LU factorization plus forward- / back-solves). The computation of the Jacobian in turn relies upon multiple independent residual calculations, as shown. The three items enclosed in the dashed oval (Jacobian computation (through at-most N Residual computations), and LU factorization) are, in practice, computed less often than the others—the old Jacobian matrix is used in the iteration loop until convergence slows intolerably.

and for generating the residuals. Tolerances, and initial derivatives must be computed and/or estimated. Furthermore, in each process column, the processes must rendezvous to finalize their communication labeling for the transmission of states and non-states to be performed during the residual calculation. This provides the basis for a reactive, deadlock-free update procedure described below.

The cleanup phase basically retrieves appropriate state values and returns them to the host for propagation to the user. Cleanup may actually be interspersed intermittently with the actual dynamic simulation. It provides simple bookkeeping of the results of simulation and terminates the concurrent processes at the simulation's conclusion.

The dynamic simulation phase consists of repetitive prediction and correction steps, and marches in time. Each successful time step requires the solution of one or more instances of Equation 9.6—additional timesteps that converge but fail to satisfy error tolerances, or fail to converge quickly enough, are necessarily discarded. In the next section, we cover the aspects of these operations in more detail, for a single step.

Single Integration Step

The Integration Computations of *DASSL* are a fixed leading-coefficient, variable-stepsize and order, backward-differentiation-formula (BDF) implicit integration scheme, described clearly in [Brenan:89a, Chapter 5] and outlined in [Petzold:83a]. *Concurrent DASSL* faithfully implements this numerical method, with no significant differences. Test problems run with the *DASSL* Fortran code and the new C code (on one and multiple computers) certify this degree of compatibility.

The sequential time complexity of the integration computations is $O(N)$, if considered separately from the residual calculation called in turn, which is also normally $O(N)$ (see below). We pose these operations on a $P \times Q = R$ grid, where we assume that each process column can compute complete residual vectors. Each process column repeats the entire prediction operations: there is no speedup associated with $Q > 1$, and we replicate all *DASSL* BDF and predictor vectors in each process column. Taller, narrower grids are likely to provide the overall greatest speedup, though the residual calculation may saturate (and slow down again) because of excessive vertical communication requirements—It's definitely not true that the $R \times 1$ shape is optimal in all cases.

The distribution of coefficients in the rows has no impact on the integra-

tion operations, and is dictated largely by the requirements of the residual calculation itself. In practical problems, the concurrent database cannot be reproduced in each process (*cf.*, [Lorenz:89b]), so a given process will only be able to compute some of the residuals. Furthermore, we may not have complete freedom in scattering these equations, because there will often be a tradeoff between the degree of scattering and the amount of communication needed to form the entire residual vector.

The amount of $O(N)$ integration-computation work is not terribly large—there is consequently a non-trivial but not tremendous effort involved in the integration computations. (Residual computations dominate in many if not most circumstances.) Integration operations consist mainly of vector-vector operations not requiring any interprocess communication and, in addition, fixed startup costs. Operations include prediction of the solution at the time point, initiation and control of the Newton iteration that "corrects" the solution, convergence and error-tolerance checking, and so forth. For example, the approximation $\mathcal{D}_i$ is chosen within this block using the BDF formulas. For these operations, each process column currently operates independently, and repetitively forms the results. Alternatively, each process column could stride with step Q, and row-*combines* could be used to propagate information across the columns [Skjellum:90a]. This alternative would increase speed for sufficiently large problems, and can easily be implemented. However, because of load-imbalance in other stages of the calculation, we are convinced that including this type of synchronization could be an overall negative rather than positive to performance. This alternative will nevertheless be a future user-selectable option.

Included in these operations are a handful of norm operations, which constitute the main interprocess communication required by the integration computations step; norms are implemented concurrently via recursive doubling (*combine*) [Stone:87a], [Skjellum:90a]. Actually, the weighted norm used by *DASSL* requires two recursive doubling operations, each *combines* a scalar: first to obtain the vector coefficient of maximum absolute value, then to sum the weighted norm itself. Each can be implemented as Q independent column *combines*, each producing the same repetitive result, or a single Q-striding norm, that takes advantage of the repetition of information, but utilizes two *combines* over the entire process grid. Both are supported in *Concurrent DASSL*, although the former is the default norm. As with the original *DASSL*, the norm function can be replaced, if desired.

Single Residuals are computed in prediction, and as needed during correction. Multiple residuals are computed when forming the finite-difference Jacobian. Single residuals are computed repetitively in each process column, whereas the multiple residuals of a Jacobian computation are computed uniquely in the process columns.

Here, we consider the single residual computation required by the integration computations just described. Given a state vector $\mathbf{Z}$, and approximation for $\dot{\mathbf{Z}}$, we need to evaluate $\mathbf{F}(\mathbf{Z}, \dot{\mathbf{Z}}, \tau_i) \equiv \mathbf{F}_{\mathcal{D}}(\mathbf{Z}, \tau_i)$. The exploitable concurrency available in this step is strictly a function of the model equations. As defined, there are N equations in this system, so we expect to use at best N computers for this step. Practically, there will be interprocess communication between the process rows, corresponding to the connectivity among the equations. This will place an upper limit on $P \leq K$ (the number of row processes) that can be used before the speed will again decrease: we can expect efficient speedup for this step provided that the cost of the interprocess communication is insignificant compared to the single-equation grain size. As estimated in [Skjellum:90a], the granularity T_{comm}/T_{calc} for the Symult s2010 multicomputer is about fifty, so this implies about four hundred and fifty floating point operations per communication in order to achieve 90% concurrent efficiency in this phase.

Jacobian Computation There is evidently much more available concurrency in this computational step than for the single residual and integration operations, since, for finite differencing, N independent residual computations are apparently required, each of which is a single-state perturbation of $\mathbf{Z}$. Based on our overview of the residual computation, we might naively expect to use $K \times N$ processes effectively; however, the simple perturbations can actually require much less model evaluation effort because of latency [Duff:86a], [Kuru:81a], which is directly a function of the sparsity structure of the model equations, Equation 9.5. In short, we can attain the same performance with much less than $K \times N$ processors.

In general, we'd like to consider the Jacobian computation on a rectangular grid. For this, we can consider using $P \times Q = R$ to accomplish the calculation. With a general grid shape, we exploit some concurrency in *both* the column evaluations and in the residual computations, with $T_{Jac,P\times Q=R}$ the time for this step, $S_{Jac,P\times Q=R}$ the corresponding speedup, $T_{res,P}$ the residual evaluation time with P row processes, and $S_{res,P}$ the apparent speedup

compared to one row process:

$$T_{Jac,P\times Q=R} \approx \lceil N/Q \rceil \times T_{res,P}$$
$$S_{Jac,P\times Q=R} \approx \frac{N}{\lceil N/Q \rceil} \times S_{res,P}$$

assuming no shortcuts are available as a result of latency. This timing is exemplified in the example below, which does not take advantage of latency.

There is additional work whenever the Jacobian structure is rebuilt for better numerical stability in the subsequent LU factorization (A-mode). Then, $O(N^2/PQ)$ work is involved in each process in the filling of the initial Jacobian. In the normal case, work proportional to the number of local nonzeroes plus fill elements is incurred in each process for re-filling the sparse Jacobian structure.

Exploitation of Latency has been considered in the *Concurrent DASSL* framework. We currently have experimental versions of two mechanisms, both of which are designed to work with the sparse-matrix structures associated with direct, sparse LU factorization (see [Skjellum:90d]). The first is called "bandlike" Jacobian evaluation. For a banded Jacobian matrix of bandwidth b, only b residuals are needed to evaluate the Jacobian. This feature is incorporated into the original *DASSL*, along with a LINPACK banded solver. In *Concurrent DASSL*, collections of Jacobian columns are placed in each process column, according to the column data distribution, which thus far is picked solely to balance LU factorization and triangular-solve performance [Skjellum:90d]. In each process column, there will be "compatible" columns that can be evaluated using a single, composite perturbation. Identification of these compatible columns is accomplished by checks on the bandwidth overlap condition. Columns that possess off-band structure are stricken from the list and evaluated separately. Presumably, a heuristic algorithm could be employed further to increase the size of the compatible sets, but this is yet to be implemented. The same algorithm "greedy" algorithm of Curtis *et al.* used for the sequential reduction of Jacobian computation effort would be applied independently to each process column (see comments by [Duff:86a, Section 12.3]). Then, clearly, the column distribution effects the performance of the Jacobian computation, and the linear-algebra performance can no longer be viewed so readily in isolation.

We have also devised a "blocklike" format, which will be applied to block n-diagonal matrices that include some off-block entries as well. Optimally,

fewer residual computations will be needed than for the banded case. The same column-by-column compatible sets will be created, and the Curtis algorithm can also be applied. Hopefully, because of the less restrictive compatibility requirement, the "blocklike" case will produce higher concurrent speedups than that attained using the conservative bandlike assumption for Jacobians possessing blocklike structure. Comparative results will be presented in a future paper.

The LU Factorization Following the philosophy of Harwell's *MA28*, we have interfaced a new concurrent sparse solver to *Concurrent DASSL*, the details of which are quoted elsewhere in this proceedings [Skjellum:90d]. In short, there is a two-step factorization procedure: A-mode, which chooses stable pivots according to a user-specified function, and builds the sparse data structures dynamically; and B-mode, which re-uses the data structures and pivot sequence on a similar matrix, but monitors stability with a growth-factor test. A-mode is repeated whenever necessary to avoid instability. We expect sub-cubic time complexity and sub-quadratic space complexity in N for the sparse solver. We attain acceptable factorization speedups for systems that are not narrow banded, and of sufficient size. We intend to incorporate multiple pivoting heuristic stategies, following [Alaghband:89a], further to improve performance of future versions of the solver. This may also contribute to better performance of the triangular solves.

Forward- and Back-solving Steps take the factored form $\mathsf{P}_R A \mathsf{P}_C^T = \hat{L}\hat{U}$, with $\hat{L}$ unit lower-triangular, $\hat{U}$ upper-triangular, and permutation matrices P_R, P_C, and solve $Ax = b$, using the implicit pivoting approach described in [Skjellum:90d]. Sequentially, the triangular solves each require work proportional to the number of entries in the respective triangular factor, including fill-in. We have yet to find an example of sufficient size for which we actually attain speedup for these operations, at least for the sparse case. At most, we try to prevent these operations from becoming competitive in cost to the B-mode factorization; we detail these efforts in [Skjellum:90d]. In brief, the optimum grid shape for the triangular solves has $Q = 1$, and P somewhat reduced than what we can use in all the other steps. As stated, P small seems better thus far, though for many examples, the increasing overhead as a function of increasing P is not unacceptable (see [Skjellum:90d] and the example below).

Residual Communication is an important aspect of the *proto-Cdyn* layer. As indicated in the startup-phase discussion, the members of a process column initially share information about the groups of states and non-states they will exchange during a residual computation. For residual communication, a reactive transmission mechanism is employed, to avoid deadlocks. Each process transmits its next group of states to the appropriate process and then looks for any receipt of state information. Along with the state values are indices that directly drive the destinations for these values. This index information is shared during the startup phase and allows the messages to drive the operation. Through non-blocking receives, this procedure avoids problems of transmission ordering. Regardless of the template structure, at most one send and receive is needed between any pair of column processes.

9.7.5 Chemical Engineering Example

The algorithms and formalism needed to run this example amount to about 70,000 lines of C code including the simulation layer, *Concurrent DASSL*, the linear algebra packages, and support functions [Skjellum:90a], [Skjellum:90c], [Skjellum:90d].

In this simulation, we consider seven distillation columns arranged in a tree-sequence [Skjellum:90c], working on the distillation of eight alcohols: methanol, ethanol, propan-1-ol, propan-2-ol, butan-1-ol, 2-methyl propan-1-ol, butan-2-ol, and 2-methyl propan-2-ol. Each column has 143 trays. Each tray is initialized to a non-steady condition, and the system is relaxed to the steady state governed by a single feed stream to the first column in the sequence. This setup generates suitable dynamic activity for illustrating the cost of a single "transient" integration step.

We note the performance in Table 9.6. Because we have not exploited latency in the Jacobian computation, this calculation is quite expensive, as seen for the sequential times on a Sun 3/260 depicted there. (The timing for the Sun 3/260 is quite comparable to a single Symult s2010 node and was lightly loaded during this test run.) As expected, Jacobian calculations speedup efficiently, and we are able to get approximately a speedup of 100 for this step using 128 nodes. The A-mode linear algebra also speeds up significantly. The B-mode factorization speeds up negligibly and quickly slows down again for more than 16 nodes. Likewise, the triangular solves are significantly slower than the sequential time. It should be noted that B-mode reflects two orders of magnitude speed improvement over A-mode.

	(time in seconds)				
Grid Shape	Jacobian	A-mode	B-mode	Back-Solve	Solve
1x1	64672.2	5089.96	61.82	2.5	4.7
8x1	6870.82	1024.41	47.827	15.619	30.825
16x1	3505.13	547.625	52.402	19.937	39.491
32x1	1829.93	316.544	56.713	24.383	47.692
64x1	1060.40	219.148	77.302	39.942	59.553
32x4	491.526	181.082	71.482	57.049	101.994
64x2	520.029	161.052	82.696	46.013	86.935
128x1	608.946	170.022	90.905	37.498	67.982

Table 9.6: Order 9009 dynamic Simulation Data. Key single-step calculation times with the 1×1 case run an unloaded Sun 3/260 (similar performance-wise to a single Symult s2010 node) for comparison. The Jacobian rows were distributed in block-linear form, with $B = 9$, reflecting the distillation-tray structure. The Jacobian columns were scattered. This is an seven column simulation of eight alcohols, with a total of 1,001 trays. See [Skjellum:90d] for more on data distributions.

This reflects the fact that we are seeing almost linear time complexity in B-mode, since this example has a narrow block tri-diagonal Jacobian with too little off-diagonal coupling to generate much fill-in. It seems hard to imagine speeding up B-mode for such an example, unless we can exploit multiple pivots. We expect multiple-pivot heuristics to do reasonably well for this case, because of its narrow structure, and nearly block tri-diagonal structure. We have used Wilson Equation Vapor-Liquid Equilibrium with the Antoine Vapor equation. We have found that the thermodynamic calculations were much less demanding than we expected, with bubble-point computations requiring "$1 + \epsilon$ iterations to converge. Consequently, there was not the greater weight of Jacobian calculations we expected beforehand. Our model assumes constant pressure, and no enthalpy balances. We include no flow dynamics and include liquid and vapor flows as states, because of the possibility of feedbacks.

Were we to utilize latency in the Jacobian calculation, we could reduce the sequential time by a factor of about 100. This improvement would also carry through to the concurrent times for Jacobian solution. At that

ratio, Jacobian computation to B-mode factorization has a sequential ratio of about 10:1. As is, we achieve legitimate speedups of about five. We expect to improve these results using the ideas quoted elsewhere here and in [Skjellum:90d].

From a modeling point-of-view, two things are important to note. First, the introduction of more non-ideal thermodynamics would improve speedup, because these calculations fall within the Jacobian computation phase and Single-Residual Computation. Furthermore, the introduction of a more realistic model will likewise bear on concurrency, and likely improve it. For example, introducing flow dynamics, enthalpy balances and vapor holdups makes the model more difficult to solve numerically (higher index). It also increases the chance for a wide range of step-sizes, and the possible need for additional A-mode factorizations to maintain stability in the integration process. Such operations are more costly, but also have a higher speedup. Furthermore, the more complex models will be less likely to have near diagonal dominance; consequently more pivoting is to be expected, again increasing the chance for overall speedup compared to the sequential case. Mainly, we plan to consider the Waveform-Relaxation approach more heavily, and also to consider new classes of dynamic distillation simulations with *Concurrent DASSL* [Skjellum:90c].

9.7.6 Conclusions

We have developed a high-quality concurrent code, *Concurrent DASSL*, for the solution of ordinary differential-algebraic equations of low index. This code, together with appropriate linear algebra and simulation layers, allows us to explore the achievable concurrent performance of non-trivial problems. In chemical engineering, we have applied it thus far to a reasonably large, simple model of coupled distillation columns. We are able to solve this large problem, which is quite demanding on even a large mainframe because of huge memory requirements and non-trivial computational requirements; the speedups achieved thus far are legitimately at least five, when compared to an efficient sequential implementation. This illustrates the need for improvements to the linear algebra code, which are feasible because sparse matrices will admit multiple pivots heuristically. It also illustrates the need to consider hidden sources of additional timelike concurrency in *Concurrent DASSL*, perhaps allowing multiple right-hand sides to be attacked simultaneously by the linear algebra codes, and amortizing their cost more efficiently. Furthermore, the performance points up the need for detailed research into

the novel numerical techniques, such as Waveform Relaxation, which we have begun to do as well [Skjellum:88a].

9.7.7 Acknowledgements

The first author acknowledges the kind assistance and helpful cooperation of Lionel F. Laroche and Henrik W. Andersen in the area of dynamic simulation for chemical process flowsheets. We have spent many hours together over the last twenty months in the discussion of design goals, features, algorithms, on realizations, post-mortems and re-designs, and in overcoming the stumbling blocks in our respective simulation codes. Thanks also to Professor A. W. Westerberg of CMU, who offered helpful suggestions when he visited Caltech in 1989.

Thanks to Drs. K. E. Brenan, S. L. Campbell and Linda Petzold, for sharing advance drafts of their monograph *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, which proved very helpful in the creation of *Concurrent DASSL*.

The first author acknowledges partial support under DOE grants DE-FG03-85ER25009 and DE-AC03-85ER40050.

Concurrent DASSL was developed using machine resources made available by the Caltech Computer Science sub-Micron System Architectures Project and the Caltech Concurrent Supercomputer Facilities (CCSF).

9.8 Concurrent Adaptive Multigrid

9.8.1 Introduction

Simple relaxation methods reduce the high frequency components of the solution error by an order of magnitude in a few iterations. This observation is used to derive the multigrid method, see Brandt [Brandt:77a], Hackbusch [Hackbusch:85a], and Hackbusch and Trottenberg [Hackbusch:82a]. In the multigrid method, a smoothed problem is projected to a coarser grid. This coarse-grid problem is then solved recursively by smoothing and coarse-grid correction. The recursion terminates on the coarsest grid, where an exact solver is used. In the full multigrid method, a coarser grid is also used to compute an initial guess for the multigrid iteration on a finer grid. With this method, it is possible to solve the problem with an operation count proportional to the number of unknowns.

Multigrid methods are best understood for elliptic problems, e.g., the Poisson equation, stationary reaction-diffusion equations, implicit time-steps in parabolic problems, etc. However, the multigrid approach is successful for many other applications, from fluid flow to computer vision. Parallelization issues are independent of particular applications, and elliptic problems are a good test bed for the study of concurrent multigrid. We chose two- and three-dimensional stationary nonlinear reaction-diffusion equations in a rectangular domain Ω as our model problem, i.e.:

$$-\Delta u + g(\vec{x}, u) = 0$$

with suitable boundary conditions.

To parallelize multigrid, we proceeds as follows (see also [Velde:87a], [Velde:87b]). First, a sequential multigrid procedure is developed. Here, the basic numerical problems are addressed: which smoothing, restriction, and prolongation operators to use, the resolution required (size of the finest grid), the number of levels (size of the coarsest grid), and the coarsest-grid solver. Second, this sequential multigrid code is generalized to include local grid refinement (in the neighborhood of singularities, for example). Three basic problems are addressed in this second stage: the algorithmic aspect of local grid refinement, the numerical treatment of interior boundaries, and the relaxation of partially overlapping grid patches. In the third and last step, the multigrid code is parallelized. This can now be done without introducing new numerical issues. Each concurrent process starts a sequential multigrid procedure, each one locally refining to a particular subdomain. To achieve this, a communication operation for the exchange of interior boundary values is needed. Depending on the size of the coarsest grid, it might be required to develop, independently, a concurrent coarsest grid solver.

This parallelization strategy has the advantage that all numerical problems can be addressed in the sequential stages. Although our implementation is for regular grids, the same strategy is also valid for irregular grids.

9.8.2 The Basic Algorithm

To simplify the switch to adaptive grids later, we use a multigrid variant known as the Full Approximation Scheme. Thus, on every level, we compute an approximation to the solution of the original equation, not of an error equation. This multigrid procedure is defined by the following basic building blocks: a coarsest-grid solver, a solution restriction operator, a right hand side restriction operator, a prolongation operator, and a smoothing operator.

Two feasible coarsest-grid solvers are relaxation until convergence and a direct solver (embedded in a Newton iteration if the problem is nonlinear). The cost of solving a problem on the coarsest grid is, of course, related to the size of the coarsest grid. If the coarsest grid is very coarse, the cost is negligible. However, numerical reasons often dictate a minimum resolution for the coarsest grid. Moreover, elaborate computations may take place on the coarsest grid, see [Bolstadt:86a], [Chan:82a], [Dinar:85a] for examples of multigrid continuation. In some instances, the performance of the computations on the coarsest grids cannot be neglected.

Many alternatives exist for smoothing. Parallelization will be easiest with point relaxations. Underrelaxed Gauss-Jacobi and red-black Gauss-Seidel relaxation are particularly suited for concurrent implementations and for adaptive grids. Hence, we shall restrict our attention to point relaxation methods.

The intergrid transfers are usually simple: linear interpolation as the prolongation operator, injection or full-weight restriction as the restriction operator.

The main data structure of the sequential nonadaptive algorithm is a doubly linked list of grids, where a grid structure provides memory for the solution and right hand side vectors, and each grid is connected to one finer and to one coarser grid. The sequential multigrid code has the following structure: a library of operations on grid functions, a code related to the construction of a doubly linked list of grids, and the main multigrid algorithm. We maintain this basic structure for the concurrent and adaptive algorithms. Although the doubly linked list of grids will be replaced by a more complex structure, the basic multigrid algorithm will not be altered. While the library of grid function operations will be expanded, the fundamental operations will remain the same. This is important, because the basic library for a general multigrid package with several options for each operator is large.

9.8.3 The Adaptive Algorithm

Here, we focus on the use of adaptive grids for sequential computations. We shall apply these ideas in the next section to achieve concurrency. Figure 9.7 illustrates the grid structure of a one dimensional adaptive multigrid procedure. Fine grids are introduced only where necessary, in the neighborhood of a singularity, for example. In two and three dimensions the topology is more complicated, and it makes sense to refine in several subdomains that

Figure 9.7: One Dimensional Adaptive Multigrid Structure.

partially overlap.

We focus first on the intergrid transfers. Although these operators are straightforward, they are the source of some implementation difficulties for the concurrent algorithm, because load balanced data distributions of fine and coarse grids are not compatible. The structure introduced here avoids these difficulties. Before introducing a fine grid on a subdomain, we construct an artificial coarse grid on the same subdomain. This artificial coarse grid, called a child-grid, differs from a normal grid data structure only because its data vectors (the solution and right hand side) are subvectors of the parent-grid data vectors. Thus, child-grids do not use extra memory for data (except for some negligible amount for bookkeeping). In figure 9.7, grid 1 is a parent-grid with two children, grids 2 and 3. With child-grids, the intergrid transfers of the nonadaptive procedure can be reused. The restriction, for example, takes place between a fine grid (defined over a subdomain) and a coarse child-grid (in figure 9.7, between grids 4 and 2 and between grids 5 and 3, respectively). Because data memory of child- and parent-grid are shared, the appropriate subvectors of the coarse grid data are updated automatically. Similarly, prolongation occurs between the child-grid and the fine grid.

The basic data structure of the nonadaptive procedure, the doubly linked list of grids, is transformed radically as a result of child-grids and their refinements. The data structure is now a tree of doubly linked lists, see figure 9.7. As indicated before, the intergrid transfers are not affected by this complicated structure. For relaxation, the only significant difference is that more than one grid may have to be relaxed on each level. When the boundary of one grid intersects the interior of another grid on the same

Figure 9.8: Boundary Interpolation.

level, the boundary values must be interpolated, see figure 9.8. This does not affect the relaxation operators, as long as the relaxation step is preceded by a boundary-interpolation step.

9.8.4 The Concurrent Algorithm

The same structure that made the multigrid code adaptive, allows us to parallelize it. For now, assume that every process starts out with a copy of the coarsest grid, defined on the whole domain. Each process is assigned a subdomain where to compute the solution to maximum accuracy. The collection of subdomains assigned to all processes covers the computational domain. Within each process, an adaptive grid structure is constructed so that the finest level at which the solution is needed envelops the assigned subdomain, see figure 9.9. The set of all grids (in whatever process they reside) forms a tree structure like the one described in the previous section. The same algorithm can be applied to it. Only one addition must be made to the program: overlapping grids on the same level but residing in different processes must communicate in order to interpolate boundary values. This is an operation to be added to the basic library.

The coarsest grid can often be ignored as far as machine efficiency is concerned. As mentioned in section 9.8.2, the computations on the coarsest grid are sometimes substantial. Then, it is crucial to parallelize the coarsest-grid computations. With relaxation until convergence as the coarsest-grid solver, one could simply divide up the coarsest grid over all concurrent processes. It is more likely, however, that the coarsest-grid computations involve a direct

Figure 9.9: Use of Adaptive Multigrid for Concurrency.

solution method. In this case, the duplicated coarsest grid is well suited as an interface to a concurrent direct solver, because it simplifies the initialization of the coefficient matrix. We refer to section 8.2 and [Velde:90a] for details on some direct solvers.

The total algorithm, adaptive multigrid and concurrent coarsest grid solver, is heterogeneous: its communication structure is irregular and varies significantly from one part of the program to the next, the data distribution for optimal load balance is difficult to predict. On the coarsest level, we achieve load balance by exploiting the data distribution independence of our linear algebra code, see [Lorenz:89b]. On the finer levels, load balance is obtained by allocating an approximately equal number of finest-grid points to each process.

9.8.5 Credits and C3P references

The concurrent multigrid program was developed by Eric F. Van de Velde. Relevant C3P references are [Lorenz:89a], [Lorenz:89b], [Velde:87a], [Velde:87b], [Velde:89b], [Velde:90a].

9.9 Concurrent Implementation Of Munkres Algorithm

9.9.1 Introduction

The so-called *Assignment Problem* is of considerable importance in a variety of applications, and can be stated as follows. Let

$$\mathcal{A} \equiv \{a_1, a_2, \ldots, a_{N_A}\} \tag{9.11}$$

and

$$\mathcal{B} \equiv \{b_1, b_2, \ldots, b_{N_B}\} \tag{9.12}$$

be two sets of items and let

$$d_{ij} \equiv d[a_i, b_j] \geq 0,\ a_i \in \mathcal{A},\ b_j \in \mathcal{B} \tag{9.13}$$

be a measure of the distance (dissimilarity) between individual items from the two lists. Taking $N_A \leq N_B$, the objective of the assignment problem is to find the particular mapping

$$i \mapsto \Pi(i),\ 1 \leq i \leq N_A,\ 1 \leq \Pi(i) \leq N_B \tag{9.14}$$

$$i \neq j \Rightarrow \Pi(i) \neq \Pi(j)$$

such that the total association score

$$S_{TOT} \equiv \sum_{i=1}^{N_A} d[i, \Pi(i)] \tag{9.15}$$

is minimized over all permutations Π.

For $N_A \leq N_B$, the naive (exhaustive search) complexity of the assignment problem is $O[N_B!/(N_B - N_A)!]$. There are, however, a variety of exact solutions to the assignment problem with reduced complexity $O[N_A^2 N_B]$, ([Blackman:86a], [Burgeios:71a], [Kuhn:55a]). Section 9.9.2 briefly describes one such method, Munkres Algorithm [Kuhn:55a], and presents a particular sequential implementation. Performance of the algorithm is examined for the particularly nasty problem of associating lists of random points within the unit square. In Section 9.9.3, the algorithm is generalized for concurrent execution, and performance results for runs on the Mark III hypercube are presented.

9.9.2 The Sequential Algorithm

The input to the assignment problem is the matrix $D \equiv \{d_{ij}\}$ of dissimilarities from Equation 9.13. The first point to note is that the particular assignment which minimizes Equation 9.15 is not altered if a *fixed* value is added to or subtracted from all entries in any row or column of the cost matrix D. Exploiting this fact, Munkres solution to the Assignment Problem can be divided into two parts

M1 : Modifications of the distance matrix D by row/column subtractions, creating a (large) number of zero enties.

M2 : With $\{R_Z(i)\}$ denoting the row indices of all zeros in column i, construction of a so-called *Minimal Representative Set*, meaning a distinct selection $R_Z(i)$ for each i, such that $i \neq j \Rightarrow R_Z(i) \neq R_Z(j)$.

The steps of Munkres algorithm generally follow those in the constructive proof of P. Hall's theorem on Minimal Representative Sets.

The preceding paragraph provides a hopelessly incomplete hint as to the number theoretic basis for Munkres Algorithm. The particular implementation of Munkres algorithm used in this work is as described in Chapter 14 of [Blackman:86a]. To be definite, take $N_A \leq N_B$, and let the columns of the distance matrix be associated with items from list $\mathcal{A}$. The first step is to subtract the smallest item in each column from all entries in the column. The rest of the algorithm can be viewed as a search for *special* zero entries (starred zeros Z^*), and proceeds as follows:

Munkres Algorithm

Step 1 : Setup

1. Find a zero Z in the distance matrix.
2. If there is no starred zero already in its row or column, star this zero.
3. Repeat steps 1.1, 1.2 until all zeros have been considered.

Step 2 : Z^* Count, Solution Assessment.

1. Cover every column containing a Z^*.
2. Terminate the algorithm if all columns are covered. In this case, the locations of the Z^* entries in the matrix provide the solution to the assignment problem.

Step 3 : Main Zero Search

1. Find an uncovered Z in the distance matrix and prime it, $Z \mapsto Z'$. If no such zero exists, go to Step 5
2. If No Z^* exists in the row of the Z', go to Step 4.
3. If a Z^* exists, cover this row and uncover the column of the Z^*. Return to Step 3.1 to find a new Z.

Step 4 : Increment Set Of Starred Zeros

1. Construct the 'Alternating Sequence' of primed and starred zeros:
   Z_0 : Unpaired Z' from Step 3.2.
   Z_1 : The Z^* in the column of Z_0
   Z_{2N} : The Z' in the row of Z_{2N-1}, *if* such a zero exists.
   Z_{2N+1} : The Z^* in the column of Z_{2N}.
   the sequence eventually terminates with an unpaired $Z' = Z_{2N}$ for some N.
2. Unstar each starred zero of the sequence.
3. Star each primed zero of the sequence, thus increasing the number of starred zeros by one.
4. Erase all primes, uncover all columns and rows, and return to Step 2.

Step 5 : New Zero Manufactures

1. Let h be the smallest uncovered entry in the (modified) distance matrix.
2. Add h to all covered rows.
3. Subtract h from all uncovered columns
4. Return to Step 3, without altering stars, primes or covers.

A (very) schematic flowchart for the algorithm is shown in Figure 9.10. Note that Steps 1,5 of the algorithm overwrite the original distance matrix.

The preceding algorithm involves flags (starred or primed) associated with zero entries in the distance matrix, as well as 'Covered' tags associated with individual rows and columns. The implementation of the zero tagging is done by first noting that there is *at most* one Z^* or Z' in any row or column. The covers and zero tags of the algorithm are accordingly implemented using five simple arrays:

Figure 9.10: Flowchart for Munkres Algorithm

CC(k) : Covered column tags, $1 \leq k \leq N_{COLS}$.

CR(j) : Covered row tags, $1 \leq j \leq N_{ROWS}$

ZS(k) : Z^* locators for columns of the matrix. If positive, ZS(k) is the row index of the Z^* in the k^{th} column of the matrix.

ZR(j) : Z^* locators for rows of the matrix. If positive, ZR(j) is the column of the Z^* in the j^{th} row of the matrix.

ZP(j) : Z' locators for rows of the matrix. If positive, ZP(j) is the column of the Z' in the j^{th} row of the matrix.

Entries in the cover arrays CC and CR are one if the row or column is covered zero otherwise. Entries in the zero-locator arrays ZS, ZR and ZP are zero if no zero of the appropriate type exists in the indexed row or column.

With the Star-Prime-Cover scheme of the preceding paragraph, a sequential implementation of Munkres algorithm is completely straightforward. At the beginning of Step 1, all cover and locator flags are set to zero, and the initial zero search provides an initial set of non-zero entries in ZS(). Step 2 sets appropriate entries in CC() to one and simply counts the covered columns. Step 3 and Step 5 are trivially implemented in terms of the Cover/Zero arrays and the 'Alternating Sequence' for Step 4 is readily constructed from the contents of ZS(), ZR() and ZP().

As an initial exploration of Munkres algorithm, consider the task of associating two lists of random points within a 2D unit square, taking the cost function in Equation 9.13 to be the usual Cartesian distance. Figure 9.11

Figure 9.11: Timing Results for the Sequential Algorithm versus Problem Size

plots total CPU times for execution of Munkres algorithm for equal size lists versus list size. The vertical axis gives CPU times in seconds for one node of the Mark III hypercube. The circles and crosses show the time spent in Step 5 and Step 3, respectively. These two steps (zero search and zero manufacture) account for essentially *all* of the CPU time. For the 190×190 case, the total CPU time spent in Step 2 was about 0.9 CPU sec, and that spent in Step 4 was too small to be reliably measured. The large amounts of time spent in Step 3 and Step 5 arise from the very large numbers of times these parts of the algorithm are executed. The 190×190 case involves 6109 entries into Step 3 and 593 entries into Step 5.

Since the zero searching in Step 3 of the algorithm is required so often, the implementation of this step is done with some care. The search for zeros is done column-by-column, and the code maintains pointers to both the last column searched and the most recently uncovered column (Step 3.3) in order to reduce the time spent on subsequent re-entries to the Step 3 box of Figure 9.10.

The dashed line if Figure 9.11 indicates the nominal $\Delta T \propto N^3$ scaling predicted for Munkres algorithm. By and large, the timing results in Figure 9.11 are consistent with this expected behavior. It should be noted, however, that both the nature of this scaling and the coefficient of N^3 are very dependent on the nature of the data sets. Consider, for example, two identical trivial lists

$$a_i \equiv b_i \equiv i, \; 1 \leq i \leq N \tag{9.16}$$

Figure 9.12: Times per loop (i.e., $N[Z^*]$ increment) for the last several loops in the solution of the 150 × 150 problem.

with the distance between items given by the absolute value function. For the data sets in Equation 9.16, the preliminaries and Step 1 of Munkres algorithm completely solve the association in a time which scales as N^2. In contrast, the random point association problem is a much greater challenge for the algorithm, as nominal pairings indicated by the initial nearest-neighbor searches of the preliminary step are tediously undone in the creation of the staircase-like sequence of zeros needed for Step 4. As a brief, instructive illustration of nature of this processing, Figure 9.12 plots the CPU time *Per Step* for the last passes through the outer loop of Figure 9.10 for the 150×150 assignment problem (recall that each pass through the outer loop increases the Z^* count by one). The processing load per step is seen to be highly non-uniform.

9.9.3 The Concurrent Algorithm

The timing results from Figure 9.11 clearly dictate the manner in which the calculations in Munkres algorithm should be distributes among the nodes of a hypercube for concurrent execution. The zero and minimum element searches for Step 3 and Step 5 are the most time consuming and should be done concurrently. In contrast, the essentially bookkeeping tasks associated with Step 2 and Step 4 require insignificant CPU time and are most naturally done in lockstep (i.e., all nodes of the hypercube perform the same calculations on the same data at the same time). The details of the concurrent algorithm are as follows.

Data Decomposition

The distance matrix $\{d_{ij}\}$ is distributed across the nodes of the hypercube, with entire columns assigned to individual nodes. (This assumes, effectively, that $N_{COLS} \gg N_{NODES}$, which is always the case for assignment problems which are big enough to be 'interesting'.) The cover and zero locator lists defined in Section 9.9.2 are duplicated on all nodes.

Task Decomposition

The concurrent implementation of Step 5 is particularly trivial. Each node first finds its own minimum uncovered value, setting this value to some 'infinite' token if all columns assigned to the node are covered. A simple loop on communication channels determines the global minimum among the node-by-node minimum values, and each node then modifies the contents of its local portion of the distance matrix according to Steps(5.2,5.3).

The concurrent implementation of Step 3 is just slightly more awkward. On entry to Step 3, each node searches for zeros according to the rules of Section 9.9.2, and fills a three-element status list:

$$L[j] \equiv L[\text{Node}_j] \equiv \{S, k_{ROW}, k_{COL}\} \tag{9.17}$$

where S is a zero-search status flag,

$$S \equiv \begin{cases} -1 & \text{No } Z \text{ was found} \\ 0 & Z \text{ with } Z^* \text{ in row (Boring)} \\ 1 & Z \text{ without } Z^* \text{ (Interesting)} \end{cases} \tag{9.18}$$

If the status is non-negative, the last two entries in the status list specify the location of the found zero. A simple channel loop is used to collect the individual status lists of each node into all nodes, and the action taken next by the program is as follows:

- If all nodes give negative status (no Z found), all nodes proceed to Step 5.
- If any node gives status one, all nodes proceed to Step 4 for lockstep updates of the zero location lists, using the row-column indices of the node which gave status one as the starting point for Step 4.1. If more than one node returns status one (highly unlikely, in practice), only the first such node (lower node number) is used.

- If all zeros uncovered are 'Boring', the cover-switching in Step 3.3 of the algorithm is performed. This is done in lockstep, processing the Z's returned by the nodes in order of increasing node number. Note that the cover rearrangements performed for one node may well cover a Z returned by a node with higher node number. In such cases, the nominal Z returned by the later node is simply ignored.

It is worth emphasizing that only the actual *searches* for zero and minimum entries in Step 3 and Step 5 are done concurrently. The updates of the cover and zero locator lists are done in unison.

The concurrent algorithm has been implemented on the Mark III hypercube, and has been tested against random point association tasks for a variety of list sizes. Before examining results of these tests, however, it is worth noting that the concurrent implementation is not particularly dependent on the hypercube topology. The only communication-dependent parts of the algorithm are

1. Determination of the ensemble-wide minimum value for Step 5.
2. Collection of the local Step 3 status lists (Equation 9.18).

either of which could be easily done for almost any MIMD architecture.

Table 9.7 presents performance results for the association of random lists of 200 points on the Mark III hypercube for various cube dimensions. (For consistency, of course, the same input lists are used for all runs.) Time values are given in CPU seconds for the total execution time, as well as the time spent in Step 3 and Step 5. Also given are the standard concurrent execution efficiencies,

$$\epsilon_N \equiv \frac{T[\text{ 1 Node }]}{N \times T[\text{ } N \text{ Nodes }]} \tag{9.19}$$

as well as the numbers of times the Step 3 box of Figure 9.10 is entered during execution of the algorithm. The numbers of entries into the other boxes of Figure 9.10 are independent of the hypercube dimension.

There is an aspect of the timing results in Table 9.7 which should be noted. Namely, essentially *all* inefficiencies of the concurrent algorithm are associated with Step 3 for two Nodes compared to Step 3 for one Node. The times spent in Step 5 are approximately halved for each increase in the dimension of the hypercube. However, the efficiencies associated with the zero searching in Step 3 are rather poorer, particularly for larger numbers of nodes.

N[Nodes]	1	2	4	8
T[Total]	654.83	372.70	205.48	119.25
T[Step 3]	183.80	128.04	81.59	56.66
T[Step 5]	462.06	237.54	117.39	57.94
ϵTotal	...	0.878	0.800	0.686
ϵStep 3	...	0.718	0.563	0.405
ϵStep 5	...	0.973	0.984	0.997
N[Step 3]	7075	4837	3483	2778

Table 9.7: Concurrent Performance for 200 x 200 Random Points

At a simple, qualitative level, the inefficiencies associated with Step 3 are readily understood. Consider the task of finding a single zero located somewhere inside an $N \times N$ matrix. The mean sequential search time is

$$\langle T_{\text{Search}}[1 \text{ Node}] \rangle \propto \frac{N \times N}{2} \tag{9.20}$$

since, on average, half of the entries of the matrix will be examined before the zero is found. Now consider the same zero search on two nodes. The node which has the half of the matrix containing the zero will find it in about half the time of Equation 9.20. *However*, the other node will *always* search through all of its $N \times N/2$ items before returning a null status for Equation 9.18. Since the node which found the zero must wait for the other node before the (lockstep) modifications of zero locators and cover tags, the node without the zero determines the actual time spent in Step 3, so that

$$\langle T_{\text{Search}}[\ 2 \text{ Nodes }] \rangle \approx \langle T_{\text{Search}}[\ 1 \text{ Node }] \rangle \tag{9.21}$$

In the full program, the concurrent bottleneck is not as bad as Equation9.21 would imply. As noted above, the concurrent algorithm can process multiple 'Boring' Z's in a single pass through Step 3. The frequency of such multiple Z's per step can be estimated by noting the decreasing number of times Step 3 is entered with increasing hypercube dimension, as indicated in Table 9.7. Moreover, each node maintains a counter of the last column searched during Step 3. On subsequent re-entries, columns prior to this marked column are searched for zeros only if they have had their cover

N[Nodes]	1	2	4	8
T[Total]	68.08	38.79	23.11	16.40
T[Step 3]	19.63	13.09	9.69	8.00
T[Step 5]	44.99	22.99	11.79	6.16
ϵTotal	...	0.878	0.736	0.519
ϵStep 3	...	0.750	0.506	0.307
ϵStep 5	...	0.978	0.954	0.913
N[Step 3]	2029	1430	1134	991

Table 9.8: Concurrent Performance for 100 x 100 Random Points

tag changed during the prior Step 3 processing. While each of these algorithm elements does diminish the problems associated with Equation 9.21, the fact remains that the search for zero entries in the distributed distance matrix is the least efficient step in concurrent implementations of Munkres algorithm.

The results presented in Table 9.7 demonstrate that an efficient implementation of Munkres algorithm is certainly feasible. It is next interesting to examine how these efficiencies change as the problem size is varied.

The results shown in Table 9.8 and Table 9.9 demonstrate an improvement of concurrent efficiencies with increasing problem size—the expected result. For the 100 x 100 problem on eight nodes, the efficiency is only about 50%. This problem is too small for eight nodes, with only 12 or 13 columns of the distance matrix assigned to individual nodes.

While the performance results in Tables 9.7–9.9 are certainly acceptable, it is nonetheless interesting to investigate possible improvements of efficiency for the zero searches in Step 3. The obvious candidate for an algorithm modification is some sort of checkpointing: at intermediate times during the zero search, the nodes exchange a 'Zero Found Yet?' status flag, with all nodes breaking out of the zero search loop if any node returns a positive result.

For message passing machines such as the Mark III, the checkpointing scheme is of little value, as the time spent in individual entries to Step 3 are not enormous compared to the node-to-node communication time. For example, for the two-node solution of the 300 x 300 problem, the mean time for a single entry to Step 3 is only about 46 msec, compared to a typical

N[Nodes]	1	2	4	8
T[Total]	2046.91	1154.27	622.53	353.30
T[Step 3]	585.61	399.41	235.49	154.57
T[Step 5]	1442.22	742.90	377.89	188.59
ϵTotal	...	0.887	0.822	0.728
ϵStep 3	...	0.733	0.621	0.473
ϵStep 5	...	0.971	0.954	0.956
N[Step 3]	13250	8583	5785	4365

Table 9.9: Concurrent Performance for 300 x 300 Random Points

node-to-node communications time which can be a significant fraction of a millisecond. The time required to perform a single Step 3 calculation is not large compared to node-to-node communications. As a (not unexpected) consequence, all attempts to improve the Step 3 efficiencies through various 'Found Anything?' schemes were completely unsuccessful.

The checkpointing difficulties for a message-passing machine could disappear, of course, on a shared memory machine. If the zero-search status flags for the various nodes could be kept in memory locations readily (i.e., rapidly) accessible to all nodes, the problems of the preceding paragraph might be eliminated. It would be interesting to determine whether significant improvements on the (already good) efficiencies of the concurrent Munkres algorithm could be achieved on a shared memory machine.

9.10 Optimization Methods for Neural Nets: Automatic Parameter Tuning and Faster Convergence

Computers and standard programming languages can be used efficiently for solving high-level, clearly-formulated problems, like computing balance sheets and income statements, solving partial differential equations or managing operations in a car factory. It is much more difficult to write efficient and fault-tolerant programs for "simple" primitive tasks like hearing, seeing, touching, manipulating parts, recognizing faces, avoiding obstacles, etc. Usually, the existing artificial systems for the above tasks are of a narrowly

Figure 9.13: Multilayer perceptron and transfer function

limited domain of application, very sensitive to hardware and software failures, difficult to modify and adapt to new environments.

Neural nets represent a new approach to bridge the gap between cheap computational power and solutions for some of the above cited tasks. We, as human beings, like to cite ourselves as a good example of the power of the neuronal approach to problem solving.

To avoid naive optimism and over-inflated expectations about "self-programming" computers, it is safer to see this development as the creation of another level of tools insulating generic users looking for fast solutions from the details of sophisticated learning mechanisms. Today generic users do not care about writing operating systems, in the near future *some* users will not care about programming and debugging. They will have to choose appropriate off-the-shelf subsystems (both hardware and software) and an appropriate set of examples and high-level specifications, neural nets will do the rest. Neural networks have already been useful in areas like pattern classification, robotics, system modeling and forecasting over time ([Borsellino:61a], [Broomhead:88a], [Gorman:88a], [Seinowski:87a], [Rumelhart:86b], [Lapedes:87a]).

The focus of this work is on "supervised learning", *i.e.*, learning an association between input and output patterns from a set of examples. The mapping is executed by a feed-forward network with different layers of units, such as the one shown in Figure 9.13.

Each unit that is not in the input layer receives an input given by a weighted sum of the outputs of the previous layer and produces an output using a "sigmoidal" transfer function, with a linear range and saturation

for large positive and negative inputs. This particular architecture has been considered because it has been used extensively in neural network research, but the learning method presented can be used for different network designs (see [Broomhead:88a]).

9.10.1 Deficiencies of Steepest Descent

The multi-layer perceptron, initialized with random weights, presents random output patterns. We would like to execute a learning stage, progressively modifying the values of the connection strengths in order to make the outputs nearer to the prescribed ones.

It is straightforward to transform the learning task into an optimization problem (*i.e.*, a search for the minimum of a specified function, henceforth called *energy*). If the *energy* is defined as the sum of the squared errors between *obtained* and *desired* output pattern over the set of examples, minimizing it will accomplish the task.

A large fraction of the theoretical and applied research in supervised learning is based on the *steepest descent* method for minimization. The negative gradient of the *energy* with respect to the weights is calculated during each iteration, and a step is taken in that direction (if the step is small enough, energy reduction is assured). In this way, one obtains a sequence of weight vectors, $\mathbf{w}_n$, that converges to a local minimum of the energy function:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \epsilon \, \mathbf{grad}\, E(\mathbf{w}_n)$$

Now, given that we are interested in converging to the local minimum in the shortest time (this is not always the case: to combat noise some slowness may be desired), there is no good reason to restrict ourselves to *steepest descent*, and there are at least a couple of reasons in favor of other methods. Firstly, the *learning speed*, ϵ, is a free parameter that has to be chosen carefully for each problem (if it is too small progress is slow, if it is too large oscillations may be created), secondly, even in the optimal case of a step along the steepest descent direction bringing the system to the absolute minimum (*along this direction*), it can be proved that steepest descent can be arbitrarily slow, particularly when "the search space contains long ravines that are characterized by sharp curvature across the ravine and a gently sloping floor" [Rumelhart:86b]. In other words, if we are unlucky with the choice of units along the different dimensions (and this is a frequent event

Figure 9.14: Gradient direction for different choice of units

when the number of weights is 1000 or 10,000), it may be the case that during each iteration the previous error is reduced by 0.000000001%!

The problem is essentially caused by the fact that the gradient does not necessarily point in the direction of the minimum, as it is shown in Figure 9.14.

If the *energy* is quadratic, a large ratio of the maximum to minimum eigenvalues cause the "zigzagging" motion illustrated. In the next sections, we will illustrate two suggestions for tuning parameters in an adaptive way and for selecting better descent directions.

9.10.2 The "Bold Driver" Network

There are *no general prescription* for selecting an appropriate learning rate ϵ in back-propagation, in order to avoid oscillations and converge to a good local minimum of the *energy* in a short time. In many applications, it is chosen employing some kind of "black magic", or trial-and-error process. In addition, usually no *fixed* learning rate is appropriate for the entire learning session.

Both problems can be solved *adapting* the learning rate to the local structure of the energy surface.

We start with a given learning rate (the value does not matter) and monitor the *energy* after each weight update. If the *energy* decreases, the learning rate for the next iteration is increased by a factor ρ. On the contrary, if the *energy* increases (an "accident" during learning), this is taken as an indication that the step made was too long, the learning rate is decreased

Figure 9.15: Learning rate magnitude as a function of the iteration number for a test problem

by a factor σ, the last change is cancelled and a new trial is done. The process of reduction is repeated until a step that decreases the energy value is found (this will inevitably happen because the search direction is that of the negative gradient). An example for the size of the learning rate as a function of the iteration number is shown in Figure 9.15.

The name "Bold Driver" was selected for the analogy with the learning process of young and inexperienced car drivers.

By using this "brutally heuristic" method, learning converges in a time that is comparable to, and usually better than that of standard (batch) back-propagation with an optimal and *fixed* learning rate. The important difference is that the time-consuming *meta-optimization* phase for choosing ϵ is avoided. The values for ρ and σ can be fixed once and for all (for example $\rho = 1.1, \sigma = 0.5$) and performance does not depend critically on their choice.

9.10.3 The Broyden-Fletcher-Goldfarb-Shanno One-Step Memoryless Quasi-Newton Method

Steepest descent suffers from a bad reputation by researchers in optimization. From the literature (for example [Gill:81a]), we found a wide selection of more appropriate optimization techniques. Following the "decision tree" and considering the characteristics of large supervised learning problems (large memory requirements and time-consuming calculations of the energy and the gradient), the Broyden-Fletcher-Goldfarb-Shanno one-step memoryless quasi-Newton method (all adjectives are necessary to define it) is a

good candidate in the competition and performed very efficiently on different problems.

Let's define the following vectors: $\mathbf{g}_n = \mathbf{grad}\ E(\mathbf{w}_n)$, $\mathbf{p}_n = \mathbf{w}_n - \mathbf{w}_{n-1}$ and $\mathbf{y}_n = \mathbf{g}_n - \mathbf{g}_{n-1}$. The one-dimensional search direction for the n'th iteration is a modification of the gradient $\mathbf{g}_n$, as follows:

$$\mathbf{d}_n = -\mathbf{g}_n + A_n \mathbf{p}_n + B_n \mathbf{y}_n$$

Every N steps (N being the number of weights in the network) the search is restarted in the direction of the negative gradient.

The coefficients A_n and B_n are combinations of scalar products:

$$A_n = -\left(1 + \frac{\mathbf{y}_n \cdot \mathbf{y}_n}{\mathbf{p}_n \cdot \mathbf{y}_n}\right) \frac{\mathbf{p}_n \cdot \mathbf{g}_n}{\mathbf{p}_n \cdot \mathbf{y}_n} + \frac{\mathbf{y}_n \cdot \mathbf{g}_n}{\mathbf{p}_n \cdot \mathbf{y}_n} \; ; \quad B_n = \frac{\mathbf{p}_n \cdot \mathbf{g}_n}{\mathbf{p}_n \cdot \mathbf{y}_n}$$

The one-dimensional minimization used in this work is based on quadratic interpolation and tuned to back-propagation where, in a single step, both the energy value and the negative gradient can be efficiently obtained. Details on this step are contained in [Williams:87b].

The computation during each step requires $O(N)$ operations (the same behavior as standard batch back-propagation), while the CPU time for each step increases by an average factor of three for the problems considered. Because the total number of steps for convergence is much smaller, we measured a large net benefit in computing time.

Last but not least, this method can be efficiently implemented on MIMD parallel computers.

9.10.4 Parallel Optimization

Neural nets are "by definition" parallel computing systems of many densely-interconnected units. Parallel computation is the basic method used by our brain to achieve response times of hundreds of milliseconds, using sloppy biological hardware with computing times of some milliseconds per basic operation.

Our implementation of the learning algorithm is based on the use of MIMD machines with large grain-size. An efficient mapping strategy consists in assigning a subset of the examples (input-output pairs) and the *entire* network structure to each processor. To obtain proper generalization, the number of example patterns has to be much larger (say $\geq 10\,X$) than the

number of parameters defining the architecture (*i.e.*, the number of connection weights). For this reason, the amount of memory used for storing the weights is not too large for significant problems.

Function and gradient evaluation is executed in parallel. Each processor calculates the contribution of the assigned patterns (with no communication) and a global combining-distributing step (see the *ADDVEC* routine in [Fox:88a]) calculates the total energy and gradient (let's remember that the energy is a *sum* of the patterns' contributions) and communicates the result to all processors.

Then the one-dimensional minimization along the search direction is completed and the weights are updated.

This simple parallelization approach is promising, it can be easily adapted to different network representations and learning strategies, and it is going to be a fierce competitor with analog implementations of neural networks, when these will be available for significant applications (let's remember that airplanes do not flap their wings ...).

9.10.5 Experiment: The Dichotomy Problem

This problem consists in classifying a set of randomly generated patterns (with real values) in two classes. An example in two dimensions is given by the "healthy food" learning problem. Inputs are given by points in the "smell" and "taste" plane, corresponding to the different foods. The learning task consists in producing the correct classification as "healthy food" or "junk food".

On this problem we obtained a speedup of 20–120 (going from six to 100 patterns in two dimensions).

9.10.6 Experiment: Time Series Prediction

In this case, the task is to predict the next value in the sequence (ergodic and chaotic) generated by the logistic map [Lapedes:87a], according to the recurrence relation:

$$x_{n+1} = 4\ x_n\ (1 - x_n)$$

We tried different architecture and obtained a speedup of 400–500, and slightly better generalization properties for the BFGS optimization method presented.

Figure 9.16: "Healthy food" has to be distinguished from "junk food" using taste and smell information.

In general, we obtained a larger speedup for problems with high precision requirements (using real values for inputs or outputs). See also [Battiti:89a].

9.10.7 Credits and C^3P References

The distributed optimization software was developed by Roberto Battiti, modifying a back-propagation program written by Steve Otto. Professor Geoffrey Fox and Roy Williams inspired our first investigations into the optimization literature (Shanno's conjugate gradient [Shanno:78a] is used in [Williams:87b]).

C^3P References: [Battiti:89a], [Battiti:89e]

Chapter 10

DIME Programming Environment

10.1 DIME: Portable Software for Irregular Meshes for Parallel or Sequential Computers

A large fraction of the problems that we wish to solve with a computer are continuum simulations of physical systems, where the interesting variable is not a finite collection of numbers but a function on a domain. For the purposes of the computation such a continuous spatial domain is given a structure, or mesh, to which field values may be attached and neighboring values compared to calculate derivatives of the field.

If the domain of interest has a simple shape, such as a cylinder or cuboid, there may be a natural choice of mesh whose structure is very regular like that of a crystal, but when we come to more complex geometries such as the space surrounding an aircraft or inside turbomachinery, there are no regular structures that can adequately represent the domain. The only way to mesh such complex domains is with a unstructured mesh, such as that shown in Figure 10.1. At the right is a plot of a solution to Laplace's equation on the domain.

Notice that the mesh is especially fine at the sharp corners of the boundary where the solution changes rapidly. A desirable feature for a mesh is the ability for it to adapt, so that when the solution begins to emerge, the mesh may be made finer where necessary.

Figure 10.1: Mesh and Solution of Laplace Equation

Naturally, we would like to run our time-consuming physical simulation with the most cost-effective general purpose computer, which we believe to be the MIMD architecture. In view of the difficulty of programming an irregular structure such as one of these meshes, and the special difficulty of doing so with an MIMD machine, I decided to write not just a program for a specialized application, but a *programming environment* for unstructured triangular meshes.

The resulting software (DIME: Distributed Irregular Mesh Environment, [Williams:90b]) is responsible for the mesh structure, and a separate application code runs a particular type of simulation on the mesh. DIME keeps track of the mesh structure, allowing mesh creation, reading and writing meshes to disk, and graphics; also adaptive refinement and certain topological changes to the mesh; it hides the parallelism from the application code, and splits the mesh among the processors in an efficient way.

The application code is responsible for attaching data to the elements and nodes of the mesh, manipulating and computing with these data and the data from its mesh-neighborhood. DIME is designed to be portable, not only between different MIMD parallel machines, but it also runs on any Unix machine, treating this as a parallel machine with just one processor. This ability to run on a sequential machine is due to DIME's use of the Cubix server (Sec. 5.2).

10.1.1 Applications and Extensions

The most efficient speed for aircraft flight is just below the speed of sound: the transonic regime. Simulations of flight at these speeds consume large quantities of computer time, and are a natural candidate for a DIME application. In addition to the complex geometries of airfoils and turbines for which these simulations are required, the flow tends to develop singular regions or shocks in places that cannot be predicted in advance; the adaptive refinement capability of a DIME mesh allows the mesh to be fine and detail resolved near shocks while keeping the regions of smooth flow coarsely meshed for economy (Section 12.3).

The current version of DIME is only able to mesh two-dimensional manifolds. The manifold may, however, be embedded in a higher-dimensional space. In collaboration with the Biology Division at Caltech, we have simulated the electrosensory system of the weakly electric fish *Apteronotus leptorhynchus*. The simulation involves creating a mesh covering the skin of the fish, and using the Boundary Element Method to calculate field strengths in the three-dimensional space surrounding the fish (Section 12.2).

In the same vein of embedding the mesh in higher dimensions, we have simulated a bosonic string of high-energy physics, embedding the mesh in up to 26 spatial dimensions. The problem here is to integrate over not only all positions of the mesh nodes, but also over all *triangulations* of the mesh (Section 7.2).

The information available to a DIME application is certain data stored in the elements and nodes of the mesh. When doing Finite Element calculations, one would like a somewhat higher level of abstraction, which is to refer to functions defined on a domain, with certain smoothness constraints and boundary conditions. We have made a further software layer on top of DIME to facilitate this: DIMEFEM. With this we may add, multiply, differentiate and integrate functions defined in terms of the Lagrangian Finite Element family, and define linear, bilinear and nonlinear operators acting on these functions. When a bilinear operator is defined, a variational principle may be solved by conjugate-gradient methods. The preconditioner for the CG method may in itself involve solving a variational principle. The DIMEFEM package has been applied to a sophisticated incompressible flow algorithm (Section 10.2).

Figure 10.2: Major Components of DIME

10.1.2 The Components of DIME

Figure 10.2 shows the structure of a DIME application. The shaded parts represent the contribution from the user, being a definition of a domain which is to be meshed, a definition of the data to be maintained at each element, node and boundary edge of the mesh, and a set of functions that manipulate this data. The user may also supply or create a script file for running the code in batch mode.

The first input is the definition of a domain to be meshed. A file may be made using the elementary CAD program `curvetool`, which allows straight lines, arcs and cubic splines to be manipulated interactively to define a domain.

Before sending a domain to a DIME application, it must be predigested to some extent, with the help of a human. The user must produce a coarse mesh that defines the topology of the domain to the machine. This is done with the program `meshtool`, which allows the user to create nodes and connect them up to form a triangulation.

The user writes a program for the mesh, and this program is loaded into each processor of the parallel machine. When the DIME function `readmesh()` is called, (or "Readmesh" clicked on the menu), the mesh created by `meshtool` is read into a single processor, and then the function `balance_orb()` may be called (or "Balance" clicked on the menu) to split the mesh into domains, one domain for each processor.

The user may also call the function `writemesh()` (or click "Writemesh" in the menu), which causes the parallel mesh to be written to disk. If that

Figure 10.3: Boundary Structure

mesh is subsequently read in, it is read in its domain-decomposed form, with different pieces assigned to different processors.

In the Figure, the Cubix server is not mandatory, but only needed if the DIME application is to run in parallel. The application **also runs on a sequential machine**, which is considered to be a one-processor parallel machine.

10.1.3 Domain Definition

Figure 10.3 shows an example of a DIME boundary structure. The filled blobs are *Points*, with *Curves* connecting the points. Each Curve may consist of a set of curve segments, shown in the figure separated by open circles. The curve segments may be straight lines, arcs of circles, or Bezier cubic sections. The program `curvetool` is for the interactive production of boundary files. When the domain is satisfactory, it should be meshed using `meshtool`.

The program `meshtool` is for taking a boundary definition and creating a triangulation of certain regions of it. `Meshtool` adds nodes to an existing triangulation using the Delaunay triangulation [Bowyer:81a]. A new node may be added anywhere except at the position of an existing node. Figure 10.4 illustrates how the Delaunay triangulation (thick gray lines) is derived from the Voronoi tesselation (thin black lines).

Each node (shown by a blob in the figure) has a "territory", or Voronoi polygon, which is the part of the plane closer to the node than to any other node. The divisions between these territories are shown as thin lines in the figure, and are the perpendicular bisectors of the lines between the nodes.

Figure 10.4: Voronoi Tesselation and Resulting Delaunay Triangulation

This procedure tesselates the plane into a set of disjoint polygons and is called the Voronoi tesselation. Joining nodes whose Voronoi polygons have a common border creates a triangulation of the nodes known as the Delaunay triangulation. This triangulation has some desirable properties, such as the diagonal dominance of a Finite Element stiffness matrix derived from the mesh [Young:71a]

10.1.4 Mesh Structure

In Figure 10.5 is shown a triangular mesh covering a rectangle, and in Figure 10.6 the logical structure of that mesh split among four processors. The logical mesh shows the elements as shaded triangles and nodes as blobs. Each element is connected to exactly three nodes, and each node is connected to one or more elements. If a node is at a boundary, it has a boundary structure attached, together with a pointer to the next node clockwise around the boundary.

Each node, element and boundary structure has user data attached to it, which is automatically transferred to another processor if load-balancing causes the node or element to be moved to another processor. DIME knows only the size of the user data structures. Thus, these structures may not contain pointers, since when those data are moved to another processor the pointers will be meaningless.

The shaded ovals in Figure 10.5 are Physical Nodes, each of which consists of one or more Logical Nodes. Each logical node has a set of aliases, which are the other logical nodes belonging to the same physical node. The

Figure 10.5: A Mesh Covering a Rectangle

Figure 10.6: The Logical Structure of the Mesh Split Among Four Processors

physical node is a conceptual object, and is unaffected by parallelism; the logical node is a copy of the data in the physical node, so that each processor which owns a part of that physical node may access the data as if it had the whole node.

DIME is meant to make distributed processing of an unstructured mesh almost as easy as sequential programming. However, there is a remaining "kernel of parallelism" that the user must bear in mind. Suppose each node of the mesh gathers data from its local environment (*i.e.*, the neighboring elements); if that node is split among several processors, it will only gather the data from those elements which lie in the same processor and consequently each node will only have part of the result. We need to combine the partial results from the logical nodes and return the combined result to each. This facility is provided by a macro in DIME called `NODE_COMBINE`, which is called each time the node data is changed according to its local environment.

10.1.5 Refinement

The Delaunay triangulation used by `meshtool` would be an ideal way to refine the working mesh, as well as making a coarse mesh for initial download. Unfortunately, adding a new node to an existing Delaunay triangulation may have global consequences; it is not possible to predict in advance how much of the current mesh should be replaced to accommodate the new node. Doing this in parallel requires an enormous amount of communication to make sure that the processors do not tread on each others toes [Williams:89c].

DIME uses the algorithm of Rivara [Rivara:84a] for refinement of the mesh, which is well suited to loosely synchronous parallel operation, but results in a triangulation which is not a Delaunay triangulation, and thus lacks some desirable properties. The process of topological relaxation changes the connectivity of the mesh to make it a Delaunay triangulation.

It is usually desirable to avoid triangles in the mesh which have particularly acute angles, and topological relaxation will reduce this tendency. Another method to do this is by moving the nodes toward the average position of their neighboring nodes; a physical analogy would be to think of the edges of the mesh as damped springs and allowing the nodes to move under the action of the springs.

Figure 10.7: Recursive Bisection

10.1.6 Load Balancing

When DIME operates in parallel, the mesh should be distributed among the processors of the machine so that each processor has about the same amount of mesh to deal with, and so that the communication time is as small as possible. There are several ways to do this, such as with a computational neural net, by simulated annealing, or even interactively.

DIME uses a strategy known as Recursive Bisection, which has the advantages of being robust, simple, and deterministic, though sometimes the resulting communication pattern may be less than optimal. The method is illustrated in Figure 10.7: each blob represents the center of an element, and the vertical and horizontal lines represent processor divisions. First a median vertical line is found which splits the set of elements into two sets of approximately equal numbers, then (with two-way parallelism) two horizontal medians which split the halves into four approximately equal quarters, then (with four-way parallelism) four vertical medians, and so on.

10.1.7 Credits and C^{3}P References

The DIME software was written by Roy Williams.

C^{3}P References: [Baillie:89i], [Williams:87a], [Williams:87b], [Williams:88a], [Williams:88d], [Williams:89c], [Williams:90b].

10.2 DIMEFEM: High-level Portable Irregular-Mesh Finite-Element Solver

DIMEFEM [Williams:89a] is a software layer which enables finite element calculations to be done with the irregular mesh maintained by DIME. The data objects dealt with by DIMEFEM are Finite Element Functions (FEFs) which may be scalar or have several components (vector fields), and also linear, multilinear and nonlinear operators which map these FEFs to numbers. The guiding principle is that interesting physical problems may be expressed in variational terms involving FEFs and operators on them [Bristeau:87a], [Glowinski:84a]. We shall use as an example a Poisson solver.

Poisson's equation is $\nabla^2 u = f$, which may also be expressed variationally as: find u such that for all v

$$a(u,v) \equiv \int \nabla u \cdot \nabla v \, d\Omega = \int f v \, d\Omega \equiv L(v) \tag{10.1}$$

where the unknown u and the dummy variable v are taken to have the correct boundary conditions. To implement this with DIMEFEM, we first allocate space in each element for the FEFs u and f, then explicitly set f to the desired function. We now define the linear operator L and bi-linear operator a as above, and call the linear solver to evaluate u.

10.2.1 Memory Allocation

When DIME creates a new element by refinement, it comes equipped with a pointer to a block of memory of user-specified size which DIMEFEM uses to store the data representing FEFs and corresponding linear operators. A template is kept of this element memory containing information about which is already allocated and which is free. When a FEF is to be created the memory allocator is called, which decides how much memory is needed per element, and returns an offset from the start of the element-data-space for storing the new FEF. Thus, a function in DIMEFEM typically consists of allocating a stack of work space, doing calculations, then freeing the work space.

An FEF thus consists of a specification of an element type, a number of fields (one for scalar, two or more for vector), and an offset into the element data for the nodal values.

10.2.2 Operations and Elements

Finite element approximations to functions form a finite-dimensional vector space, and as such may be multiplied by a scalar and added, and functions are provided to do these operations. If the function is expressed as Lagrangian elements it may also be differentiated, which changes the order of representation, for example differentiating a quadratic element produces a linear element.

At present DIMEFEM provides two kinds of element, Lagrangian and Gaussian, although strictly speaking the latter is not a Finite Element because it possesses no interpolation functions. The Gaussian element is simply a collection of function values at points within each triangle and a set of weights, so that integrals may be done by summing the function values multiplied by the weights. As with one-dimensional Gaussian integration, integrals are exact up to some polynomial order. We cannot differentiate Gaussian FEF's, but can apply pointwise operators such as multiplication and function evaluation that cannot be done in the Lagrangian representation.

Consider the nonlinear operator L defined by

$$L(u) = \int \exp[du/dx] d\Omega \tag{10.2}$$

The most accurate way to evaluate this is to start with u in Lagrangian form, differentiate, convert to Gaussian representation, exponentiate, then multiply by the weights and sum. This can be done explicitly with DIMEFEM, but in the future we hope to create an environment which 'knows' about representations, linearity, etc., and can parse an expression such as the above and evaluate it correctly.

The computational kernel of any finite element software is the linear solver. We have implemented this with preconditioned conjugate gradient, so that the user supplies a linear operator L, an elliptic bilinear operator a, and a scalar product S (a strongly elliptic symmetric bilinear operator which satisfies the triangle inequality), and an initial guess for the solution. The conjugate gradient solver replaces the guess by the solution u of the standard variational equation

$$a(u, v) = L(v) \qquad \forall v \tag{10.3}$$

using the preconditioner S.

10.2.3 Navier-Stokes Solver

We have implemented a sophisticated incompressible flow solver using DIME and DIMEFEM. The algorithm is described more completely in [Bristeau:87a]. The evolution equation for an incompressible Newtonian fluid of viscosity n is

$$du/dt + \nu\nabla^2 u + (u \cdot \nabla)\, u + \nabla p = f\nabla \cdot u = 0 \tag{10.4}$$

We use a three-stage operator-split scheme whereby for each timestep of length dt, the equation is integrated

- from t to $t + \vartheta dt$ with incompressibility and no convection, then
- from $t + \vartheta dt$ to $t + (1 - \vartheta)dt$ with convection and no incompressibility condition, then
- to $t + dt$ as in stage one with incompressibility and no convection.

The parameter ϑ is $1 - 1/\sqrt{2} \approx 0.29$.

Each of these three implicit steps involves the solution of either a Stokes problem:

$$\alpha u - \nu\nabla^2 u + \nabla p = f\nabla \cdot u = 0 \tag{10.5}$$

or the nonlinear problem:

$$\alpha u - \nu\nabla^2 u + (u \cdot \nabla)u = f \tag{10.6}$$

where α is a parameter inversely proportional to the timestep. We solve the Navier-Stokes equation, and consequently also these subsidiary problems, with given velocity at the boundary (Dirichlet boundary conditions).

10.2.4 Results

With velocity approximated by quadratic, and pressure by linear Lagrangian elements, we found that both the Stokes and nonlinear solvers converged in three to five iterations. We ran the square cavity problem as a benchmark.

To reach a steady state solution, we adopted the following strategy; with a coarse mesh keep advancing simulation time until the velocity field no longer changes, then refine the mesh, iterate until the velocity stabilizes, refine, and so on. The refinement strategy was as follows. The velocity is approximated with quadratic elements with discontinuous derivatives, so we

Figure 10.8: Results for Square Cavity Problem with Reynolds Number 1000

can calculate the maximum of this derivative discontinuity for each element, then refine those elements above the 70th percentile of this quantity.

Figure 10.8 shows the results. At top left is the mesh after four cycles of this refinement and convergence, at Reynolds number 1000. We note heavy refinement at the top left and top right, where the boundary conditions force a discontinuity in velocity, and also along the right side where the near discontinuous vorticity field is being transported around the primary vortex. Bottom left shows the logical structure, split among four transputers. The top right and bottom right show streamlines and vorticity respectively. The results accord well with the benchmark of [Schreiber:83a].

10.2.5 Credits and C^{3}P References

The DIMEFEM software was written by Roy Williams, and the flow algorithm developed by R. Glowinski of the University of Houston.

C^{3}P References: [Williams:89a], [Williams:90b].

Chapter 11

Load Balancing

11.1 Physical Computation

*** Contribution needed from G. C. Fox

11.2 Load Balancing and Examples

11.2.1 Load Balancing for Finite Element Meshes

The challenge of parallel computing is to split a problem into pieces which may be executed concurrently. If the problem can be split into many pieces rather than few, then more processors may be used and greater speedup achieved. For example, a climate modeling program may treat the atmosphere and ocean separately, with each being represented by a large number of grid points. A simple split, appropriate for two processors, would be to process the atmosphere and ocean separately. Unfortunately it is then not possible to use more than two processors; a better split would be to treat each grid point as a separate entity, enabling a large number of processors to be used.

Thus, we wish to split the problem into as many smalltasks as possible from the start, then assign each processor some of these tasks, hopefully optimizing the efficiency, which in turn depends on how the tasks communicate with each other. Aspects of the communication pattern include such factors as

- Do the tasks communicate at all?

- Is there a fixed set of paths along which messages pass repeatedly or do the messages move in a different and unpredictable way each time there is a communication?

- Does each task communicate with most others or only with a small number of 'neighbors'?

If the tasks do not communicate, a simple and effective way to do the assignment is by having one processor, or host, distribute tasks from a queue, and each processor request further work from the host when it is finished. This is analogous to a queue of customers waiting for the next available teller in a bank, and keeps each teller busy even though some customers take longer than others. Notice that this method is not scaleable to very large numbers of processors, since there is a sequential bottleneck at the point where tasks are distributed.

In this section, we examine load balancing strategies for a computation where each task communicates regularly with a small fixed set of other tasks for many cycles. With this repeated fixed communication pattern, it is worthwhile doing a good job of the assignment problem, since a small change in efficiency may make a large gain in the total time taken by the calculation.

The primary example of such a set of communicating tasks is a finite element mesh such as that shown in Figure 11.1. Each triangle or element represents a task, which communicates with its neighboring three triangles. In doing, for example, a simulation of fluid flow on the mesh, each element of the mesh communicates regularly with its neighbors, and this pattern may be repeated thousands of times.

We may classify load balancing strategies into four broad types, depending on when the optimization is made and whether the cost of the optimization is included in the optimization itself:

- **By Inspection:** The load balancing strategy may be determined by inspection, such as with a rectangular lattice of grid points split into smaller rectangles, so that the load balancing problem is solved before the program is written.

- **Static:** The optimization is non-trivial, but may be done by a sequential machine before starting the parallel program, so that the load balancing problem is solved before the parallel program begins.

- **Quasi-Dynamic:** The circumstances determining the optimal balance change during program execution, but discretely and infrequently.

Figure 11.1: An unstructured triangular mesh surrounding a four-element airfoil. The mesh is distributed among 16 processors, with divisions shown by heavy lines.

Because the change is discrete, the load balance problem, and hence its solution, remain the same until the next change. If these changes are infrequent enough, any savings made in the subsequent computation make up for the time spent solving the load balancing problem. The difference between this and the static case is that the load balancing must be carried out in parallel to prevent a sequential bottleneck.

- **Dynamic:** The circumstances determining the optimal balance change frequently or continuously during execution, so that the cost of the load balancing calculation after each change should be minimized in addition to optimizing the splitting of the actual calculation. This means that there must be a decision made every so often to decide if load balancing is necessary, and how much time to spend on it.

If the mesh is solution-adaptive, *i.e.*, the mesh, and hence the load balancing problem, change discretely during execution of the code, then it is most efficient to decide the optimal mesh distribution in parallel. In this section, three parallel algorithms, Orthogonal Recursive Bisection (ORB), Eigenvector Recursive Bisection (ERB) and a simple parallelization of Simulated Annealing (SA) have been implemented for load balancing a dynamic unstructured triangular mesh on 16 processors of an NCUBE machine.

The test problem is a solution-adaptive Laplace solver, with an initial mesh of 280 elements, refined in seven stages to 5,772 elements. We present execution times for the solver resulting from the mesh distributions using

the three algorithms, as well as results on imbalance, communication traffic and element migration.

In this section, we shall consider the quasi-dynamic case with observations on the time taken to do the load balancing that bear on the dynamic case. The testbed is an unstructured-mesh finite element code, where the elements are the atoms of the problem, which are to be assigned to processors. The mesh is solution-adaptive, meaning that it becomes finer in places where the solution of the problem dictates refinement.

We shall show that a class of finite element applications share common load balancing requirements, and formulate load balancing as a graph coloring problem. We shall discuss three methods for solving this graph coloring problem, one based on statistical physics, one derived from a computational neural net, and one cheap and simple method.

We present results from running these three load balancing methods, both in terms of the quality of the graph coloring solution (machine-independent results), and in terms of the particular machine (16 processors of an NCUBE) on which the test was run.

11.2.2 Load Balancing a Finite Element Mesh

An important class of problems are those which model a continuum system by discretizing continuous space with a mesh. Figure 11.1 shows an unstructured triangular mesh surrounding a cross-section of a four-element airfoil from an Airbus A-310. The variations in mesh density are caused by the nature of the calculation for which the mesh has been used; the airfoil is flying at Mach 0.8 to the left, so that a vertical shock extends upward at the trailing edge of the main airfoil, which is reflected in the increased mesh density.

The mesh has been split among 16 processors of a distributed machine, with the divisions between processors shown by heavy lines. Although the areas of the processor domains are different, the numbers of triangles or elements assigned to the processors are essentially the same. Since the work done by a processor in this case is the same for each triangle, the workloads for the processors are the same. In addition, the elements have been assigned to processors so that the number of adjacent elements which are in different processors is minimized.

In order to analyze the optimal distribution of elements among the processors, we must consider the way the processors need to exchange data during a calculation. In order to design a general load balancer for such

calculations, we would like to specify this behavior with the fewest possible parameters, which do not depend on the particular mesh being distributed. The following remarks apply to several application codes which have been written to run with the DIME software (Section 10.1) which use two-dimensional unstructured meshes, these being:

- **Laplace:** A scalar Laplace solver with linear finite elements, using Jacobi relaxation;
- **Wing:** A finite-volume transonic Euler solver, with harmonic and biharmonic artificial dissipation [Williams:89b];
- **Convect:** A simple finite-volume solver which convects a scalar field with uniform velocity, with no dissipation;
- **Stress:** A plane-strain elasticity solver with linear finite elements using conjugate gradient to solve the stiffness matrix;
- **Fluid:** An incompressible flow solver with quadratic elements for velocity and linear elements for pressure, using conjugate gradient to solve the Stokes problem and a nonlinear least-squares technique for the convection [Williams:89a].

As far as load balancing is concerned, each of these codes are rather similar, which is because the algorithms used are local: each element or node of the mesh gets data from its neighboring elements or nodes. In addition, a small amount of global data is needed; for example when solving iteratively, each processor calculates the norm of the residual over its part of the mesh, and all the processors need the minimum value of this to decide if the solve has converged.

The computational kernel of each of these applications is iterative, and each iteration may be characterized by three numbers:

- The number of floating point operations during the iteration, which is proportional to the number of elements owned by the processor;
- The number of global combining operations during the iteration.
- The number and size of local communication events, in which the elements at the boundary of the processor region communicate data loosely synchronously [Fox:88a] with their neighboring elements in other processors, which is proportional to the number of elements at the boundary of the processor domain.

These numbers are listed in the following table for the five applications listed above:

Application	Algorithm	Flops per element	Global combines	Local communications per boundary element	c
Laplace	Jacobi relaxation	12	1	1	12
Wing	Finite volume	742		40	19
Convect	Finite volume	100		5	20
Stress	Conjugate gradient	85	2	2	42
Fluid	Quadratic CG	330	2	4	83

The two finite-volume applications do not have iterative matrix solves, so they have no convergence checking and thus have no need for any global data exchange. The ratio c in the last column is the ratio of the third to the fifth columns and may be construed as follows. Suppose a processor has E elements, of which B are at the processor boundary. Then the amount of communication the processor must do compared to the amount of calculation is given by

$$\frac{\text{floats communicated}}{\text{floats multipled or added}} = \frac{B}{cE}$$

It follows that a large value of c corresponds to an eminently parallelizable operation, since the communication rate is low compared to calculation. The 'Stress' example has a high value of c because the solution being sought is a two-dimensional strain field; while the communication is doubled, the calculation is quadrupled, because the elements of the scalar stiffness matrix are replaced by 2×2 block matrices, and each block requires four multiplies instead of one. For the 'Fluid' example, with quadratic elements, there are the two components of velocity being communicated at both nodes and edges, which is a factor of four for communication, but the local stiffness matrix is now 6×6 because of the quadratic elements. Thus we conclude that the more interacting fields, and the higher the element order, the more efficiently the application runs in parallel.

11.2.3 The Optimization Problem

We wish to distribute the elements among the processors of the machine to minimize both load imbalance (one processor having more elements than another), and communication between elements.

Our approach here is to write down a cost function which is minimized when the total running time of the code is minimized and is reasonably simple and independent of the details of the code. We then minimize this cost function and distribute the elements accordingly.

The load balancing problem [Fox:88a], [Fox:88mm] may be stated as a graph-coloring problem: given an undirected graph of N nodes (finite elements), color these nodes with P colors (processors), to minimize a cost-function H which is related to the time taken to execute the program for a given coloring. For DIME applications, it is the finite elements which are to be distributed among the processors, so the graph to be colored is actually the dual graph to the mesh, where each graph node corresponds to an element of the mesh and has (if it is not at a boundary) three neighbors.

We may construct the cost function as the sum of a part that minimizes load imbalance and a part that minimizes communication:

$$H = H_{calc} + \mu H_{comm}$$

where H_{calc} is the part of the cost-function which is minimized when each processor has equal work, H_{comm} is minimal when communication time is minimized, and μ is a parameter expressing the balance between the two: μ is related to the number c discussed above. If H_{calc} and H_{comm} were proportional to the times taken for calculation and communication, then μ should be inversely proportional to c. For programs with a great deal of calculation compared to communication, μ should be small, and *vice versa.*

As μ is increased, the number of processors in use will decrease until eventually the communication is so costly that the entire calculation must be done on a single processor.

Let e, f, ... label the nodes of the graph, and $p(e)$ be the color (or processor assignment) of graph node e. Then the number of graph nodes of color q is:

$$N_q = \sum_e \delta_{q,p(e)}$$

and H_{calc} is proportional to the maximum value of N_q, because the whole calculation runs at the speed of the slowest processor, and the slowest processor is the one with the most graph nodes. The formulation as a maximum of N_q is, however, not satisfactory when a perturbation is added to the cost function, such as that from the communication cost function. If, for example, we were to add a linear forcing term proportional to N_0, the cost function would be:

$$H_{calc}^{perturbed} = \max N_q + \varepsilon N_0$$

and the minimum of this perturbed cost function is either $N_0 = N_1 = \ldots = N/P$ if ε is less than $1/(P-1)$, or $N_0 = 0$, $N_1 = N_2 = N/(P-1)$ if ε is larger than this. This discontinuous behavior as a result of perturbations is undesirable, so we use a sum of squares instead, whose minima change smoothly with the magnitude of a perturbation:

$$H_{calc} = \zeta \sum_q N_q^2$$

where ζ is a scaling constant to be determined.

We now consider the communication part of the cost function. Let us define the matrix

$$B_{qr} = \sum_{e \leftrightarrow f} 1 - \delta_{q,p(e)}\ \delta_{r,p(f)}$$

which is the amount of communication between processor q and processor r, and the notation $e \leftrightarrow f$ means that the graph nodes e and f are connected by an edge of the graph.

The cost of communication from processor q to processor r depends on the machine architecture; for some parallel machines it may be possible to write down this metric explicitly. For example, with the early hypercubes, the cost is the number of bits which are different in the binary representations of the processor numbers q and r. The metric may also depend on the message-passing software, or even on the activities of other users for a shared machine. A truly portable load balancer would have no option but to send sample messages around and measure the machine metric, then distribute the graph appropriately. In this book, however, we shall avoid the question of the machine metric by simply assuming that all pairs of processors are equally far apart, except of course a processor may communicate with itself at no cost.

The cost of sending the quantity B_{qr} of data also depends on the programming: the cost will be much less if it is possible for the B_{qr} messages to be bundled together and sent as one, rather than sent separately. The problem is latency: the cost to send a message in any distributed system is the sum of an initial fixed price and a price proportional to the size of the message. This is also the case for the pricing of telephone calls, freight shipping, mail service and many other examples from the everyday world. If the message is large enough, we may ignore latency: for the NCUBE used in Section 11.2.8 of this book, latency may be ignored if the message is longer than a hundred bytes or so. In the tests of Section 11.2.8, most of the messages

are indeed long enough to neglect latency, though there is certainly further work needed on load balancing in the presence of this important effect.

The result of this discussion is that we shall assume that the cost of communicating the quantity B_{qr} of data is proportional to B_{qr}, unless $q = r$, in which case the cost is zero.

We shall now make the assumption that the total communication cost is the sum of the individual communications between processors:

$$H_{comm} = \varepsilon \sum_{q \neq r} B_{qr}$$

where ε is a constant to be determined. Notice that any parallelism in communication is ignored. Substituting the expression for B_{qr}, the expression for the load balance cost function simplifies to

$$H = \zeta \sum_{q} N_q^2 + \mu\varepsilon \sum_{e \leftrightarrow f} 1 - \delta_{p(e),p(f)}$$

The assumptions made to derive this cost function are significant. The most serious deviation from reality is neglecting the parallelism of communication, so that a minimum of this cost function may have grossly unbalanced communication loads. This turns out not to be the case, however, because when the mesh is equally balanced, there is a lower limit to the amount of boundary, analogous to a bubble having minimal surface area for fixed volume; if we then minimize the sum of surface areas for a set of bubbles of equal volumes, each surface must be minimized and equal.

We may now choose the scaling constants ζ and ε. A convenient choice is such that the optimal H_{calc} and H_{comm} have contributions of size about one from each processor; the form of the scaling constant ε is because the surface area of a compact shape in d dimensions varies as the $d-1$ power of the size, while volume varies as the d power. The final form for H is

$$H = \frac{P^2}{N^2} \sum_{q} N_q^2 + \mu \left(\frac{P}{N}\right)^{\frac{d-1}{d}} \sum_{e \leftrightarrow f} 1 - \delta_{p(e),p(f)}$$

where d is the dimensionality of the mesh from which the graph came.

11.2.4 Algorithms for Load Balancing

This book presents performance evaluation of three load balancing algorithms, all of which run in parallel. With a massively parallel machine, it

would not be possible to load balance the mesh sequentially. This is because (1) there would be a serious sequential bottleneck, (2) there would not be enough memory in a host machine to store the entire distributed mesh, and (3) the large cost incurred in communicating the entire mesh.

The three methods are:

- **SA**—Simulated Annealing: We directly minimize the above cost function by a process analogous to slow physical cooling.
- **ORB**—Orthogonal Recursive Bisection: A simple method which cuts the graph into two by a vertical cut, then cuts each half into two by a horizontal cut, then each quarter is cut vertically, and so on.
- **ERB**—Eigenvector Recursive Bisection: This method also cuts the graph in two then each half into two, and so on, but the cutting is done using an eigenvector of a matrix with the same sparsity structure as the adjacency matrix of the graph. The method is an approximation to a computational neural net.

11.2.5 Simulated Annealing

Simulated annealing [Fox:88mm], [Hajek:88a], [Kirkpatrick:83a], [Otten:89a] is a very general optimization method which stochastically simulates the slow cooling of a physical system. The idea is that there is a cost function H (in physical terms a Hamiltonian) which associates a cost with a state of the system, a 'temperature' T, and various ways to change the state of the system. The algorithm works by iteratively proposing changes and either accepting or rejecting each change. Having proposed a change we may evaluate the change δH in H. The proposed change may be accepted or rejected by the *Metropolis* criterion; if the cost function decreases ($\delta H < 0$) the change is accepted unconditionally, otherwise it is accepted but only with probability $\exp(-\delta H/T)$. A crucial requirement for the proposed changes is *reachability*: that there be a sufficient variety of changes than there is a sequence of changes so that any system state may be reached from any other.

When the temperature is zero, changes are accepted only if H decreases, an algorithm also known as the *greedy algorithm* or *hill-climbing*. The system soon reaches a state in which none of the proposed changes can decrease the cost function, but this is usually a poor optimum. In real life, we might be trying to achieve the highest point of a mountain range by simply walking upwards; we soon arrive at the peak of a small foothill and can go no further.

Figure 11.2: Simulated annealing of a ring graph of size 200, with the four graph colors shown by gray shades. The time history of the annealing runs vertically, with the maximum temperature and the starting configuration at the bottom; zero temperature and the final optimum at the top. The basic move is to change the color of a graph node to a random color.

On the contrary, if the temperature is very large, all changes are accepted, and we simply move at random ignoring the cost function. Because of the reachability property of the set of changes, we explore all states of the system, including the global optimum.

Simulated annealing consists of running the accept/reject algorithm between the temperature extremes. We propose many changes, starting at a high temperature and exploring the state space, and gradually decreasing the temperature to zero while hopefully settling on the global optimum. It can be shown that if the temperature decreases sufficiently slowly (the inverse of the logarithm of the time), then the probability of being in a global optimum tends to certainty [Hajek:88a].

Figure 11.2 shows simulated annealing applied to the load balancing cost function in one dimension. The graph to be colored is a periodically connected linear array of 200 nodes, to be colored with four colors. The initial configuration, at the bottom of the figure, is the left 100 nodes colored white, two domains of 50 each in mid grays, and with no nodes colored in the darkest gray. We know that the global optimum is 50 nodes of each color, with all the nodes of the same color consecutive.

At each iteration of the annealing, a random node is chosen, and its color changed to a random color. This proposed move is accepted if the Metropolis criterion is accepted. At the end of the annealing, at the top of the figure,

Figure 11.3: Same as Figure 11.2, except the basic move is to change the color of a graph node to the color of one of the neighbors.

a good balance is achieved, with each color having equal numbers of nodes, but there are 14 places where the color changes (communication cost = 14), rather than the minimum four.

Heuristics

In choosing the change to be made to the state of the system, there may be intuitive or heuristic reasons to choose a change which tends to reduce the cost function. For our example of load balancing, we know that the optimal coloring of the graph has equal sized compact 'globules'; if we were to restrict the new color of a node to be the color of one of its two neighbors, then the boundaries between colors move without creating new domains.

The effect of this algorithm is shown in Figure 11.3, with the same number of iterations as Figure 11.2. The imbalance of 100 white nodes is quickly removed, but there are only three colors of 67 nodes each in the (periodically connected) final configuration. The problem is that the changes do not satisfy reachability; if a color is not present in graph coloring, then it can never come back.

Even if reachability is satisfied, a heuristic may degrade the quality of the final optimum, because a heuristic is coercing the state toward local minima in much the same way that a low temperature would. This may reduce the ability of the annealing algorithm to explore the state space, and cause it to drop into a local minimum and stay there, resulting in poor performance overall.

In Figure 11.4 is shown a solution to this problem. With high probability

Figure 11.4: Same as Figure 11.2, except the basic move is to change the color of a graph node to the color of one of the neighbors with large probability, and to a random color with small probability.

the new color is one of the neighbors, but also there is a small probability of a 'seed' color, which is a randomly chosen color. Now we see a much better final configuration, close to the global optimum. The balance is perfect and there are five separate domains instead of the optimal four.

Collisional Simulated Annealing

As presented so far, simulated annealing is a sequential algorithm, since whenever a move is made an acceptance decision must be made before another move may be evaluated. A parallel variant, which we shall call *collisional* simulated annealing, would be to propose several changes to the state of the system, evaluate the Metropolis criterion on each simultaneously, then make those changes which are accepted. Figure 11.5 shows the results of the same set of changes as Figure 11.4, but doing 16 simultaneous changes instead of sequentially. Now there are eight domains in the final configuration rather than five. The essential difference from the sequential algorithm is that δH resulting from several simultaneous changes is not the sum of the δH values if the changes are made in sequence. We tend to get *parallel collisions*, where there may be two changes each of which is beneficial, but the two together are detrimental. For example a married couple might need to buy a lawn mower: if either buys it, the result is beneficial to the couple, but if both simultaneously buy lawn mowers, the result is detrimental because they only need one.

Figure 11.5 shows how parallel collisions can adversely affect the load

Figure 11.5: Same as Figure 11.4, except the optimization is being carried out in parallel by 16 processors. Note the fuzzy edges of the domains caused by parallel collisions.

balancing process. At left, two processors share a small mesh, shown by the two colors, with a sawtooth division between them. There are seven edges with different colors on each side. In the middle are shown the separate views of the situation by each processor, and each processor discovers that by changing the color of the teeth of the sawtooth it can reduce the boundary from seven to four. On the right is shown the result of these simultaneous changes; the boundary has increased to 15, instead of the four that would result if only one processor went ahead.

The problem with this parallel variant is of course that we are no longer doing the correct algorithm, since each processor is making changes without consulting the others. As noted in [Baiardi:89a], [Barajas:87a], [Braschi:90a], [Williams:86b], we have an algorithm which is highly parallel, but not particularly efficient. We should note that when the temperature is close to zero, the success rate of changes (ratio of accepted to proposed changes) falls to zero: since a parallel collision depends on two successful changes, the parallel collision rate is proportional to the square of the low success rate, so that the effects of parallel collisions must be negligible at low temperatures.

One approach [Fox:88a] [Johnson:86a] to the parallel collision problem is *rollback*. We make the changes in parallel, as above, then check to see if any parallel collisions have occurred, and if so, undo enough of the changes so that there are no collisions. While rollback ensures that the algorithm is carried out correctly, there may be a great deal of overhead, especially in a tightly coupled system at high temperature, where each change may collide

Figure 11.6: Illustration of a parallel collision during load balance. Each processor may take changes which decrease the boundary length, but the combined changes increase the boundary.

with many others, and where most changes will be accepted. In addition, of course, rollback involves a large software and memory overhead since each change must be recorded in such a way that it can be rescinded, and a decision must be reached about which changes are to be undone.

For some cost functions and sets of changes, it may be possible to divide the possible changes into classes such that parallel changes within a class do not collide. An important model in statistical physics is the *Potts model* [Wu:82a], whose cost function is the same as the communication part of the load balance cost function. If the underlying graph is a square lattice, the graph nodes may be divided into 'red' and 'black' classes, so called because the arrangement is like the red and black squares of a checkerboard. Then we may change all the red nodes or all the black nodes in parallel with no collisions.

Some highly efficient parallel simulated annealing algorithms have been implemented [Coddington:90a] for the Potts model using clustering. These methods are based on the locality of the Potts cost function: the change in cost function from a change in the color of a graph node depends only on the colors of the neighboring nodes of the graph. Unfortunately, the balance part of the cost function interferes with this locality in that widely separated (in terms of the Hamming distance) changes may collide, so these methods are not suitable for load balancing.

In this book, we shall use the simple collisional simulated annealing algorithm, making changes without checking for parallel collisions. Further

Figure 11.7: Same as Figure 11.4, except the basic move is to change the color of a connected cluster of nodes.

work is required to invent and test more sophisticated parallel algorithms for simulated annealing, which may be able to avoid the degradation of performance caused by parallel collisions without unacceptable inefficiency from the parallelism [Baiardi:89a].

Clustering

Since the basic change made in the graph coloring problem is to change the color of one node, a boundary can move at most one node per iteration. The boundaries between processors are diffusing toward their optimal configurations. A better change to make is to take a connected set of nodes which are the same color, and change the color of the entire set at once [Coddington:90a]. This is shown in Figure 11.7 where the cluster is chosen first by picking a random node; we then add nodes probabilistically to the cluster, in this case the neighbor is added with probability 0.8 if it has the same color, and never if it has a different color. Once a neighbor has failed to be added, the cluster generation finishes. The coloring of the graph becomes optimal extremely quickly compared to the single color change method of Figure 11.4.

Figure 11.8 shows the clustered simulated annealing running in parallel, where 16 clusters are chosen simultaneously. The performance is degraded, but still better than Figure 11.5, which is parallel but with single color changes.

Summary of the Algorithm

Figure 11.8: Same as Figure 11.5, except that the cluster method is being carried out in parallel by 16 processors.

The annealing algorithm as presented so far requires several parameters to be chosen for tuning, which are in *italic font* in the description below.

First, we pick the initial coloring of the graph so that each graph node takes a color corresponding to the processor in which it currently resides. We form a population table, of which each processor has a copy of N_q, the number of nodes which have color q. We pick a value for μ, the *importance of communication.*

We pick a *maximum temperature* and the *number of stages* during which the temperature is to be reduced to zero. Each stage consists of a number of changes to the graph coloring which may be accepted or rejected, with no communication between the processors. At the end of the stage, each processor has a different idea of the population table, and the colors of neighboring graph nodes which are in different processors, because each processor has made changes without knowledge of the others. At the end of the stage, the processors communicate to update the population tables and local neighbor information so that each processor has up-to-date information. Each stage consists of either having a given *number of accepted changes*, or having a *given number of rejected changes*, whichever comes first, followed by a loosely synchronous communication between processors.

Each trial move within a stage consists of looking for a cluster of uniform color, choosing a new color for the cluster, evaluating the change in cost function, and using the Metropolis criterion to decide whether to accept it. The cluster is chosen by first picking a random graph node as a seed, and probabilistically forming a cluster. Neighboring nodes are added to the

cluster with a given *cluster probability* if they are the same color as the seed and reside in the same processor.

The proposed new color for the cluster is chosen to be either random, with given *seed probability*, or to be a random color chosen from the set of neighbors of the cluster. The Metropolis criterion is then used to decide if the color change is to be accepted, and if so, the local copy of the population table is updated.

11.2.6 Recursive Bisection

Rather than coloring the graph by direct minimization of the load balance cost function, we may do better to reduce the problem to a number of smaller problems. The idea of recursive bisection is that it is easier to color a graph with two colors than many colors. We first split the graph into two halves, minimizing the communication between the halves. We can then color each half with two colors, and so on, recursively bisecting each subgraph.

There are two advantages to recursive bisection, firstly that each subproblem (coloring a graph with two colors) is easier than the general problem, and secondly that there is natural parallelism. While the first stage is splitting a single graph in two, and is thus a sequential problem, there is two-way parallelism at the second stage, when the two halves are being split, and four-way parallelism when the four quarters are being split. Thus, coloring a graph with P colors is achieved in a number of stages which is logarithmic in P.

Both of the recursive bisection methods we shall discuss split a graph into two by associating a scalar quantity, s_e, with each graph node, e, which we may call a *separator* field. By evaluating the median S of the s_e, we can color the graph according to whether s_e is greater or less than S. The median is chosen as the division so that the numbers of nodes in each half are automatically equal; the problem is now reduced to that of choosing the field, s_e, so that the communication is minimized.

Orthogonal Recursive Bisection

A simple and cheap choice [Fox:88mm] for the separator field is based on the position of the finite elements in the mesh. We might let the value of s_e be the x-coordinate of the center of mass of the element, so that the mesh is split in two by a median line parallel to the y-axis. At the next stage, we split the submesh by a median line parallel to the x-axis, alternating between x and y stage by stage, as shown in Figure 11.9.

Figure 11.9: Load balancing by ORB for four processors. The elements (left) are reduced to points at their centers of mass (middle), then split into two vertically, then each half split into two horizontally. The result (right) shows the assignment of elements to processors.

11.2.7 Eigenvalue Recursive Bisection

Better but more expensive methods for splitting a graph are based on finding a particular eigenvector of a sparse matrix which has the structure of the adjacency matrix of the graph, and using this eigenvector as a separator field [Barnes:82a], and [Boppana:87a], [Pothen:89a].

Neural Net Model

For our discussion of eigenvector bisection, we use the concept of a computational neural net, based on the model of Hopfield and Tank [Fox:88tt], [Hopfield:85b]. When the graph is to be colored with two colors, these may be conveniently represented by the two states of a neuron, which we conventionally represent by the numbers -1 and $+1$. The Hopfield-Tank neural net finds the minimum of a 'computational energy', which is a negative-definite quadratic form over a space of variables which may take these values -1 and $+1$, and consequently is ideally suited to the two-processor load balance problem. Rewriting the load balance cost function,

$$H \propto \sum_{e \leftrightarrow f} \frac{1}{2}(V_e - V_f)^2 + \mu^{-1}\left(\sum_e V_e\right)^2 - \xi \sum_e (V_e^2 - 1)$$

where the V_e are 'neural firing rates', which are continuous variables during the computation and tend to one as the computation progresses. The first

term of this expression is the communication part of the cost function, the second term ensures equal numbers of the two colors if μ is small enough, and the third term is zero when the V_e are one, but pushes the V_e away from zero during the computation; the latter is to ensure that H is negative-definite, and the large but arbitrary constant ξ plays no part in the final computation. The firing rate or output V_e of a neuron is related to its *activity* u_e by a sigmoid function which we may take to be $V_e = \tanh \beta u_e$. The constant β adjusts the 'gain' of the neuron as an amplifier. The evolution equations to be solved are then:

$$\frac{d}{dt} u_e = -\frac{u_e}{\tau} - d_e V_e + \sum_{e \leftrightarrow f} V_f + \xi V_e - \mu^{-1} \sum_e V_e$$

where τ is a time constant for the system and d_e is the degree (number of neighbors) of the graph node e. If the gain is sufficiently low, the stable solution of this set of equations is that all the u_e are zero, and as the gain becomes large, the u_e grow and the V_e tend to either -1 or $+1$. The neural approach to load balancing thus consists of slowly increasing the gain from zero while solving this set of coupled nonlinear differential equations.

Let us now linearize this set of equations for small values of u_e, meaning that we neglect the hyperbolic tangent, because for small x, $\tanh x \approx x$. This linear set of equations may be written in terms of the vector $\mathbf{u}$ of all the u_e values and the adjacency matrix $\mathbf{A}$ of the graph, whose element A_{ef} is one if and only if the distinct graph nodes e and f are connected by an edge of the graph. We may write

$$\frac{d\mathbf{u}}{dt} + \frac{\mathbf{u}}{\tau} = \beta(-\mathbf{D} + \mathbf{A} + \xi \mathbf{I} - \mu^{-1}\mathbf{E})\mathbf{u} = \beta \mathbf{N} \mathbf{u}$$

where $\mathbf{D}$ is a diagonal matrix whose elements are the degrees of the graph nodes, $\mathbf{I}$ is the identity matrix, and $\mathbf{E}$ is the matrix with one in each entry. This linear set of equations may be solved exactly from a knowledge of the eigenvalues and eigenvectors of the symmetric matrix $\mathbf{N}$. If ξ is sufficiently large, all eigenvalues of $\mathbf{N}$ are positive, and when β is greater than a critical value, the eigenvector of $\mathbf{N}$ corresponding to its largest eigenvalue grows exponentially. Of course when the neuron activities are no longer close to zero, the growth is no longer exponential, but this initial growth determines the form of the emerging solution.

If μ is sufficiently small, so that balance is strongly enforced, then the eigenspectrum of $\mathbf{N}$ is dominated by that of $\mathbf{E}$. The highest eigenvalue of

$\mathbf{N}$ must be chosen from the space of the lowest eigenvalue of $\mathbf{E}$. The lowest eigenvalue of $\mathbf{E}$ is zero, with eigenspace given by those vectors with $\mathbf{Ex} = 0$, which is just the balance condition. We observe that multiples of the identity matrix make no difference to the eigenvectors, and conclude that the dominant eigenvector $\mathbf{s}$ satisfies $(-\mathbf{D} + \mathbf{A})\mathbf{s} = \lambda\mathbf{s}$ and $\mathbf{Es} = 0$, where λ is maximal. The matrix $\mathbf{D} - \mathbf{A}$ is the *Laplacian* matrix of the graph [Pothen:89a], and is positive semi-definite. The lowest eigenvector of the Laplacian has eigenvalue zero, and is explicitly excluded by the condition $\mathbf{Es} = 0$. Thus, it is the second eigenvector which we use for load balancing.

In summary, we have set up the load balance problem for two processors as a neural computation problem, producing a set of nonlinear differential equations to be solved. Rather than solve these, we have assumed that the behavior of the final solution is governed by the eigenstate which first emerges at a critical value of the gain. This eigenstate is the second eigenvector of the Laplacian matrix of the graph.

If we split a connected graph in two equal pieces while minimizing the boundary, we expect each half to be a connected subgraph of the original graph. This intuition is supported by a theorem of Fiedler [Fiedler:75a], [Fiedler:75b] that when we do the splitting by the second eigenvector of the Laplacian matrix, the halves are indeed connected.

To calculate this second eigenstate, we use the Lanczos method [Golub:83a], [Parlett:80a], [Pothen:89a]. We can explicitly exclude the eigenvector of value zero, because the form of this eigenvector is equal entries for each element of the vector. The accuracy of the Lanczos method increases quickly with the number of *Lanczos vectors* used. We find that 30 Lanczos vectors are sufficient for splitting a graph of 4,000 nodes.

A closely related eigenvector method [Barnes:82a], [Boppana:87a] is based on the second highest eigenvector of the adjacency matrix of the graph, rather than the second lowest eigenvector of the Laplacian matrix. The advantage of the Laplacian method is in the implementation: the first eigenvector is known exactly (the vector of all equal elements), so that it can be explicitly deflated in the Lanczos method.

11.2.8 Testing Method

The applications described in Section 11.2.2 have been implemented with DIME (Distributed Irregular Mesh Environment), described in Section 10.1.

We have tested these three load balancing methods using the application code 'Laplace' described in Section 11.2.2. The problem to be solved is to

Figure 11.10: Solution of the Laplace equation used to test load balancing methods. The outer boundary has voltage increasing linearly from -1.2 to 1.2 in the vertical direction, the light shade is voltage 1, and the dark shade voltage -1.

solve Laplace's equation with Dirichlet boundary conditions, in the domain shown in Figure 11.10. The square outer boundary has voltage linearly increasing vertically from -1.2 to $+1.2$, the lightly shaded S-shaped internal boundary has voltage $+1$, and the dark shaded hook-shaped internal boundary has voltage -1. Contour lines of the solution are also shown in the figure, with contour interval 0.08.

The test begins with a relatively coarse mesh of 280 elements, all residing in a single processor, with the others having none. The Laplace equation is solved by Jacobi iteration, and the mesh refined based on the solution obtained so far, then the mesh is balanced by the method under test. This sequence: solve, refine, balance, is repeated seven times until the final mesh has 5,772 elements. The starting and ending meshes are shown in Figure 11.11.

The refinement is solution-adaptive, so that the set of elements to be refined is based on the solution that has been computed so far. The refinement criterion is the magnitude of the gradient of the solution, so that the most heavily refined part of the domain is that between the S-shaped and hook-shaped boundaries where the contour lines are closest together. At each refinement, the criterion is calculated for each element of the mesh, and a value is found such that a given proportion of the elements are to be refined, and those with higher values than this are refined loosely synchronously. For this test of load balancing, we refined 40% of the elements of the mesh at each stage.

Figure 11.11: Initial and final meshes for the load balancing test. The initial mesh with 280 elements is essentially a uniform meshing of the square, and the final mesh of 5,772 elements is dominated by the highly refined S-shaped region in the center.

This choice of refinement criterion is not particularly to improve the accuracy of the solution, but to test the load balancing methods as the mesh distribution changes. The initial mesh is essentially a square covered in mesh of roughly uniform density, and the final mesh is dominated by the long, thin S-shaped region between the internal boundaries, so the mesh changes character from two-dimensional to almost one-dimensional.

We ran this test sequence on 16 nodes of an NCUBE/10 parallel machine, using ORB and ERB and two runs with SA, the difference being a factor of ten in cooling rate, and different starting temperatures.

The Eigenvalue Recursive Bisection used the deflated Lanczos method for diagonalization, with three iterations of 30 Lanczos vectors each to find the second eigenvector. These numbers were chosen so that more iterations and Lanczos vectors produced no significant improvement, and fewer degraded the performance of the algorithm.

The parameters used for the collisional annealing were as follows:

- The starting temperature for the run labelled SA1 was 0.2, and for the run labelled SA2 was 1.0. In the former case, movement of the boundaries is allowed, but a significant memory of the initial coloring is retained. In the latter case, large fluctuations are allowed, the system is heated to randomness, and all memory of the initial configuration is erased.

Figure 11.12: Processor divisions resulting from the load balancing algorithms. Top, ORB at the fourth and fifth stages; lower left ERB at the fifth stage; lower right SA2 at the fifth stage.

- The boundary importance was set at 0.1, which is large enough to make communication important in the cost function, but small enough that all processors will get their share of elements.
- The curves labelled SA1 correspond to cooling to zero temperature in 500 stages, those labelled SA2 to cooling in 5,000 stages.
- Each stage consisted of finding either one successful change (per processor) or 200 unsuccessful changes before communicating, and thus getting the correct global picture.
- The cluster probability was set to 0.58, giving an average cluster size of about 22. This is a somewhat arbitrary choice and further work is required to optimize this.

In Figure 11.12, we show the divisions between processor domains for the three methods at the fifth stage of the refinement, with 2,393 elements in the mesh. The figure also shows the divisions for the ORB method at the fourth stage: note the unfortunate processor division to the left of the S-shaped boundary which is absent at the fifth stage.

11.2.9 Test Results

We made several measurements of the running code, which can be divided into three categories:

Figure 11.13: Machine-independent measures of load balancing performance. Left, percentage load imbalance; lower left, total amount of communication; below, total number of messages.

Machine-independent Measurements

These are measurements of the quality of the solution to the graph coloring problem which are independent of the particular machine on which the code is run.

Let us define *load imbalance* to be the difference between the maximum and minimum numbers of elements per processor compared to the average number of elements per processor.

The two criteria for measuring communication overhead are the *total traffic size*, which is the sum over processors of the number of floating-point numbers sent to other processors per iteration of the Laplace solver, and the *number of messages*, which is the sum over processors of the number of messages used to accomplish this communication.

These results are shown in Figure 11.13. The load imbalance is significantly poorer for both the SA runs, because the method does not have the exact balance built in as do the RB methods, but instead exchanges load imbalance for reducing the communication part of the cost function. The imbalance for the RB methods comes about from splitting an odd number of elements, which of course cannot be exactly split in two.

There is a sudden reduction in total traffic size for the ORB method between the fourth and fifth stages of refinement. This is caused by the geometry of the mesh as shown at the top of Figure 11.12; at the fourth stage the first vertical bisection is just to the left of the light S-shaped region

creating a large amount of unnecessary communication, and for the fifth and subsequent stages the cut fortuitously misses the highly refined part of the mesh.

Machine-dependent Measurements

These are measurements which depend on the particular hardware and message-passing software on which the code is run. The primary measurement is of course the time it takes the code to run to completion; this is the sum of start-up time, load balancing time, and the product of the number of iterations of the inner loop times the time per iteration. For quasi-static load balancing, we are assuming that the time spent in the inner loop is much longer than the load balance time, so this is our primary measurement of load balancing performance. Rather than use an arbitrary time unit such as seconds for this measurement, we have counted this time per iteration as an equivalent number of floating point operations (flops). For the NCUBE this time unit is 15 μs for a 64-bit multiply. Thus, we measure *flops per iteration* of the Jacobi solver.

The secondary measurement is the *communication time* per iteration, also measured in flops. This is just the local communication in the graph, and does not include the time for the global combine which is necessary to decide if the Laplace solver has reached convergence.

Figure 11.14 shows the timings measured from running the test sequence on the 16-processor NCUBE. For the largest mesh, the difference in running time is about 18% between the cheapest load balancing method (ORB) and the most expensive (SA2). The ORB method spends up to twice as much time communicating as the others, which is not surprising, since ORB pays little attention to the structure of the graph it is splitting, concentrating only on getting exactly half of the elements on each side of an arbitrary line.

The curves on the right of Figure 11.14 show the time spent in local communication at each stage of the test run. It is encouraging to note the similarity with the lower left panel of Figure 11.13, showing that the time spent communicating is roughly proportional to the total traffic size, confirming this assumption made in Section 11.2.3.

Measurements for Dynamic Load Balancing

After refinement of the mesh, one of the load balancing algorithms is run and decisions are reached as to which of a processor's elements are to be sent away, and to which processor they are to be sent. As discussed in Section 10.1, a significant fraction of the time taken by the load balancer is

Figure 11.14: Machine-dependent measures of load balancing performance. Left, running time per Jacobi iteration in units of the time for a floating-point operation (flop); right, time spent doing local communication in flops.

taken in this migration of elements, since not only must the element and its data be communicated, but space must be allocated in the new processor and other processors must be informed of the new address of the element, and so on. Thus, an important measure of the performance of an algorithm for dynamic (in contrast to quasi-dynamic) load balancing is the number of *elements migrated*, as a proportion of the total number of elements.

Figure 11.15 shows the percentage of the elements which migrated at each stage of the test run. The one which does best here is ORB, because refinement causes only slight movement of the vertical and horizontal median lines. The SA runs are different because of the different starting temperatures: SA1 started at a temperature low enough that the edges of the domains were just 'warmed up', in contrast to SA2 which started at a temperature high enough to completely forget the initial configuration and, thus, essentially all the elements are moved. The ERB method causes the largest amount of element migration, which is because of two reasons. The first is because some elements are migrated several times because the load balancing is done in $\log_2 P$ stages for P processors; this is not a fundamental problem, and arises from the particular implementation of the method used here. The second reason is that a small change in mesh refinement may lead to a large change in the second eigenvector; perhaps a modification of the method could use the distribution of the mesh before refinement to create an inertial term so that the change in eigenvector as the mesh is refined could be controlled.

Figure 11.15: Percentage of elements migrated during each load balancing stage. The percentage may be greater than 100 because the recursive bisection methods may cause the same element to be migrated several times.

The migration time is only part of the time taken to do the load balancing, the other part being that taken to make the decisions about which element goes where. The total times for load balancing during the seven stages of the test run (solving the coloring problem plus the migration time) are shown in the table below:

Method	Time (minutes)
ORB	5
ERB	11
SA1	25
SA2	230

For the test run, the time per iteration was measured in fractions of a second, and it took few iterations to obtain full convergence of the Laplace equation, so that a high-quality load balance is obviously irrelevant for this simple case. The point is that the more sophisticated the algorithm for which the mesh is being used, the greater the time taken in using the distributed mesh compared to the time taken for the load balance. For a sufficiently complex application, for example unsteady reactive flow simulation, the calculations associated with each element of the mesh may be enough that a few minutes spent load balancing is completely negligible, so that the quasi-dynamic assumption is justified.

11.2.10 Conclusions

The Laplace solver that we used for the test run embodies the typical operation that is done with finite element meshes, which is matrix-vector multiply. Thus, we are not testing load balancing strategies just for a Laplace solver but for a general class of applications, namely those which use matrix-vector multiply as the heart of a scheme which iterates to convergence on a fixed mesh, then refines the mesh and repeats the convergence.

The case of the Laplace solver has a high ratio of communication to calculation, as may be seen from the discussion of Section 11.2.2, and thus brings out differences in load balancing algorithms particularly well.

Each load balancing algorithm may be measured by three criteria:

- The quality of the solution it produces, measured by the time per iteration in the solver;
- The time it takes to do the load balancing, measured by the time it takes to solve the graph coloring problem and by the number of elements which must then be migrated;
- The portability of the method for different kinds of applications with different kinds of meshes, and the number of parameters that must be set to obtain optimal performance from the method.

Orthogonal Recursive Bisection is certainly cheap, both in terms of the time it takes to solve the graph coloring problem and the number of elements which must be migrated. It is also portable to different applications, the only required information being the dimensionality of the mesh, and easy to program. Our tests indicate, however, that more expensive methods can improve performance by over 20%. Because ORB pays no attention to the connectivity of the element graph, one suspects that as the geometry of the underlying domain and solution become more complex, this gap will widen.

Simulated Annealing is actually a family of methods for solving optimization problems. Even when run sequentially, care must be taken in choosing the correct set of changes that may be made to the state space, and in choosing a temperature schedule to ensure a good optimum. We have tried a 'brute force' parallelization of simulated annealing, essentially ignoring the parallelism. For sufficiently slow cooling, this method produces the best solution to the load balancing problem when measured either against the load balance cost function, or by timings on a real parallel computer.

Unfortunately, it takes a long time to produce this high-quality solution, perhaps because some of the numerous input parameters are not set optimally. More probably a more sensitive treatment is required to reduce or eliminate parallel collisions [Baiardi:89a]. Clearly, further work is required to make SA a portable and efficient parallel load balancer for parallel finite element meshes. True portability may be difficult to achieve for SA, because the problem being solved is graph coloring, and graphs are extremely diverse; perhaps something approaching an expert system may be required to decide the optimal annealing strategy for a particular graph.

Eigenvalue Recursive Bisection seems to be a good compromise between the other methods, providing a solution of quality near that of SA at a price a little more than that of ORB. There are few parameters to be set, which are concerned with the Lanczos algorithm for finding the second eigenvector. Mathematical analysis of the ERB method takes place in the familiar territory of linear algebra, in contrast to analysis of SA in the jungles of non-equilibrium thermodynamics. A major point in favor of ERB for balancing finite element meshes is that the software for load balancing with ERB is shared to a large extent with the body of finite element software: the heart of the eigenvector calculation is a matrix-vector multiply, which has already been efficiently coded elsewhere in the finite element library.

11.3 An Improved Method for the Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is probably the most well-known member of the wider field of Combinatorial Optimization (CO) problems. These are dificult optimization problems where the set of feasible solutions (trial solutions which satisfy the constraints of the problem but are not necessarily optimal) is a finite, though usually very large set. The number of feasible solutions grows as some combinatoric factor such as $N!$ where N characterizes the size of the problem.

It has often been the case that progress on the TSP has led to progress on many CO problems and more general optimization problems. In this way, the TSP is a playground for the study of CO problems. Though the present work concentrates on the TSP, a number of our ideas are general and apply to all optimization problems.

The most significant issues occur as one tries to find extremely good or exact solutions to the TSP. Many algorithms exist which are fast and find

feasible solutions which are within a few percent of the optimum length. Here, we present algorithms which will usually find exact solutions to substantial instances of the TSP. We are limited by space considerations to a brief presentation of the method—more details may be found in [Martin:89a].

In a general instance of the TSP one is given N "cities" and a matrix d_{ij} giving the distance or cost function for going from city i to j. Without loss of generality, the distances can be assumed positive. A "tour" consists of a list of N cities, $tour[i]$, where each city appears once and only once. In the TSP, the problem is to find the tour with the minimum "length," where length is defined to be the sum of the lengths along each step of the tour,

$$length = \sum_{k=0}^{N-1} d_{tour[k],tour[k+1]},$$

and $tour[N]$ is identified with $tour[0]$ to make it periodic.

Most common instances of the TSP have a symmetric distance matrix; we will hereafter focus on this case. All CO problems can be formulated as optimizing an objective function (e.g., the length) subject to constraints (e.g., legal tours).

11.3.1 Background on Local Search Heuristics

In a local search method, one first defines a neighborhood topology on the set of all tours. For instance, one might define the neighborhood of a tour T_1 to be all those tours which can be obtained by changing at most k edges of T_1. A tour is said to be locally opt if no tour in its neighborhood is shorter than it. One can search for locally opt tours by starting with a random tour T_1 and performing k-changes on it as long as the tour length decreases. In this way, one constructs a sequence of tours T_1, T_2, ... Eventually the process stops and one has reached a local opt tour. Lin [Lin:65a] studied the case of $k = 2$ and $k = 3$, and showed that one could get quite good tours quickly. Furthermore, since in general there are quite a lot of locally opt tours, in order to find the globally optimal tour, he suggested repeating this process from random starts many times until one was confident all the locally opt tours had been found. Unfortunately, the number of local opt tours rises exponentially with N, the number of cities. Thus in general, it is more efficient to use a more sophisticated local opt (say higher k) than to try to repeat the search from random starts many times. The current state-of-the-art optimization heuristic is an algorithm due to Lin and Kernighan

[Lin:73a]. It is a variable depth k-neighborhood search, and it is the benchmark against which all heuristics are tested. Since it is significantly better than three-opt, for any instance of the TSP, there are many fewer L–K-opt tours than there are three-opt tours. This postpones the problem of doing exponentially many random starts until one reaches N on the order of a few hundred. For still larger N, the number of L–K-opt tours itself gets unmanageable. Given that one really does want to tackle these larger problems, there are two natural ways to go. First, one can try to extend the neighborhood which L–K considers, just as L–K extended the neighborhood of three-changes. Second, one expects that instead of sampling the local opt tours in a random way as is done by applying the local searches from random starts many times, it might be possible to obtain local opt tours in a more efficient way, say via a sampling with a bias in favor of the shorter tours. We will see that this gives rise to an algorithm which indeed enables one to solve much larger instances.

11.3.2 Background on Markov Chains and Simulated Annealing

Given that any local search method will stop in one of the many local opt solutions, it may be useful to find a way for the iteration to escape by temporarily allowing the tour length to increase. This leads to the popular method of "Simulated Annealing" [Kirkpatrick:83a].

One starts by constructing a sequence of tours T_1, T_2, etc. At each step of this chain, one does a k-change (moves to a neighboring tour). If this decreases the tour length, the change is accepted; if the tour length increases, the change is rejected with some probability in which case one simply keeps the old tour at that step. Such a stochastic construction of a sequence of T's is called a Markov chain. It can be viewed as a rather straightforward extension of the above local search to include 'noisiness' in the search for shorter tours. Because increases in the tour length are possible, this chain never reaches a fixed point. For many such Markov chains, it is possible to show that given enough time, the chain will visit every possible tour T, and that for very long chains, the T's appear with a calculable probability distribution. Such Markov chains are closely inspired by physical models where the chain construction procedure is called a Monte Carlo. The stochastic accept/reject part is supposed to simulate a random fluctuation due to temperature effects, and the temperature is a parameter which measures the bias towards short tours. If one wants to get to the

globally optimal tour, one has to move the temperature down towards zero, corresponding to a strong bias in favor of short tours. Thus one makes the temperature vary with time, and the way this is done is called the annealing schedule, and the result is simulated annealing.

If the temperature is taken to zero too fast, the effect is essentially the same as setting the temperature to zero exactly, and then the chain just traps at a local opt tour forever. There are theoretical results on how slowly the annealing has to be done to be sure that one reaches the globally optimum solution, but in practice the running times are astronomical. Nevertheless, simulated annealing is a standard and widely used approach for many minimization problems. For the TSP, it is significantly slower than Lin-Kernighan, but it has the advantage that one can run for long times and slowly improve the quality of the solutions. See for instance the studies Johnson et al. [Johnson:91a] have done. The advantage is due to the improved sampling of the short length tours: simulated annealing is able to ignore the tours which are not near the minimum length. An intuitive way to think about it is that for a long run, simulated annealing is able to try to improve an already very good tour, one which probably has many links in common with the exact optimum. The standard Lin-Kernighan algorithm, by contrast, continually restarts from scratch, throwing away possibly useful information.

11.3.3 The New Algorithm—Large-Step Markov Chains

Simulated annealing does not take advantage of the local opt heuristics. This means that instead of sampling local opt tours as does L–K repeated from random starts, the chain samples all tours. It would be a great advantage to be able to restrict the sampling of a Markov chain to the local opt tours only. Then the bias which the Markov chain provides would enable one to sample the shortest local opt tours more efficiently than local opt repeated from random starts. This is what our new algorithm does.

To do this, one has to find a way to go from one local opt tour, T_n, to another, T_{n+1}, and this is the heart of our procedure. We propose to do a change on T_n, which we call a 'kick'. This can be a random p-change for instance but we will choose something smarter than that. Follow this kick by the local opt tour improvement heuristic until reaching a new local opt tour T_{n+1}. Then accept or reject T_{n+1} depending on the increase or decrease in tour length compared to T_n. This is illustrated in Figure 11.16. Since there are many changes in going from T_n to T_{n+1}, we call this method

Figure 11.16: Schematic Representation of the Objective Function and of the Tour Modification Procedure used in the Large-step Markov Chain

a 'Large-Step Markov Chain'. It can also be called 'Iterated Local Opt', but it should be realized that it is precisely finding a way to iterate which is the difficulty! The algorithm is far better than the small step Markov chain methods (conventional simulated annealing) because the accept/reject procedure is not implemented on the intermediate tours which are almost always of longer length. Instead, the accept/reject does not happen until the system has returned to a local minimum. The method directly steps from one local minimum to another. It is thus much easier to escape from local minima.

At this point, let us mention that this method is no longer a true simulated annealing algorithm. That is, the algorithm does NOT correspond to the simulation of any physical system undergoing annealing. The reason is that a certain symmetry property, termed "detailed balance" in the physics community, is not satisfied by the large step algorithm. [Martin:89a] says a bit more about this. One consequence of this is that the parameter "temperature" which one anneals with no longer plays the role of a true, physical temperature—instead it is merely a parameter which controls the bias towards the optimum. The lack of a physical analogy may be the reason that this algorithm has not been tried before, even though much more exotic algorithms (such as appealing to quantum mechanical analogies!) have been proposed.

We have found that in practice, this methodology provides an efficient sampling of the local opt tours. There are a number of criteria which need to be met for the biased sampling of the Markov chain to be more efficient

than plain random sampling. These conditions are satisfied for the TSP, and more generally whenever local search heuristics are useful. Let us stress before proceeding to specifics that this large step Markov chain approach is extremely general, being applicable to any optimization problem where one has local search heuristics. It enables one to get a performance which is at least as good as local search, with substantial improvements over that if the sampling can be biased effectively. Finally, although the method is general, it can be adapted to match the problem of interest through the choice of the kick. We will now discuss how to choose the kick for the TSP.

Consider for instance the case where the local search is three-opt. If we used a kick consisting of a three-change, the three-opt would very often simply bring us back to the previous tour with no gain. Thus it is probably a good idea to go to a four-change for the kick when the local search is three-opt. For more general local search algorithms, a good choice for the kick would be a k-change which does not occur in the local search. Surprisingly, it turns out that two-opt, three-opt, and especially L–K are structured so that there is one kick choice which is natural for all of them. To see this, it is useful to go back to the paper by Lin and Kernighan. In that paper, they define 'sequential' changes and they also show that if the tour is to be improved, one can force all the partial gains during the k-change to be positive. A consequence of this is that the check-out time for sequential k-changes can be completed in $O(N)$ steps. It is easy to see that all two and three changes are sequential, and that the first non-sequential change occurs at $k = 4$ (figure 2 of their paper). We call this graph a 'double-bridge' change because of what it does to the tour. It can be constructed by first doing a two-change which disconnects the tour; the second two-change must then reconnect the two parts, thereby creating a bridge. Note that both of the two-changes are bridges in their own way, and that the double-bridge change is the only non-sequential four-change which cannot be obtained by composing changes which are both sequential and leave the tour connected. If we included this double-bridge change in the definition of the neighborhood for a local search, check-out time would require $O(N^2)$ steps (a factor N for each bridge essentially). Rather than doing this change as part of the local search, we include such changes stochastically as our kick. The double bridge kick is the most natural choice for any local search method which considers only sequential changes. Because L–K does so many changes for k greater than three, but misses double-bridges, one can expect that most of what remains in excess length using L–K might be removed with our extension. The results below indicate that this is the case.

11.3.4 Results

At first we implemented the Large-Step Markov Chain for the three-opt local search. We checked that we could solve to optimality problems of sizes up to two hundred by comparing with a branch and bound program. For $N = 100$, the optimum was found in a few minutes on a SUN-3, while for $N = 200$ an hour or two was required. For larger instances, we used problems which had been solved to optimality by other people. We ran our program on the Lin-318 instance solved to optimality by Padberg and Crowder. Our iterated three-opt found the optimal tour on each of five separate runs, with an average time of less than 20 hours on the SUN-3. We also ran on the AT&T-532 instance solved to optimality by Padberg and Rinaldi. By using a post-reduction method inspired by tricks explained in the Lin-Kernighan paper, the program finds the optimum solution in 100 hours. It is of interest to ask what is the expected excess tour length for very large problems using our method with a reasonable amount of time. We have run on large instances of cities randomly distributed in the unit square. Ordinary three-opt gives an average length 3.6% above the Held-Karp bound, whereas the iterated three-opt does better than L–K (which is 2.2% above): it leads to an average of less than 2.0% above H–K. Thus we see that without much more algorithmic complexity, one can improve three-opt by more than 1.6-

In [Martin:89a], we suggested that such a dramatic improvement should also carry over to the L–K local opt algorithm. Since then, we have implemented a version of L–K and have run it on the instances mentioned above. Johnson [Johnson:90b] and also Cook, Applegate, Chvatal [Cook:90b] have similarly investigated the improvement of iterated L–K over repeated L–K. It is now clear that the iterated L–K is a big improvement. Iterated L–K is able to find the solution to the Lin-318 instance in minutes, and the solution to the AT&T-532 problem in an hour. At a recent TSP workshop [TSP:90a], a 783 city instance constructed by Pulleyblank was solved to optimality by ourselves, Johnson, and Cook et. al., all using the large-step method.

For large instances (randomly distributed cities), Johnson finds that iterated L–K leads to an average excess length of 0.84% above the Held-Karp bound. Previously it was expected that the exact optimum was somewhere above 1% from the Held-Karp bound, but iterated L–K disproves this conjecture.

One of the most exciting results of the experiments which have been performed to date is this: for "moderate" sized problems (such as the AT&T 532

or the 783 instance mentioned above), no careful "annealing" seems to be necessary. It is observed that just setting the temperature to zero (no uphill moves at all!) gives an algorithm which can often find the exact optimum. The implication is that, for the large-step Markov chain algorithm, the effective energy landscape has only one (or few) local minima! Almost all of the previous local minima have been modified to saddle points by the extended neighborhood structure of the algorithm.

11.3.5 Credits

Steve Otto had the original idea for the large-step Markov chain. Olivier Martin has made (and continues to) many improvements towards developing new, fast local search heuristics. Steve Otto and Edward Felten have developed the programs.

Chapter 12

Irregular Loosely Synchronous

12.1 Problem Structure

*** Contribution needed from G. C. Fox

12.2 Simulation of the Electrosensory System of the Fish *Ganathonemus petersii* [Williams:90c]

All animals are faced with the computationally intense task of continuously acquiring and analyzing sensory data from their environment. To ensure maximally useful data, animals appear to use a variety of motor strategies or behaviors to optimally position their sensory apparatus. In all higher animals, neural structures which process both sensory and motor information are likely to exist which can coordinate this exploratory behavior for the sake of sensory acquisition.

To study this feedback loop, we have chosen the weakly electric fish, which use a unique electrically based means of exploring their environment [Bullock:86a], [Lissman:58a]. These nocturnal fish, found in murky waters of the Congo and Amazon, have developed electrosensory systems to allow them to detect objects without relying on vision. In fact, in some species this electric sense appears to be their primary sensory modality.

This sensory system relies on an electric organ which generates a weak electric field surrounding the fish's body that in turn is detected by spe-

cialized electroreceptor cells in the fish's skin. The presence of animate or inanimate objects in the local environment causes distortions of this electric field, which are interpreted by the fish. The simplicity of the sensory signal, in addition to the distributed external representation of the detecting apparatus, makes the weakly electric fish an excellent animal with which to study the involvement in sensory discrimination of the motor system in general, and body position in particular.

Simulations in two dimensions [Bacher:83a], [Heiligenberg:75a] and measurements with actual fish have shown that body position, especially the tail angle, significantly alter the fields near the fish's skin.

To study quantitatively how the fish's behavior affects the "electric images" of objects, we are developing three-dimensional computer simulations of the electric fields that the fish generate and detect. These simulations, when calibrated with the measured fields, should allow us to identify and focus on behaviors that are most relevant to the fish's sensory acquisition tasks, and to predict the electrical consequences of the behavior of the fish with higher spatial resolution than possible in the tank.

Being able to visualize the electric fields, in false color on a simulated fish's body as it swims, may provide a new level of intuition into how these curious animals sense and respond to their world. For this simulation, we have chosen the fish *Apteronotus leptorhynchus.*

12.2.1 Physical Model

We need to reduce the great complexity of a biological organism to a manageable physical model. The ingredients of this model are the fish body, shown in Figure 12.1, the object that the fish is sensing, and the water exterior to both the fish and the object.

The real fish has some projecting fins, and our first approximation is to neglect these because their electrical properties are essentially the same as those of water.

We will assume that the fish is exploring a small conductive object, such as a small metal sphere. First, we reduce the geometrical aspect of the object to being pointlike, yet retaining some relevant electrical properties. Except when the object is another electric fish, we expect the object to have no active electrical properties, but only to be an *induced dipole.*

We now come to the modelling of the fish body itself. This consists of a skin with electroreceptor cells which can detect potential difference, and a rather complex internal structure. We shall assume that the source

Figure 12.1: Side and Top views of the Fish, and Internal Potential Model

voltage is maintained at the interface between the internal structure and the skin, so that we need not be concerned with the details of the internal structure. Thus, the fish body is modelled as two parts: an internal part with a given voltage distribution on its surface, surrounded by a skin with variable conductivity.

The upshot of this model is that we need to solve Laplace's equation in the water surrounding the fish, with an induced dipole at the position of the object the fish is investigating, with a mixed or Cauchy boundary condition at the surface of the fish body.

12.2.2 Mathematical Theory

The Boundary Element method [Brebbia:83a], [Cruse:75a] has been used for many applications where it is necessary to solve a linear elliptic partial differential equation. Because of the linearity of the underlying differential equation, there is a Green's function expressing the solution throughout the three-dimensional domain in terms of the behavior at the boundaries, so that the problem may be transformed into an integral equation on the boundary.

The discrete approximation to this integral equation results in the solution of a full set of simultaneous linear equations, one equation for each node of the boundary mesh; the conventional Finite difference method would result in solving a sparse set of equations, one for each node of a mesh filling space. Let us compare these methods in terms of efficiency and software cost.

To implement the finite difference method, we would first make a mesh

filling the domain of the problem, that is a three-dimensional mesh, then for each mesh point set up a linear equation relating its field value to that of its neighbors. We would then need to solve a set of sparse linear equations. In the case of an exterior problem such as ours, we would need to pay special attention to the far-field, making sure the mesh extends out far enough and that the proper approximation is made at this outer boundary.

With the Boundary Element method, we discretize only the surface of the domain, and again solve a set of linear equations, except that now they are no longer sparse. The far-field is no longer a problem, since this is taken care of analytically.

If it is possible to make a regular grid surrounding the domain of interest, then the Finite Difference method is probably more efficient, since multigrid methods or alternating direction methods will be faster than the solution of a full matrix. It is with complex geometries however, that the Boundary Element method can be faster and more efficient on sequential or distributed memory machines. It is much easier to produce a mesh covering a curved two-dimensional manifold than a three-dimensional mesh filling the space exterior to the manifold. If the manifold is changing from step to step, the two-dimensional mesh need only be distorted, whereas a three-dimensional mesh must be completely remade, or at least strongly smoothed, to prevent tangling. If the three-dimensional mesh is not regular, the user faces the not inconsiderable challenge of explicit load balancing and communication at the processor boundaries.

12.2.3 Results

Figure 12.2 shows a view of four of the model fish in some rather unlikely positions, with natural shading.

Figure 12.3 shows a side view of the fish with the free field (no object) shown in gray scale, and we can see how the potential ramp at the skin-body interface has been smoothed out by the resistivity of the skin. Figure 12.4 shows the computed potential contours for the midplane around the fish body, showing the dipole field emanating from the electric organ in the tail.

Figure ?? shows the difference field (voltage at the skin with and without the object) for three object positions, near the tail (left), at the center (middle) and near the head (right). It can be seen that this difference field, which is the sensory input for the fish, is greatest when the object is close to the head. A better view of the difference voltage is shown in Figure 12.5, which shows the envelope of the difference voltage on the midline of the fish,

Figure 12.2: Four fish with simple shading

Figure 12.3: Potential distribution on the surface of the fish, with no external object.

Figure 12.4: Potential contours on the midplane of the fish, showing dipole distribution from the tail.

for various object positions. Again, we see that the maximum sensory input occurs when the object is close to the head of the fish, rather than at the tail, where the electric organ is.

12.2.4 Credits and C3P References

The BEM algorithm was written as a DIME application by Roy Williams, Brian Rasnow of the Biology Division, and Chris Assad of the Engineering Division.

C3P References: [Williams:90b], [Williams:90c].

12.3 Transonic Flow

Unstructured meshes have been widely used for calculations with conventional sequential machines. Jameson [Jameson:86a], [Jameson:86b] uses explicit finite-element based schemes on fully unstructured tetrahedral meshes to solve for the flow around a complete aircraft, and other workers [Dannenhoffer:89a], [Holmes:86a], [Lohner:84a], [Lohner:85a], [Lohner:86a] have used unstructured triangular meshes. Jameson and others [Jameson:87a], [Jameson:87b], [Mavriplis:88a], [Perez:86a], [Usab:83a] have used multigrid methods to accelerate convergence. For this work [Williams:89b], we have used the two-dimensional explicit algorithm of Jameson and Mavriplis [Mavriplis:88a].

Figure 12.5: Gray-scale plots of voltage differences due to an object at positions (left) near tail, (middle) at center and (right) near head. Each object is 3 cm above mid-plane.

An explicit update procedure is local, and hence well matched to a massively parallel distributed machine, whereas an implicit algorithm is more difficult to parallelize. The implicit step consists of solving a sparse set of linear equations, where matrix elements are non-zero only for mesh-connected nodes. Matrix multiplication is easy to parallelize since it is also a local operation, and the solve may thus be accomplished by an iterative technique such as Conjugate Gradient, which consists of repeated matrix multiplications. If, however, the same solve is to be done repeatedly for the same mesh, the most efficient (sequential) method is first decomposing the matrix in some way, resulting in fill-in. In terms of the mesh, this fill-in represents non-local connection between nodes: indeed, if the matrix were completely filled the communication time would be proportional to N^2 for N nodes.

12.3.1 Compressible Flow Algorithm

The governing equations are the Euler equations, which are of advective type with no diffusion,

$$\frac{\partial \mathbf{U}}{\partial l} + \nabla \cdot \mathbf{F}(\mathbf{U}) = 0 \tag{12.1}$$

where $\mathbf{U}$ is a vector containing the information about the fluid at a point. I have used bold symbols to indicate an *information* vector, or a set of fields describing the state of the fluid. In this implementation, $\mathbf{U}$ consists of density, velocity, and specific total energy (or equivalently pressure); it could

also include other information about the state of the fluid such as chemical mixture or ionization data. **F** is the flux vector, and has the same structure as **U** in each of the two coordinate directions.

The numerical algorithm is explained in detail in [Mavriplis:88a], so only an outline is given here. The method uses linear triangular elements to approximate the field. First, a timestep is chosen for each node which is constrained by a local Courant condition. The calculation consists of two parts:

Advection:

At each node calculate **F** from **U**. Each element then averages **F** from its neighboring nodes, and calculates the flux across each edge of the element, which is then added back into the node opposite the edge. This change in **U** is combined across the representations of the node in different processors. If the node is at a boundary which is a hard surface, a modification is made so that no flux of mass or energy occurs through the surface.

Artificial Dissipation:

The artificial dissipation is calculated as a combination of approximations to the Laplacian and the double Laplacian of **U**, involving a combine step for each. The double Laplacian is only used where the flow is smooth, to prevent dissipation of strong shocks.

The time-stepping is done with a five-stage Runge-Kutta scheme, where the advection step is done five times, and the dissipation step is done twice. Since advection takes one communication stage and dissipation two, each full time step requires nine loosely synchronous communication stages.

12.3.2 Adaptive Refinement

After the initial transients have dispersed and the flow has settled, the mesh may be refined. The criterion used is based on the gradient of the pressure for deciding which elements are to be refined. The user specifies a percentage of elements which are to be refined, and a criterion

$$R_e = \nabla_e |\nabla p| \tag{12.2}$$

is calculated for each element. A value, R_{crit}, of this criterion is found such that the given percentage of elements have a value of R_e greater than R_{crit},

Figure 12.6: Results for Transonic Flow Simulation with DIME

and those elements are refined. The criterion is not simply the gradient of the pressure, because the strongest shock in the simulation would soak up all the refinement leaving weaker shocks unresolved. With the element area, ∇_e, in the criterion, regions will 'saturate' after sufficient refinement, allowing weaker shocks to be refined.

12.3.3 Example

Figure 12.6 shows the flow-field resulting from Mach 0.8 flow over a NACA0012 airfoil at 1.25 degrees angle of attack, computed with a 32-node machine. This problem is that used by the AGARD working group [AGARD:83a] in their benchmarking of compressible-flow algorithms. At the top left is the mesh used for the computation, with 5135 elements, after four stages of adaptive refinement. At top right is the logical structure of the mesh, showing elements and nodes separately. Notice that each processor owns about the same number of elements. At bottom left are Mach-number contours from this work, and at lower right is the same data from the AGARD benchmark group. The sonic line is shown by a heavy line in both plots.

Note the shock above the airfoil, and the corresponding increase in mesh density there and at the leading edge.

12.3.4 Timing [Williams:90a]

The efficiency of any parallel algorithm increases as the computational load dominates the communication load. In the case of a domain-decomposed

Figure 12.7: Timings for Transonic Flow

mesh, the computational time depends on the number of elements per processor, and the communication time on the number of nodes at the boundary of the processor domain. If there are N elements in total, distributed among n processors, we expect the computation to go as N/n and the communication as the square root of this, so that the efficiency should approach unity as the square root of n/N.

I have run the example described above starting with a mesh of 525 elements, and refining 50% of the elements. In fact, more than 50% will be refined because of the nature of the refinement algorithm: in practice it is about 70%. The refinement continues until the memory of the machine runs out.

Figure 12.7 shows timing results. At top right are results for one, four, 16, 64, and 256 NCUBE processors. The time taken per simulation time-step is shown for the compressible flow algorithm against number of elements in the simulation. The curves end when the processor memory is full. Each processor offers a nominal 512 Kb memory, but when all the software and communication buffers are accounted for, there is only about 120 Kb available for the mesh.

The top left of figure 12.7 shows the same curves for one, four, 16, 64, and 128 Symult processors, and at bottom left the results for one, four, 16, and 32 processors of the Meiko computing surface. For comparison, the bottom right shows the results for one head of the CRAY Y-MP, and also for the Sun Sparcstation.

Each figure has diagonal lines to guide the eye; these are lines of constant time per element. We expect the curves for the sequential machines to be

parallel to these because the code is completely local and the time should be proportional to the number of elements. For the parallel machines, we expect the discrepancy from parallel lines to indicate the importance of communication inefficiency.

12.3.5 Credits and C^{3}P References

The transonic flow algorithm was written as a DIME application by Roy Williams, using the algorithm of A. Jameson of Princeton University and D. Mavriplis of NASA ICASE.

C3P References: [Williams:89b], [Williams:90a], [Williams:90b].

12.4 N-Body Simulations with 180,000 Bodies

*** Contribution needed from J. Salmon

12.5 Fast Vortex Algorithm and Parallel Computing

Vortex methods are a powerful tool for the simulation of incompressible flows at high Reynolds number. They rely on a discrete Lagrangian representation of the vorticity field to approximately satisfy the Kelvin and Helmholtz theorems which govern the dynamics of vorticity for inviscid flows. A time splitting technique can be used to include viscous effects. The diffusion equation is considered separately after convecting the particles with an inviscid vortex method. In our work, the viscous effects are represented by the so-called deterministic method. The approach was extended to problems where a flux of vorticity is used to enforce the no-slip boundary condition.

In order to accurately compute the viscous transport of vorticity, gradients need to be well resolved. As the Reynolds number is increased, these gradients get steeper and more particles are required to achieve the requisite resolution. In practice, the computing cost associated with the convection step dictates the number of vortex particles and puts an upper bound on the Reynolds number that can be simulated with confidence. That threshold can be increased by reducing the asymptotic time complexity of the convection step from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$. The near-field of every vortex particle is identified. Within that region, the velocity is computed by considering the pairwise interaction of vortices. The speed-up is achieved by approximating

the influence of the rest of the domain, the far-field. In that context, the interaction of two vortex particles is treated differently depending on their spatial relation. The resulting computer code does not lend itself to vectorization but has been successfully implemented on concurrent computers.

12.5.1 Vortex Methods

Vortex methods (see [Leonard:80a]) are used to simulate incompressible flows at high Reynolds number. The two-dimensional inviscid vorticity equation,

$$\frac{\partial \omega}{\partial t} + \mathbf{u} \cdot \nabla \omega = 0 \quad ,$$

is solved by discretizing the vorticity field into Lagrangian vortex particles,

$$\omega(\mathbf{x}, t) = \sum_{j}^{N} \Gamma_j(t)\, \delta\left(\mathbf{x} - \mathbf{x}_j(t)\right) \quad ,$$

where α_j is the strength or the circulation of the j^{th} particle. For an incompressible flow, the knowledge of the vorticity is sufficient to reconstruct the velocity field. Using complex notation, the induced velocity is given by

$$\mathrm{w}(z, t) = \frac{i}{2\pi} \sum_{j}^{N} \frac{\Gamma_j}{(z - z_j)^*} \quad .$$

The velocity is evaluated at each particle location and the discrete Lagrangian elements are simply advected at the local fluid velocity. In this way, the numerical scheme approximately satisfies Kelvin and Helmholtz theorems that govern the motion of vortex lines. The numerical approximations have transformed the original partial differential equation into a set of $2N$ ordinary differential equations: an N-body problem. This class of problems is encountered in many fields of computational physics, e.g., molecular dynamics, gravitational interactions, plasma physics and of course, vortex dynamics.

12.5.2 Fast Algorithms

When each pairwise interaction is considered, distant and nearby pairs of vortices are treated with the same care. As a result, a disproportionate amount of time is spent computing the influence of distant vortices that

have little influence on the velocity of a given particle. This is not to say that the far-field is to be totally ignored since the accumulation of small contributions can have a significant effect. The key element in making the velocity evaluation faster is to approximate the influence of the far-field by considering groups of vortices instead of the individual vortices themselves. When the collective influence of a distant group of vortices is to be evaluated, the very accurate representation of the group provided by its vortices can be overlooked and a cruder description that retains only its most important features can be used. These would be the group location, circulation, and possibly, some coarse approximation of its shape and vorticity distribution.

A convenient approximate representation is based on multipole expansions. It would be possible to build a fast algorithm by evaluating the multipole expansion at the location of particles that do not belong to the group. This is basically the scheme used by Barnes and Hut [Appel:85a] (the concurrent implementation of this algorithm is discussed in Section 12.4). Greengard and Rokhlin [Barnes:86a] went a step further by proposing group to group interactions. In this case, the multipole expansion is transformed into a Taylor series around the center of the second group, where the influence of the first one is sought. The expansions provides an accurate representation of the velocity field when the distance between the groups is large compared to their radii.

One now needs a data structure that is going to facilitate the search for acceptable approximations. As proposed by Appel [Greengard:87b], a binary tree is used. In that framework, a giant cluster sits on top of the data structure; it includes all the vortex particles. It stores all the information relevant to the group, i.e., its location, its radius and the coefficients of the multipole expansion. In addition, it carries the address of its two children, each of them responsible for approximately half of the vortices of the parent group. Whenever smaller groups are sought, these pointers are used to rapidly access the relevant information. The children carry the description of their own group of vortices and are themselves pointing at two smaller groups, their own children, the grand-children of the patriarchal group. More subgroups are created by equally dividing the vortices of the parent groups along the "x" and "y" axis alternatively. This splitting process stops when all groups have approximately $J_{\min}$ vortices. Then, instead of pointing toward two smaller groups, the parent node points toward a list of vortices. This data structure provides a quick way to access groups, from the largest to the smallest ones, and ultimately to the individual vortices themselves. Appel's data structure is Lagrangian since it is built on top of the vortices and moves

Figure 12.8: Data Structure Assigned to Processor 1

with them. As a result, it can be used for many time steps.

Comparing the speed of this algorithm with the classical $\mathcal{O}(N^2)$ approach, the crossover occurs for as few as 150 vortices. At this point, the extra cost of maintaining the data structure is balanced by the savings associated with the approximate treatment of the far field. When N is increased further, the savings outweigh the extra bookkeeping and the proposed algorithm is faster than its competitor by a margin that increases with the number of vortices.

12.5.3 Hypercube Implementation

The global nature of the N^2 approach has made its parallel implementation fairly straightforward (see [Fox:80a]). However, that character was drastically changed by the fast algorithm as it introduced a strong component of locality. Globality is still present since the influence of particle is felt throughout the domain, but more care and computational effort is given to its near field. The fast parallel algorithm has to reflect that dual nature, otherwise an efficient implementation will never be obtained. Moreover, the domain decomposition can no longer ignore the spatial distribution of the vortices. Nearby vortices are strongly coupled computationally, so it makes sense to assign them to the same processor. Binary bisection is used in the host to spatially decompose the domain. Then, only the vortices are sent to the processors where a binary tree is locally built on top of them. For example, Figure 12.8 shows the portion of the data structure assigned to processor 1 in a four processor environment.

Figure 12.9: Data Structure Known to Processor 1 After Broadcast

In a fast algorithm context, sending a copy of local data structure to half the other processors does not necessarily result in a load balanced implementation. The work associated with processor to processor interactions now depends on their respective location in physical space. A processor whose vortices are located at the center of the domain is involved in more costly interactions than a peripheral processor. To achieve the best possible load balancing, that central processor could send a copy of its data to more than half of the other processors and hence, be itself responsible for a smaller fraction of the work associated with its vortices.

Before a decision is made on which one is going to visit and which one is going to receive, we minimize the number of pairs of processors that need to exchange their data structure. Following the domain decomposition, the portion of the data structure that sits above the subtrees is not present anywhere in the hypercube. That gap is filled by broadcasting the description of the largest group of every processor. By limiting the broadcast to one group per processor, a small amount of data is actually exchanged but, as seen on Figure 12.9, this step gives every processor a coarse description of its surroundings and helps it find its place in the universe.

If the vortices of processor `A` are far enough from those of processor `B`, it is even possible to use that coarse description to compute the interaction of `A` and `B` without an additional exchange of information. The far field of every processor can be quickly disposed of. After thinking globally, one now has to act locally; if the vortices of `A` are adjacent to those of `B`, a more detailed description of their vorticity field is needed to compute their mutual influence.

Figure 12.10: Parallel Efficiency of the Fast Algorithm

This requires a transfer of information from either A to B or from B to A. In the latter case, most of the work involved in the A-B interaction takes place in processor A. Obviously, processor B should not always send its information away since it would then remains idle while the rest of the hypercube is working. Load balancing concerns will dictate the flow of information.

12.5.4 Efficiency of Parallel Implementation

Since our objective is to compute the flow around a cylinder, the efficiency of the parallel implementation was tested on such a problem. The region for which $1 < r < 1.6$ is uniformly covered with N particles. The parallel efficiency is shown on Figure 12.10 as a function of the hypercube size. The parallel implementation is fairly robust the parallel efficiency, E, remains larger than 0.7. The number of vortices per processor was kept roughly constant at 1500 even if the parallel efficiency is not a strong function of the size of the problem. It is, however, much more sensitive to the quality of the domain decomposition. The fast parallel algorithm performs better when all the sub-domains have approximately the same squarish shape or in other words, when the largest group assigned to a processor is as compact as possible.

The results of Figure 12.10 were obtained at early times when the Lagrangian particles are still distributed evenly around the cylinder which makes the domain decomposition an easier task. At later times, the distribution of the vortices does not allow the decomposition of the domain in groups having approximately the same radius and the same number of

Figure 12.11: Load imbalance (solid), communication and synchronization time (dash) and extra work (dot-dash) as a function of the number of processors.

vortices. Some subdomains cover a larger region of space and as a result, the efficiency drops to approximately 0.6. This is mainly due to the fact that more processors end up in the near field of a processor responsible for a large group; the request lists are longer and more data has to be moved between processors.

The sources of overhead corresponding to Figure 12.10 are shown on Figure 12.11 normalized with the useful work. Load imbalance, the largest overhead contributor, is defined as the difference between the maximum useful work reported by a processor and the average useful work per processor. Secondly, the extra work includes the time spent making a copy of one's own data structure, the time required to absorb the returning information and the work that was duplicated in all processors, namely, the search for acceptable interactions in the upper portion of the tree and the subsequent creation of the request lists. The remaining overhead has been lumped under communication time although most of it is probably idle time (or synchronization time) that was not included in the definition of load imbalance.

We expected that as P increases, the near field of a processor would eventually contain a fixed number of neighboring processors. The number of messages and the load imbalance would then reach an asymptote and the loss of efficiency would be driven by the much smaller communication and extra times. However, this has yet to happen at 32 processors and the communication time is already starting to make an impact.

Nevertheless, the fast algorithm, its reasonably efficient parallel imple-

Figure 12.12: Vorticity Field for $Re = 3000$ at $T = 5.0$.

mentation and the speed of the Mark III have made possible simulations with as many as 80,000 vortex particles.

12.5.5 Results

These 80,000 particles were used to compute the flow past an impulsively started cylinder. Figure 12.12 shows the vorticity field after five time units meaning that the cylnder has been displaced by five radii; the Reynolds number is 3000. The pair of primary eddies induced by the body's motion is clearly visible as well as a number of small structures produced by the interaction of the wake with the rear portion of the cylinder. It should be noted that symmetry has been enforced in the simulation. Streamlines derived from this vorticity distribution are presented in Figure 12.13 and compared with Bouard and Coutanceau's flow visualization [Bouard:80a] obtained at the same dimensionless time and Reynolds number.

12.6 Cluster Algorithms for Spin Models

12.6.1 Monte Carlo Calculations of Spin Models

The goal of computer simulations of spin models is to generate configurations of spins typical of statistical equilibrium and measure physical observables on this ensemble of configurations. The generation of configurations is traditionally performed by Monte Carlo methods such as the Metropolis algorithm [Metropolis:53a], which produce configurations with a probability given by

Figure 12.13: Comparison of Computed Streamlines with Bouard and Coutanceau Experimental Flow Visualization at $Re = 3000$ and $T = 5.0$.

the Boltzmann distribution $e^{-\beta S(\phi)}$, where $S(\phi)$ is the action, or energy, of the system in configuration ϕ, and β is the inverse temperature. One of the main problems with these methods in practice is that successive configurations are not statistically independent, but rather are correlated with some autocorrelation time, τ, between effectively independent configurations.

A key feature about traditional (Metropolis-like) Monte Carlo algorithms is that the updates are *local*, that is, one spin at a time is updated, and its new value depends only on the values of spins which affect its contribution to the action, *i.e.*, only on local (usually nearest neighbor) spins. Thus, in a single step of the algorithm, information about the state of a spin is transmitted only to its near neighbors. In order for the system to reach a new effectively independent configuration, this information must travel a distance of order the (static or spatial) correlation length ξ. As the information executes a random walk around the lattice, one would suppose that the autocorrelation time, $\tau \sim \xi^2$. However, in general $\tau \sim \xi^z$, where z is called the dynamical critical exponent. Almost all numerical simulations of spin models have measured $z \approx 2$ for local update algorithms. (See also Sections 4.2, 4.3, 7.3, 12.8, and 15.2).

For a spin model with a phase transition, as the inverse temperature β approaches the critical value, ξ diverges to infinity so that the computational efficiency rapidly goes to zero! This behavior is called critical slowing down (CSD), and until very recently it has plagued Monte Carlo simulations of statistical mechanical systems, in particular spin models, at or near their phase transitions. Recently, however, several new "cluster algorithms" have

been introduced which decrease z dramatically by performing *non-local* spin updates, thus reducing (or even eliminating) CSD and facilitating much more efficient computer simulations.

12.6.2 Cluster Algorithms

The aim of the cluster update algorithms is to find a suitable collection of spins which can be flipped with relatively little cost in energy. We could obtain non-local updating very simply by using the standard Metropolis Monte Carlo algorithm to flip randomly selected bunches of spins, but then the acceptance would be tiny. Therefore, we need a method which picks sensible bunches or clusters of spins to be updated. The first such algorithm was proposed by Swendsen and Wang [Swendsen:87a], and was based on an equivalence between a Potts spin model [Potts:52a], [Wu:82a] and percolation models [Stauffer:78a], [Essam:80a], for which cluster properties play a fundamental role.

The Potts model is a very simple spin model of a ferromagnet, in which the spins can take q different values. The case $q = 2$ is just the well-known Ising model. In the Swendsen and Wang algorithm, clusters of spins are created by introducing bonds between neighboring spins with probability $1 - e^{-\beta}$ if the two spins are the same, and zero if they are not. All such clusters are created and then updated by choosing a random new spin value for each cluster and assigning it to all the spins in that cluster.

A variant of this algorithm, for which only a single cluster is constructed and updated at each sweep, has been proposed by Wolff [Wolff:89a]. The implementation of this algorithm is shown in Figure 12.14 to Figure 12.16, which shows a $q = 3$ Potts model at its critical temperature, with different colors representing the three different spin values. From the starting configuration (Figure 12.14), we choose a site at random, and construct a cluster around it by bonding together neighboring sites with the appropriate probabilities (Figure 12.15). All sites in this cluster are then given the same new spin value, producing the new configuration shown in Figure 12.16, which is obviously far less correlated with the initial configuration than the result of a single Metropolis update (Figure 12.17). Although Wolff's method is probably the best *sequential* cluster algorithm, the Swendsen and Wang algorithm seems to be better suited for parallelization, since it involves the entire lattice rather than just a single cluster. We have, therefore, concentrated our attention on the method of Swendsen and Wang, where *all* the clusters must be identified and labeled.

Figure 12.14: I need a caption for this figure

Figure 12.15: I need a caption for this figure

Figure 12.16: I need a caption for this figure

Figure 12.17: I need a caption for this figure

First we outline a sequential method for labeling clusters, the so-called "ants in the labyrinth" algorithm. The reason for its name is that we can visualize the algorithm as follows [Dewar:87a]. An ant is put somewhere on the lattice, and notes which of the neighboring sites are connected to the site it is on. At the next time-step, this ant places children on each of these connected sites which are not already occupied. The children then proceed to reproduce likewise until the entire cluster is populated. In order to label all the clusters, we start by giving every site a negative label, set the initial cluster label to be zero, and then loop through all the sites in turn. If a site's label is negative, then the site has not already been assigned to a cluster so we place an ant on this site, give it the current cluster label, and let it reproduce passing the label on to all its offspring. When this cluster is identified, we increment the cluster label and carry on repeating the ant-colony birth, growth and death cycle until all the clusters have been identified.

12.6.3 Parallel Cluster Algorithms

As with the percolation models upon which the cluster algorithms are based, the phase transition in a spin model occurs when the clusters of bonded spins become large enough to span the entire lattice. Thus, near criticality (which in most cases is where we want to perform the simulation), clusters come in all sizes, from order N (where N is the number of sites in the lattice) right down to a single site. The highly irregular and non-local nature of the clusters means that cluster update algorithms do not vectorize, and hence give

poor performance on vector machines. On this problem, a CRAY X-MP is only about ten times faster than a Sun 4 workstation. The irregularity of the problem also means that SIMD machines are not well suited to this problem, whereas for the Metropolis type algorithms, they are perhaps the best machines available. It, therefore, appears that the optimum performance for this type of problem will come from MIMD parallel computers.

A parallel cluster algorithm involves distributing the lattice onto an array of processors using the usual domain decomposition. Clearly, a sequential algorithm can be used to label the clusters on each processor, but we need a procedure for converting these labels to their correct *global* values. We need to be able to tell many processors, which may be any distance apart, that some of their clusters are actually the same, to agree on which of the many different local labels for a given cluster should be assigned to be the global cluster label, and to pass this label to all the processors containing a part of that cluster. We have implemented two such algorithms, "self-labeling" and "global equivalencing".

12.6.4 Self-labeling

We shall refer to this algorithm as "self-labeling", since each site figures out which cluster it is in by itself from local information. We begin by assigning each site, i, a unique cluster label, S_i. In practice, this is simply chosen as the position of that site in the lattice. At each step of the algorithm in parallel, every site looks in turn at each of its neighbors in the positive directions. If it is bonded to a neighboring site, n, which has a different cluster label, S_n, then both S_i and S_n are set to the minimum of the two. This is continued until nothing changes, by which time all the clusters will have been labeled with the minimum initial label of all the sites in the cluster. Note that to check termination of the algorithm involves each processor sending a termination flag (finished or not finished) to every other processor after each step, which can become very costly for a large processor array. This is an SIMD algorithm and can, therefore, be run on machines like the AMT DAP and TMC Connection Machine. However, the SIMD nature of these computers leads to very poor load balancing. Most processors end up waiting for the few in the largest cluster which are the last to finish. We implemented this on the AMT DAP and obtained only about 20% efficiency.

We can improve this method on a MIMD machine by using a faster sequential algorithm, such as "ants in the labyrinth", to label the clusters in the sub-lattice on each processor, and then just use self-labeling on the

Figure 12.18: I need a caption for this figure

sites at the edges of each processor to eventually arrive at the global cluster labels. The number of steps required to do the self-labeling will depend on the largest cluster which, at the phase transition, will generally span the entire lattice. The number of self-labeling steps will, therefore, be of the order of the maximum distance between processors, which for a square array of P processors is just $2\sqrt{P}$. Hence, the amount of communication (and calculation) involved in doing the self-labeling, which is proportional to the number of iterations times the perimeter of the sub-lattice, goes like L for an $L \times L$ lattice, whereas the time taken on each processor to do the local cluster labeling goes like the area of the sub-lattice, which is L^2/P. Therefore, as long as L is substantially greater than the number of processors, we can expect to obtain a reasonable speedup. Of course, this algorithm suffers from the same type of load imbalance as does the SIMD version. However, in this case, it is much less severe since most of the work is done with "ants in the labyrinth" which is well load balanced. The speedups obtained on the Symult 2010, for a variety of lattice sizes, are shown in Figure 12.18. The dashed line indicates perfect speedup (*i.e.*, 100% efficiency). The lattice sizes for which we actually need large numbers of processors are of the order of 512^2 or greater, and we can see that running on 64 nodes (or running multiple simulations of 64 nodes each) gives us quite acceptable efficiencies of about 70% for 512^2 and 80% for 1024^2.

Figure 12.19: I need a caption for this figure

12.6.5 Global Equivalencing

In this method, we again use the fastest sequential algorithm to identify the clusters in the sub-lattice on every node. Each node then looks at the labels of sites along the edges of the neighboring nodes in the positive directions, and works out which ones are connected and should be matched up. These lists of "equivalences" are all passed to one of the nodes, which uses an algorithm for finding equivalence classes [Knuth:68a], [Press:86a] (which, in this case, are the global cluster labels) to match up the connected clusters. This node then broadcasts the results back to all the other nodes.

This part of the algorithm is purely sequential, and is thus a potentially disastrous bottleneck for large numbers of processors. It also requires this node to have a large amount of memory in which to store all the labels from every other node. The amount of work involved in doing the global matchup goes like P times the perimeter of the sub-lattice on each node, so that the efficiency should be less than for self-labeling, although we might still expect reasonable speedups if the number of nodes is not extremely large. The speedups obtained on the Symult 2010 for a variety of lattice sizes are shown in Figure 12.19. The figure for 512^2 on 128 nodes is missing due to memory constraints. Global equivalencing gives about the same speedups as self-labeling for small numbers of processors, but as expected self-labeling does much better as the number of nodes increases.

12.6.6 Other Algorithms

Currently, the only other parallel cluster algorithm proposed for spin models is due to Burkitt and Heermann [Burkitt:89a], but it is much more complicated, and less efficient, than the self-labeling algorithm. However, the problem of labeling clusters of spins is very similar to an important problem in image analysis, that of identifying and labeling the connected components in a binary or multi-colored image composed of an array of pixels, and there have been a number of parallel algorithms implemented for this problem [Cypher:89a], [Embrechts:89a], [Lim:87a]. The most promising of these parallel algorithms for spin models has a hierarchical divide-and-conquer approach [Embrechts:89a]. The processor array is divided up into smaller sub-arrays of, for example, 2×2 processors. In each subarray, the processors look at the edges of their neighbors for clusters which are connected across processor boundaries. As in global equivalencing, these equivalences are all passed to one of the nodes of the sub-array, which places them in equivalence classes. The results of these partial matchings are similarly combined on each 4×4 sub-array, and this process is continued until finally all the partial results are merged together on a single processor to give the global cluster values.

Finally, we should mention the trivial parallelization technique of running independent Monte Carlo simulations on different processors. This method works well until the lattice size gets too big to fit into the memory of each node. In the case of the Potts model, for example, only lattices of size less than about 300^2 or 50^3 will fit into 1 Mbyte, though most other spin models are more complicated and more memory intensive. The smaller lattices which are seen to give poor speedups in Figure 12.18 and Figure 12.19 can be run with 100% efficiency in this way. Note, of course, that this requires an MIMD computer. In fact, we have used this method to calculate the dynamical critical exponents of various cluster algorithms for Potts models [Baillie:89l] on lattices up to 128^2. In order to distinguish whether the dynamical critical exponent is small or is in fact zero, we need to go to larger lattices. Thus, we are now simulating lattices of size 256^2 and 512^2 using the self-labeling parallel cluster algorithm, on a number of MIMD computers.

12.6.7 Credits

This research is being performed by C. F. Baillie and P. D. Coddington.

12.7 Sorting

This section discusses sorting: the rearrangement of data into some set sequential order. Sorting is a common component of many applications and so it is important to do it well in parallel. Quicksort (to be discussed below) is fundamentally a divide and conquer algorithm and the parallel version is closely related to the recursive bisection algorithm discussed in Section 11.2. Here, we have concentrated on the best general-purpose sorting algorithms: Bitonic, Shellsort, and Quicksort. No special properties of the list are exploited. If the list to be sorted has special properties, such as a known distribution (e.g., random numbers with a flat distribution between 0 and 1) or high degeneracy (many redundant items, for example, text files), then other strategies can be faster. In the case of known data distribution, a bucketsort strategy (e.g., radix sort) is best, while the case of high degeneracy is best handled by the distribution counting method ([Knuth:73a], pp. 379-81).

The ideas presented here are appropriate for MIMD machines and are somewhat specific to hypercubes (we will assume 2^d processors), but can easily be extended to other topologies.

There are two ways to measure the quality of a concurrent algorithm. The first may be termed "speed at any cost," and here one optimizes for the highest absolute speed possible for a fixed-size problem. The other we can call "speed per unit cost," where one, in addition to speed, worries about efficient use of the parallel machine. It is interesting that in sorting, different algorithms are appropriate depending upon which criterion is employed. If one is interested only in absolute speed, then one should pay for a very large parallel machine and run the bitonic algorithm. This algorithm, however, is inefficient. If efficiency also matters, then one should only buy a much smaller parallel machine and use the much more efficient Shellsort or Quicksort algorithms.

Another way of saying this is: for a *fixed size* parallel computer (the realistic case), Quicksort and Shellsort are actually the *fastest* algorithms on all but the smallest problem sizes. We continue to find the misconception that "Everyone knows that the Bitonic algorithm is fastest for sorting." This is *not* true for most combinations of machine size and list size.

The data are assumed to initially reside throughout the parallel computer, spread out in a random, but load balanced fashion (that is, each processor begins with an approximately equal number of datums). In our experiments, the data were positive integers and the sorting key was taken

to be simply their numeric value. We require that, at the end of the sorting process, the data residing in each node are sorted internally and these sublists are also sorted globally across the machine in some way.

12.7.1 The Merge Strategy

In the merging strategy to be used by our sorting algorithms, the first step is for each processor to sort its own sublist using some fast algorithm. We take for this a combined quicksort / insertion sort which is described in detail as Algorithm Q by Knuth ([Knuth:73a], pp. 118-9). Once the local (processor) sort is complete, it must be decided how to merge all of the sorted lists in order to form one globally sorted list. This is done in a series of compare-exchange steps. In each step, two neighboring processors exchange items so that each processor ends up with a sorted list and all of the items in one processor are greater than all of the items in the other. Thus, two sorted lists of m items each are merged into a sorted list of $2m$ items (stored collectively in the memory of the two processors). The compare-exchange algorithm is interesting in it's own right, but we do not have the space here to discuss it. The reader is referred to Chapter 18 of [Fox:88a] for the details.

12.7.2 The Bitonic Algorithm

Many algorithms for sorting on concurrent machines are based upon Batcher's Bitonic sorting algorithm ([Batcher:68a], [Knuth:73a] pp.232-3). The first step in the merge strategy is for each processor to internally sort via quicksort. One is then left with the problem of constructing a series of compare-exchange steps which will correctly merge $N = 2^d$ sorted sublists. This problem is completely isomorphic to the problem of sorting a list of 2^d items by pairwise comparisons between items. Each one of our sublist compare-exchange operations is equivalent to a single compare-exchange between two individual items. The pattern of compare-exchanges for the Bitonic algorithm for the $d = 3$ case is shown in Figure 12.20. More details and a specification of the Bitonic algorithm can be found in Chapter 18 of [Fox:88a].

Table 12.1 shows the actual times and efficiencies for our implementation of the Bitonic algorithm. Results are shown for sorting lists of sizes 1 k to 2048 k items on hypercubes with dimensions, d, ranging from one (two nodes) to seven (128 nodes). Efficiencies are computed by comparing with single processor times to quicksort the entire list (we take quicksort to be

Figure 12.20: Bitonic scheme for $d = 3$. This figure illustrates the six compare-exchange steps of the bitonic algorithm for $d = 3$. Each diagram illustrates four compare-exchange processes which happen simultaneously. A boldface arrow represents a compare-exchange between two processors. The largest items go to the processor at the point of the arrow, and the smallest items to the one at the base of the arrow.

our benchmark sequential algorithm). The same information is also shown graphically in Figure 12.21.

Clearly, the efficiencies fall off rapidly with increasing d. From the standpoint of cost-effectiveness, this algorithm is a failure. On the other hand, Table 12.1 shows that for fixed list sizes and increasing machine size, the execution times continue to decrease. So, from the speed at any cost point of view, the algorithm is a success. We attribute the inefficiency of the Bitonic algorithm partly to communication overhead and some load imbalance during the compare-exchanges, but mostly to non-optimality of the algorithm itself. In our definition of efficiency we are comparing the parallel Bitonic algorithm to sequential quicksort. In Bitonic, the number of cycles grows quadratically with d. This suggests that efficiency can be improved greatly by using a parallel algorithm that sorts in fewer operations without sacrificing concurrency.

12.7.3 Shellsort or Diminishing Increment Algorithm

This algorithm again follows the merge strategy and is motivated by the fact that d compare-exchanges in the d different directions of the hypercube result in an almost-sorted list. Global order is defined via *ringpos*, that is,

Total List Size*	Execution times (sec) 1 k	2	4	8	16	32	64	128	256	512	1024	2048
d												
1	.25	.54	1.19	2.52	5.50	11.72						
2	.15	.33	.71	1.49	3.24	6.66	13.92					
3	.10	.20	.44	.92	1.95	3.98	8.41	17.56				
4	.07	.14	.29	.60	1.26	.260	5.39	11.31	23.11			
5	.05	.10	.20	.40	.82	1.71	3.50	7.15	14.80	30.34		
6	.03	.06	.13	.26	.53	1.08	2.20	4.48	9.15	18.74	38.26	
7	.03	.04	.09	.17	.33	.67	1.35	2.77	5.60	11.37	23.22	47.00

Total List Size*	Efficiency ε 1 k	2	4	8	16	32	64	128	256	512	1024	2048
d												
1	.92	.94	.93	.95	.94	.94						
2	.77	.77	.78	.80	.80	.83	.85					
3	.58	.63	.63	.65	.66	.69	.70	.71				
4	.41	.45	.48	.50	.51	.53	.55	.55	.57			
5	.29	.32	.35	.37	.39	.40	.42	.44	.45	.46		
6	.24	.26	.27	.29	.30	.32	.34	.35	.36	.37	.39	
7	.12	.20	.19	.22	.24	.26	.27	.28	.30	.31	.32	.33

*In units of 1 k = 1024 items. Divide by 2^d for items per processor.

Table 12.1: Bitonic Sort

Figure 12.21: The efficiency of the bitonic algorithm versus list size for various size cubes.

the list will end up sorted on a embedded ring in the hypercube. After the d compare-exchange stages, the algorithm switches to a simple mopping-up stage which is specially designed for almost-sorted lists. This stage is optimized for moving relatively few items quickly through the machine and amounts to a parallel bucket brigade algorithm. Details and a specification of the parallel Shellsort algorithm can be found in Chapter 18 of [Fox:88a].

It turns out that the mop-up algorithm takes advantage of the MIMD nature of the machine and that this characteristic is crucial to its speed. Only the few items which need to be moved are examined and processed. The Bitonic algorithm, on the other hand, is natural for a SIMD machine. It involves much extra work in order to handle the worst case, which rarely occurs.

We refer to this algorithm as Shellsort ([Shell:59a], [Knuth:73a] pp. 84-5, 102-5) or a diminishing increment algorithm. This is not because it is a strict concurrent implementation of the sequential namesake, but because the algorithms are similar in spirit. The important feature of Shellsort is that in early stages of the sorting process, items take very large jumps through the list reaching their final destinations in few steps. As shown in Figure 12.22, this is exactly what occurs in the concurrent algorithm.

The algorithm was implemented and tested with the same data as the Bitonic case. The timings appear in Table 12.2 and are also shown graphically in Figure 12.23. This algorithm is much more efficient than the Bitonic algorithm, and offers the prospect of reasonable efficiency at large d. The remaining inefficiency is the result of both communication overhead and

Figure 12.22: The parallel Shellsort on a $d = 3$ hypercube. The left side shows what the algorithm looks like on the cube, the right shows the same when the cube is regarded as a ring.

algorithmic non-optimality relative to quicksort. For most list sizes, the mop-up time is a small fraction of the total execution time, though it begins to dominate for very small lists on the largest machine sizes.

12.7.4 Quicksort or Samplesort Algorithm

The classic quicksort algorithm is a divide-and-conquer sorting method ([Hoare:62a], [Knuth:73a] pp.118-23). As such it would seem to be amenable to a concurrent implementation, and with a slight modification (actually, an improvement of the standard algorithm) this turns out to be the case.

The standard algorithm begins by picking some item from the list and using this as the splitting key. A loop is entered which takes the splitting key and finds the point in the list where this item will ultimately end up once the sort is completed. This is the first splitting point. While this is being done, all items in the list which are less than the splitting key are placed on the low side of the splitting point, all higher items are placed on the high side. This completes the first divide. The list has now been broken into two independent lists, each of which still needs to be sorted.

The essential idea of the concurrent (hypercube) quicksort is the same. The first splitting key is chosen (a global step to be described below) and then the entire list is split, in parallel, between two halves of the hypercube. All items higher than the splitting key are sent in one direction in the hypercube, and all items less are sent the other way. The procedure is then called

Total List Size	Execution times (sec) 1 k	2	4	8	16	32	64	128	256	512	1024	2048 k
d												
1	.25	.54	1.20	2.52	5.51	11.73						
2	.15	.31	.68	1.43	3.13	6.40	13.44					
3	.07	.16	.36	.76	1.62	3.30	7.03	14.96				
4	.04	.09	.19	.40	.85	1.82	3.70	7.91	16.66			
5	.05	.07	.11	.23	.44	.91	1.93	4.01	8.47	17.51		
6	.07	.08	.09	.16	.27	.52	1.05	2.12	4.36	9.15	18.81	
7	.21	.16	.16	.17	.23	.37	.67	1.30	2.44	5.23	10.22	21.03

Total List Size	Efficiency ε 1 k	2	4	8	16	32	64	128	256	512	1024	2048 k
d												
1	.92	.94	.92	.95	.94	.94						
2	.77	.82	.81	.84	.82	.86	.88					
3	.82	.79	.77	.79	.80	.84	.84	.84				
4	.72	.70	.73	.75	.76	.76	.80	.79	.80			
5	.29	.45	.63	.65	.73	.76	.76	.78	.78	.80		
6	.10	.20	.38	.47	.60	.66	.70	.74	.76	.77	.78	
7	.02	.05	.11	.22	.35	.47	.55	.60	.68	.67	.72	.74

Table 12.2: Shellsort

Figure 12.23: Same as Figure 12.21, but for the Shellsort algorithm

Figure 12.24: An Illustration of the Parallel Quicksort

recursively, splitting each of the subcubes' lists further. As in Shellsort, the ring-based labeling of the hypercube is used to define global order. Once d splits occur, there remain no further interprocessor splits to do, and the algorithm continues by switching to the internal quicksort mentioned earlier. This is illustrated in Figure 12.24

So far, we have concentrated on standard quicksort. For quicksort to work well, even on sequential machines, it is essential that the splitting points land somewhere near the median of the list. If this isn't true, quicksort behaves poorly, the usual example being the quadratic time that standard quicksort takes on almost sorted lists. To counteract this, it is a good idea to choose the splitting keys with some care so as to make evenhanded splits of the list.

This becomes much more important on the concurrent computer. In this case, if the splits are done haphazardly, not only will an excessive number of operations be necessary, but large load imbalances will also occur. Therefore, in the concurrent algorithm, the splitting keys are chosen with some care. One reasonable way to do this is to randomly sample a subset of the entire list (giving an estimate of the true distribution of the list) and then pick splitting keys based upon this sample. To save time, all $2^d - 1$ splitting keys are found at once. This modified algorithm should perhaps be called samplesort and consists of the following steps:

- each processor picks sample of l items at random
- sort the sample of $l\ 2^d$ items using the parallel Shellsort

Figure 12.25: Efficiency Data for the Parallel Quicksort Described in the Text.

- choose splitting keys as if this was the entire list
- broadcast splitting keys so that all processors know all splitting keys
- perform the splits in the d directions of the hypercube
- each processor quicksorts its sublist

Times and efficiencies for the parallel quicksort algorithm are shown in Table 12.3. The efficiencies are also plotted in Figure 12.25. In some cases, the parallel quicksort out-performs the already high performance of the parallel Shellsort discussed earlier. There are two main sources of inefficiency in this algorithm. The first is a result of the time wasted sorting the sample. The second is due to remaining load imbalance in the splitting phases. By varying the sample size, l, we achieve a trade-off between these two sources of inefficiency. Chapter 18 of [Fox:88a] contains more details regarding the choice of l and other ways to compute splitting points.

Before closing, it may be noted that there exists another way of thinking about the parallel quicksort / samplesort algorithm. It can be regarded as a bucketsort, in which each processor of the hypercube comprises one bucket. In the splitting phase, one attempts to determine reasonable limits for the 2^d buckets so that approximately equal numbers of items will end up in each bucket. The splitting process can be thought of as an optimal routing scheme on the hypercube which brings each item to its correct bucket. So, our version of quicksort is also a bucketsort in which the bucket limits are chosen dynamically to match the properties of the particular input list.

Total List Size	Execution times (sec) 1 k	2	4	8	16	32	64	128	256	512	1024	2048 k
d												
1	.27	.60	1.18	2.51	5.24	11.37						
2	.18	.34	.71	1.41	2.93	6.10	12.71					
3	.12	.20	.43	.83	1.59	3.32	6.69	13.25				
4	.09	.13	.24	.44	.96	1.83	3.80	7.46	15.64			
5	.06	.09	.15	.28	.51	1.02	2.02	3.94	8.26	19.45		
6	.08	.11	.13	.20	.34	.61	1.11	2.12	4.26	10.11	25.87	
7	.20	.16	.18	.20	.24	.45	.76	1.29	2.38	5.59	13.66	37.91

Total List Size	Efficiency ϵ 1 k	2	4	8	16	32	64	128	256	512	1024	2048 k
d												
1	.85	.84	.94	.95	.98	.97						
2	.64	.75	.78	.85	.88	.91	.93					
3	.48	.63	.64	.72	.81	.83	.88	.95				
4	.32	.49	.58	.68	.67	.76	.78	.84	.85			
5	.24	.35	.46	.53	.63	.68	.73	.80	.80	.72		
6	.09	.14	.27	.37	.47	.57	.66	.74	.78	.69	.57	
7	.02	.05	.10	.19	.34	.38	.49	.61	.70	.63	.54	.41

Table 12.3: Quicksort

Credits

The sorting work began as a collaboration between Steve Otto and summer students Ed Felten and Scott Karlin. Ed Felten invented the parallel Shellsort; Felten and Otto developed the parallel Quicksort.

12.8 Exchange Interactions in Solid He_3

*** Contribution needed from G. C. Fox

12.9 Nbody v. Multigrid

*** Contribution needed from D. Edelsohn

Chapter 13

Performance Studies

*** Contribution needed from P. C. Messina

Chapter 14

Data Parallel C and Fortran

14.1 C^3P to CRPC Evolution

*** Contribution needed from G. C. Fox

14.2 A Software Tool for Data Partitioning and Distribution

Programming a distributed memory parallel computer is a complicated task. It involves two basic steps, (1) specifying the partitioning of the data and (2) writing the communication that is necessary in order to preserve the correct data flow and computation order. The former requires some intellectual effort, while the latter is straightforward but tedious work.

We have observed that programmers use several well known tricks to optimize the communication in their programs. Many of these techniques are purely mechanical, relying more on clever juxtapositions and transformations of the code rather than on a deep knowledge of the algorithm. This is not surprising, since once the data domain has been partitioned, the data dependencies in the program completely define the communication necessary between the separate partitions. It should, therefore, be possible for a software tool to automate step (2), once step (1) has been accomplished by the programmer.

This would allow the program to be written in a traditional sequential language extended with annotations for specifying data distribution, and have a software tool or compiler mechanically generate the node program for the distributed memory computer. This strategy, illustrated by stages II

Figure 14.1: The Program Development Process

and iii in Figure 14.1, is being studied by several researchers [Callahan:88d], [Chen:88b], [Koelbel:87a], [Koelbel:90a], [Rogers:89b], [Zima:88a].

What is missing in this scheme? Although the tedious step has been automated, the hard intellectual step of partitioning the data domain is still left entirely to the programmer. The choice of a partitioning strategy often involves some deeper knowledge of the algorithm itself, so we clearly cannot hope to automate this process completely. We could, however, provide some assistance in the data partitioning process, so that the programmer can make a better choice of partitioning schemes from all the available options. This section describes the design of an interactive data partitioning tool that provides exactly this kind of assistance.

14.2.1 Is Any Assistance Really Needed?

The ultimate goal of the programmer is peak performance on the target computer. The realization of peak performance requires the understanding of many subtle relationships between the algorithm, the program and the target machine architecture. Factors such as input data size, data dependencies in the code, target machine characteristics and the data partitioning scheme are related in very non-intuitive ways, and jointly determine the performance of the program. Thus, a data partitioning scheme that is chosen purely on the basis of some algorithmic property, may not always be the best choice.

Let us examine the relationship between these aspects more closely, to illustrate the subtle complexities that are involved in choosing the partitioning of the data domain. Consider the following program:

```
subroutine example (A, B, N)
    double precision A(N, N), B(N, N)
                .  .
    do k=1, cycles
        do j=1,N
            do i=2,N-1
                A(i, j) = F ( B(i-1, j), B(i+1, j) )
            enddo
        enddo
        do j=2,N-1
            do i=2,N-1
                B(i,j) = F' ( A(i-1, j), A(i+1, j), A(i, j),
                    A(i, j-1), A(i, j+1) )
            enddo
        enddo
    enddo
end
```

$\mathcal{F}$ and $\mathcal{F}'$ represent functions with four and ten double precision floating point operations, respectively. This program segment does not represent any particular realistic computation; rather, it was chosen to illustrate all the aspects of our argument using a small piece of code. The program segment was executed on 64 processors of an NCUBE, with array sizes ranging from $N = 64$ to $N = 320$. A and B were first partitioned as columns, so that each processor was assigned $N/64$ successive columns. The program was then run once again, this time with A and B partitioned as blocks, so that each processor was assigned a block of $N^2/64$ elements. The resulting execution and communication times for column and block partitioning schemes are shown in Figure 14.2. The communication time was measured by removing all computation in the loops.

When employing a column partitioning scheme for arrays A and B, communication is only necessary after the first j loop. Each processor has to exchange boundary values with its left and right neighbor. In a block partitioning scheme, each processor has to communicate with its four neighbors after the first loop and with its north and south neighbors after the second loop. For small message lengths, the communication cost is dominated by the message startup time, whereas the transmission cost begins to dominate as the messages get longer (*i.e.*, more data is exchanged at each communication step). This explains why communication cost for the column partition is greater than for the block partition for array sizes larger than 128 x 128. It is clear from the graph that column partitioning is preferable when the

Figure 14.2: Timing results on an NCUBE, using 64 processors.

array sizes are less than 128×128, and block partitioning is preferable for larger sizes.

The steps in the execution time graphs are caused mainly by load imbalance effects. For example, the step between $N = 128$ and $N = 129$ for the column partition is due to the fact that for size 129 one subdomain has an extra column, so that the processor assigned to that subdomain is still busy after all the others have finished, causing load imbalance in the system. Similar behavior can be observed for the block partition, but here the steps occur at smaller increments of the array size N. The steps in the communication time graphs are due to the fact that the packet size on the NCUBE is 1 Kbyte, so that messages that are even a few bytes longer need an extra packet to be transmitted.

The above example indicates that several factors contribute to the observed performance of a chosen partitioning scheme, making it difficult for a human to predict this behavior statically. Our aim is to make the programmer aware of these performance effects without having to run the program on the target computer. We hope to do this by providing an interactive tool, that can give performance estimates in response to a data layout specification. The tool's performance estimates will allow the programmer to gauge the effect of a data partitioning scheme and thus provide some guidance in making a better choice.

14.2.2 Overview of the Tool

When using the tool we envision, the programmer will select a program segment for analysis, and the system will provide assistance in choosing an efficient data partitioning for the computation in that program segment, for various problem sizes. In a first step, the user determines a set of reasonable partitionings based on the data dependence information and interprocedural analysis information provided by the tool. An important component of the system is the performance estimation module, which is subsequently used to select the best partitionings and distributions from among those examined. In the present version, the do loop is the only kind of program segment that can be selected. For simplicity, the set of possible partitions of an array is restricted to regular rectangular patterns such as by row, by column, or by block for a two dimensional array and their higher dimensional analogs for arrays of larger dimensions. This permits the examination of all reasonable partitionings of the data in an acceptable amount of time.

The tool will permit the user to generalize from local partitionings to layouts for an entire program in easy steps, using repartitioning and redistribution whenever it leads to a better performance overall. In addition, the tool will support many program transformations that can lead to more efficient data layouts.

The principal value of such an environment for data partitioning and distribution is that it supports an exploratory programming style in which the user can experiment with different data partitioning strategies, and estimate the effect of each strategy for different input data sizes or different target machines without having to change the program or run the program each time.

14.2.3 Dependence-based Data Partitioning

Given a sequential Fortran program and a selected program segment (which in the preliminary version can only be a loop nest), the tool provides assistance in deriving a set of reasonable data partitions for the arrays accessed in that segment. The assistance is given in the form of data dependence information for variables accessed within the selected segment. When partitioning data, we must ensure that the parallel computations done by all the processors on their local partitions preserve the data dependence relations in the sequential program segment. If the computations done by the processors on the distributed data satisfy all the data dependences, the results of the

computation will be the same as that produced by a sequential execution of the original program segment. There are two ways to achieve this: (1) by "internalizing" data dependences within each partition, so that all values required by computations local to a processor are available in its local data subdomain, or (2) by inserting appropriate communication to get the non-local data.

Let us consider a sample program segment, and see how data dependence information can be used to help derive reasonable data partitionings for the arrays accessed in the segment.

P1. Example program segment.

```
do j = 1, n
    do i = 1, n
        A(i, j) = F( A(i-1, j) )
        B(i, j) = F'( A(i, j), B(i, j-1), B(i, j) )
    enddo
enddo
```

$\mathcal{F}$ and $\mathcal{F}'$ represent arbitrary functions, and their exact nature is irrelevant to this discussion. When the programmer selects the "`do i`" loop, the tool indicates that there is one data dependence that is carried by the `i` loop: the dependence of $A(i,j)$ on $A(i-1,j)$. This dependence indicates that the computation of an element of A cannot be started until the element immediately above it in the previous row has been computed. The programmer then selects the outer "`do j`" loop to get the data dependences that are carried by the `j` loop. There is one such dependence, that of $B(i,j)$ on $B(i,j-1)$. This dependence indicates that the computation of an element of B cannot be started until the computation of the element immediately to the left of it in the previous column has been computed. Figure 14.3(a) illustrates the pattern of data dependences for the above program segment.

The pattern of data dependences between references to elements of an array gives the programmer clues about how to partition the array. It is usually a good strategy to partition an array in a manner that internalizes all data dependences within each partition, so that there is no need to move data between the different partitions that are stored on different processors. This avoids expensive communication via messages. For example, the data dependence of $A(i,j)$ on $A(i-1,j)$ can be satisfied by partitioning A in a column-wise manner, so that the dependences are "internalized" within each partition. The data dependence of $B(i,j)$ on $B(i,j-1)$ can be satisfied by

Figure 14.3: Data dependences satisfied by internalization and communication for the partitioning schemes (a) A by column, B by column (b) A by column, B by row and (c) A by block, B by block. Dotted lines represent partition boundaries and numbers indicate virtual processor ids (the figures are shown for $p = 4$ virtual processors). For clarity, only a few of the dependences are shown.

partitioning B row-wise, since this would internalize the dependences within each partition.

It is not enough to examine only the dependences that arise due to references to the same array. In some cases, the data flow in the program implicitly couples two different arrays together, so that the partitioning of one affects the partitioning of the other. In our example, each point $B(i, j)$ also requires the value $A(i, j)$. We treat this as a special data dependence (3) called a *value* dependence (read "B is value dependent on A"), to distinguish it from the traditional data dependence that is defined only between references to the same array. This value dependence must also be satisfied either by internalization or by communication. Internalization of the value dependence is possible only by partitioning B in the same manner as A, so that each $B(i, j)$ and the $A(i, j)$ value required by it are in the same partition.

Based on the pattern of data dependences in the program segment, the following are a possible list of partitioning choices that can be derived:

(a) Partition A by column and B by column. This satisfies the dependences within A and the value dependences of B on A by internalization but communication is required to satisfy the data dependences within B (Figure 14.3(a)). An analogous case is to partition both A and B by row. This would require communication to satisfy dependences within

A.

(b) Partition A by column and B by row. Dependences within B are now satisfied by internalization, but communication is needed to satisfy the value dependence of B on A (Figure 14.3(b)).

(c) Partition both A and B as two dimensional blocks. This would result in communication to satisfy dependences within both A and B, while the value dependence of B on A is satisfied by internalization (Figure 14.3(c)).

The partitioning of A by row and B by column was not considered among the possible choices because in this scheme, none of the dependences are internalized, thus requiring greater communication compared to (a), (b) or (c). Communication overhead is a major cause of performance degradation on most machines, so a reasonable first choice would be the partitioning scheme that requires the least communication. This can be determined either by analyzing the number of dependences that are cut by the partitioning (indicating the need for communication), or more accurately using the performance estimation module that is described in the next section.

14.2.4 Mapping Data to Processors

For the selected program segment, the programmer picks one of the choices (a)–(c), and specifies the data partitioning via an interface provided by the tool. The tool responds by creating an internal data mapping that specifies the mapping of the data to a set of virtual processors. The number of virtual processors is equal to the number of partitions indicated by the data partitioning. The mapping of the virtual processors onto the physical processors is assumed to be done by the run-time system, and this mapping is unspecified in the software layer. Henceforth, we will use the term "processor" synonymously with "virtual processor". The internal data mapping is used by the performance estimator to compute an estimate of communication and other costs for the program segment. It is also used by the tool to determine the data that needs to be communicated between the processors.

Let us continue with our example program segment, and see how the internal mapping is constructed for partitioning (b), *i.e.*, A partitioned by column and B by row. The data mappings for the other two cases can be constructed in a similar manner. Let A and B be of size $n \times n$ and the number of (virtual) processors be p. For simplicity we assume that p divides n. The following two data mappings are computed:

- A$(1:n, 1:n)$ partitioned by column: Create a virtual array A\$$(1:p)$, where A\$$(k)$ represents the kth column partition of A, *i.e.*, A\$ consists of the elements A$(1:n, (n/p)(k-1)+1:(n/p)k)$. The virtual array is only an internal entity, used within the tool to maintain the mapping of data to (virtual) processors. It does not have any physical storage on the machine. The partition of A represented by A\$$(k)$ is assumed to be mapped onto the kth processor by default.

- B$(1:n, 1:n)$ partitioned by row: Create a virtual array B\$$(1:p)$, where B\$$(k)$ represents the ith row partition of B, *i.e.*, B\$ consists of the elements B$((n/p)(k-1)+1:(n/p)k, 1:n)$. B\$$(k)$ is assigned to the kth processor by default.

The internal data mapping is used to solve the following two problems:

(I) Given a processor q, what part of A is local to it? This is given by the section of A that belongs to the partition A\$$(q)$.

(II) Given a section A$(x1:x2, y1:y2)$, what processors contain elements of this section? This is given by the set of processors $\{q|\text{A\$}(q) \cap \text{A}(x1:x2, y1:y2) \neq \phi\}$.

The values n and p are assumed to be known statically.

A useful technique that we will subsequently use on these sections is called "translation". Translation refers to the conversion of an accessed section computed with respect to a particular loop to the section accessed with respect to an enclosing loop. For example, consider a reference to a two dimensional array within a doubly nested loop. The section of the array accessed within each iteration of the innermost loop is a single element. The same reference when evaluated with respect to the entire inner loop (*i.e.*, all iterations of the inner loop) may access a larger section, such as a column of the array. If we evaluated the reference with respect to the outer loop (*i.e.*, all iterations of the outer loop), we may notice that the reference results in an access of the entire array, in a column-wise manner. Translation is thus a method of converting array sections in terms of enclosing loops, and we will denote this operation by the symbol "$\Uparrow$".

The tool uses (I) to determine which processors should do what computations. The general rule used is: each processor executes only those program statements whose l-values are in its local storage. The l-values computed by a processor are said to be *owned* by the processor. In order to compute an l-value, several r-values may be required, and not all of them may be local

to that processor. The inverse mapping (Π) is used to determine the set of processors that own the desired r-values. These processors must send the r-value they own to the processor that will execute the statement.

The data mapping scheme described above works only for arrays. Scalar variables are assumed to be replicated, *i.e.*, every processor stores a copy of the scalar variable in its local memory. By the rule stated earlier, this implies that any statement that computes the value of a scalar is executed by all the processors.

14.2.5 Communication Analysis and Performance Improvement Transformations

The communication analysis algorithm takes the internal data mappings, the dependence graph, and the loop nesting structure of the specified program segment as its input. For each processor the algorithm determines information about all communications the processor is involved in. We will now illustrate the communication analysis algorithm using the example program segments P1, P2, and P3, where P2 is derived from P1, and P3 from P2, respectively, by a transformation called *loop distribution*.

Substantial performance improvement can be achieved by performing various code transformations on the program segment. For example, the *loop-distribution* transformation [Wolfe:89a] often helps reduce the overhead of communication. Loop-distribution splits a loop into a set of smaller loops, each containing a part of the body of the original loop. Sometimes, this allows communication to be done between the resulting loops, which may be more efficient than doing the communication within the original loop.

Consider the program segment P1. If A is partitioned by column and B by row, communication will be required within the inner loop to satisfy the value dependence of B on A. Each message communicates a single element of A. For small message sizes and a large number of messages, the fraction of communication time taken up by message startup overhead is usually quite large. Thus, program P1 will most likely give poor performance because it involves the communication of a large number of small messages.

However, if we loop-distributed the inner `do i` loop over the two statements, the communication of A from the first `do i` loop to the second `do i` loop can be done between the two new inner loops. This allows each processor to finish computing its entire column partition of A in the first `do i` loop, and then send its part of A to the appropriate processors as larger messages, before starting computation of a partition of B in the second `do`

i loop. This communication is done only once for each iteration of the outer do j loop, *i.e.*, a total of O(n) communication steps. In comparison, program P1 requires communication within the inner loop, which gives a total of $O(n^2)$ communication steps:

P2. After loop-distribution of i loop.

```
do j = 2, n
    do i = 2, n
        A(i, j) = F( A(i−1, j) )
    enddo
    do i = 2, n
        B(i, j) = F'( A(i, j), B(i, j−1), B(i, j) )
    enddo
enddo
```

The reduction in the number of communication steps also results in greater parallelism, since the two inner do i loops can be executed in parallel by all processors, without any communication. This effect is much more dramatic if we apply loop-distribution once more, this time on the outer do j loop:

P3. After loop-distribution of j loop.

```
do j = 2, n
    do i = 2, n
        A(i, j) = F( A(i−1, j) )
    enddo
enddo
do j = 2, n
    do i = 2, n
        B(i, j) = F'( A(i, j), B(i, j−1), B(i, j) )
    enddo
enddo
```

For the same partitioning scheme (*i.e.*, A by column and B by row), we now need only O(1) communication steps, which occur between the two outer do j loops. The computation of A in the first loop can be done in parallel by all processors, since all dependences within A are internalized in the partitions. After that, the required communication is performed to satisfy the value dependence of B on A. Then the computation of B can proceed in parallel, because all dependences within B are internalized in the partitions. The absence of any communication within the loops considerably improves efficiency.

Currently, the tool provides a menu of several program transformations, and the programmer can choose which one to apply. When a particular transformation is chosen by the programmer, the tool responds by automatically performing the transformation on the program segment, and updating all internal information automatically.

14.2.6 Communication Analysis Algorithm

For the sake of illustration, let the size of A and B be 8×8 (*i.e.*, $n = 8$), and let the number of (virtual) processors be $p = 4$. The following is a possible sequence of actions that the programmer could do using the tool.

After examining the data dependences within the program segment as reported by the tool, let us assume that the programmer decides to partition A by column and B by row. The tool computes the internal mapping:

$$\begin{aligned}
&\text{A\$(1)} = \text{A(1:8, 1:2) and B\$(1)} = \text{B(1:2, 1:8).} \\
&\text{A\$(2)} = \text{A(1:8, 3:4) and B\$(2)} = \text{B(3:4, 1:8).} \\
&\text{A\$(3)} = \text{A(1:8, 5:6) and B\$(3)} = \text{B(5:6, 1:8).} \\
&\text{A\$(4)} = \text{A(1:8, 7:8) and B\$(4)} = \text{B(7:8, 1:8).}
\end{aligned}$$

To determine the communication necessary, the tool uses Algorithm COMM, shown in Figure 14.4. For simple partitioning schemes as found in many applications, the communication computed by algorithm COMM can be parameterized by processor number, *i.e.*, evaluated once for an arbitrary processor. In addition, we are also investigating other methods to speed up the algorithm.

Consider program P1 for example. According to algorithm COMM, when the kth processor executes the first statement, the required communication is given by

$$\{(q, \lambda) | \lambda = \text{A\$}(q) \cap \text{A}(i-1, j) \neq \phi\}$$

where the range of i and j are determined by the section of the LHS owned by processor k, in this case $i = 2:8$ and $j = 2(k-1)+1:2k$ (since A is partitioned column-wise). But the partitioning of A ensures that $\forall k$, the data $\text{A}(*, 2(k-1)+1:2k)$ is always local to k. The set of (q, λ) pairs will, therefore, be an empty set for any k. Thus, the execution of the first statement with A partitioned by column requires no communication.

When the kth processor executes the second statement, the communication as computed by Algorithm COMM is given by

Algorithm COMM

Input: The data mapping specified by the chosen partitioning scheme, the dependence graph, and the selected loop.
Output: A set of pairs (q,λ), indicating that processor q must send the section λ of data that it owns, to processor k, and the level at which the communication occurs.

for each processor k **do**
  for each statement **do**
    Let def(X) be the section of the LHS array X that is owned by the kth processor;
    for each RHS array reference Y **do**
      Let use(Y) be the section of the RHS array Y that is needed to compute each element of def(X);
      We need to determine the communication required, if any, to get use(Y) from all processors $q \neq k$;
      Define `commlevel` of a dependence to be:

$$\begin{cases} \text{level of the dependence,} & \text{if it is loop-carried} \\ \text{common nesting level of src and sink of dependence,} & \text{if it is loop-independent} \end{cases}$$

      Let `lmax` = max(`commlevels` of all dependences with Y as sink reference);
      Let $\Uparrow$use(Y) be the section use(Y) "translated" to the level `lmax`;
      Then, the set of all (q,λ) pairs is given by $\{(q,\lambda),\ q \neq k \mid \lambda = Y\$(q) \cap \Uparrow\text{use}(Y) \neq \phi\}$, with the communications occurring at level `lmax`;
    endfor
  endfor
endfor

Figure 14.4: Algorithm to determine the communication induced by the data partitioning scheme.

$$\begin{aligned} & \{(q,\lambda)|\lambda = \mathrm{A\$}(q) \cap \mathrm{A}(i,j) \neq \phi\} \\ \cup\ & \{(q,\lambda)|\lambda = \mathrm{B\$}(q) \cap \mathrm{B}(i,j-1) \neq \phi\} \\ \cup\ & \{(q,\lambda)|\lambda = \mathrm{B\$}(q) \cap \mathrm{B}(i,j) \neq \phi\}. \end{aligned}$$

The ranges of i and j are determined by the section of the LHS that is owned by processor k, in this case $i = 2(k-1)+1 : 2k$ and $j = 2 : 8$ (since B is partitioned row-wise). The second and third terms will be ϕ, because the row partitioning of B ensures that $\forall k$, the data $\mathrm{B}(2(k-1)+1 : 2k, *)$ is always local to k. The first term can be a non-empty set, because processor k owns a column of A (*i.e.*, j in the range $2(k-1)+1 : 2k$)), while the range of j in the first term is $2 : 8$. Thus, communication may be required to get the non-local element of A before the kth processor can proceed with the computation of its $\mathrm{B}(i,j)$. The dependence from the definition of $\mathrm{A}(i,j)$ to its use is loop-independent. Algorithm COMM therefore computes `commlevel`, the common nesting level of the source and sink of the dependence, to be the level of the inner i loop. The section $\mathrm{A}(i,j)$ translated to the level of the inner i loop is simply the single element $\mathrm{A}(i,j)$. Thus, each message communicates this single element and the communication occurs within the inner i loop.

The execution of program P1 results in a large number of messages because each message only communicates a single element of A, and the communication occurs within the inner loop. Message startup and transmission costs are specified by the target machine parameters, and the average cost of each message is determined from the performance model. The tool computes the communication cost by multiplying the number of messages by the average cost of sending a single element message. This cost estimate is returned to the programmer.

Now consider the program P2, with the same partitioning scheme for A and B. When the kth processor executes the first statement, the required communication as determined by Algorithm COMM is given by

$$\{(q,\lambda)|\lambda = \mathrm{A\$}(q) \cap \mathrm{A}(1:7,j) \neq \phi\}$$

where the range of j is determined by the section of the LHS owned by processor k, in this case $j = 2(k-1)+1 : 2k$ (since A is partitioned column-wise). Note that in this case, $\Uparrow \mathrm{A}(i-1,j) = \mathrm{A}(1:7,j)$. This is because `commlevel` is now the level of the outer j loop, so that the section $\mathrm{A}(i-1,j)$ must be translated to the level of the j loop. In other words, the reference to $\mathrm{A}(i-1,j)$ in the first statement results in an access of the first seven elements of the jth column of A, during each iteration of the j loop. Since A is partitioned column-wise, this section will always be available locally

in each processor, so that the above set is empty and no communication is required.

When processor k executes the second statement, the communication required is given by

$$\begin{array}{ll} & \{(q,\lambda)|\lambda = \mathrm{A\$}(q) \cap \mathrm{A}(2(k-1)+1:2k, j) \neq \phi\} \\ \cup & \{(q,\lambda)|\lambda = \mathrm{B\$}(q) \cap \mathrm{B}(2(k-1)+1:2k, j-1) \neq \phi\} \\ \cup & \{(q,\lambda)|\lambda = \mathrm{B\$}(q) \cap \mathrm{B}(2(k-1)+1:2k, j) \neq \phi\} \end{array}$$

The second and third terms will be empty sets since the required part of B is local to each k (because B is partitioned row-wise). The first term will be non-empty, because each processor owns $\mathrm{A}(*, 2(k-1)+1 : 2k)$, and the range of j in the first term is outside the range $2(k-1)+1 : 2k$. The data required by processor k from processor q will therefore be a strip $\mathrm{A}(2(k-1)+1:2k, j)$, from each $q \neq k$.

This data can be communicated between the two inner `do i` loops. Each message will communicate a 2×1 size strip of A. Fewer exchanges will be required compared to program P1, because each exchange now communicates a strip of A, and the communication occurs outside the inner loop. Once again, the performance model and target machine parameters are used by the tool to estimate the total communication cost, and this cost is returned to the programmer.

For most target machines, the communication cost in program P2 will be considerably less than in program P1, because of larger message size and fewer messages.

Next, let us consider program P3. Assuming that the same partitioning scheme is used for A and B, the execution of the first loop by the kth processor will require communication given by

$$\{(q,\lambda)|\lambda = \mathrm{A\$}(q) \cap \mathrm{A}(1:7, 2(k-1)+1:2k)\}.$$

But this is an empty set because of the column partitioning of A. Here $\Uparrow \mathrm{A}(i-1, j) = \mathrm{A}(1:7, 2(k-1)+1:2k)$, because `commlevel` for this case is the level of the subroutine that contains the two loops. The section is, therefore, translated to this level by substituting the appropriate bounds for i and j. The translated section indicates that the reference $\mathrm{A}(i-1, j)$ in the first statement results in an access of the section $\mathrm{A}(1:7, 2(k-1)+1:2k)$ during all iterations of the outer j loop that are executed by processor k.

When the kth virtual processor executes the second loop, the required communication is

Figure 14.5: Timing results for programs P1, P2 and P3 on the NCUBE, using 16 processors.

$$\begin{array}{ll} & \{(q,\lambda)|\lambda = \mathrm{A\$}(q) \cap \mathrm{A}(2(k-1)+1:2k, 2:8) \neq \phi\} \\ \cup & \{(q,\lambda)|\lambda = \mathrm{B\$}(q) \cap \mathrm{B}(2(k-1)+1:2k, 1:7) \neq \phi\} \\ \cup & \{(q,\lambda)|\lambda = \mathrm{B\$}(q) \cap \mathrm{B}(2(k-1)+1:2k, 2:8) \neq \phi\}. \end{array}$$

The second and third terms will be empty sets because of the row partitioning of B. The first term will be non-empty, and the data required by processor k from processor q will be the block $\mathrm{A}(2(k-1)+1:2k, 2(q-1)+1:2q)$, for each $q \neq k$. This block can be communicated between the two `do j` loops.

This communication can be done between the two loops, allowing computation within each of the two loops to proceed in parallel. The number of messages is the fewest for this case because a 2×2 block of A is communicated during each exchange. Program P3 is thus likely to give superior performance compared to P1 or P2, on most machines. We ran programs P1, P2 and P3 with A partitioned by column and B by row, on 16 processors of the NCUBE at Caltech. The functions $\mathcal{F}$ and $\mathcal{F}'$ consisted of one and two double precision floating point operations, respectively. The results of the experiment are shown in Figure 14.5. The graphs clearly illustrate the performance improvement that occurs due to reduction in number of messages and increase in length of each message.

14.2.7 Static Performance Estimator

Given the results of the communication analysis in a program segment, the performance estimator can be used to predict the performance of that pro-

gram segment on the target machine. The realization of such an estimator requires a simple static model of performance that is based on (1) target machine parameters such as the number of processors, the message startup and transmission costs and the average times to perform different floating point operations, (2) the size of the input data set, and (3) the data partitioning scheme.

We undertook a study of published performance models [Chen:88b], [Fox:88a], [Gustafson:88a], [Saltz:87b] for use in the performance estimator, and noticed that these theoretical models did not give accurate predictions in many cases. We concluded that the theoretical models suffered from the following deficiencies:

1. Most of the models suggested in the literature were aimed at being "general-purpose", *i.e.*, intended to model the performance of any distributed memory MIMD computer. This generality created problems in some cases, when machine-specific peculiarities tended to skew the observed results from the ones predicted by the model.

2. The models also did not account for all of the software overhead involved in implementing the low-level communication utilities on the machine. While the models accounted for things like message startup costs and packetizing costs, they often ignored factors such as internal buffer sizes, peculiarities of the algorithms used to implement the message passing protocols, etc..

Our effort to correct these defects resulted in an increased complexity of the model, and also necessitated the introduction of several machine-specific features. We felt that this was undesirable, and decided to investigate alternative methods [Balasundaram:90d].

We constructed a program that tested a series of communication patterns using a set of basic low-level portable communication utilities. This program, called a "training set", is executed once on the target machine. The program computes timings for the different communication operations, and averages them over all the processors. These timings are determined for a sequence of increasing data sizes. Since the graph of communication cost versus data size is usually a linear function, it can easily be described by specifying a few parameters (such as the slope, etc.). The training set thus generates a table, whose entries contain the minimal information necessary to completely define the performance characteristic for each communication utility. This

table is used in place of the theoretical model for the purposes of performance prediction.

Figure 14.6 shows some communication cost characteristics created using a part of our training set on 32 processors of an NCUBE. The data space was assumed to be a two dimensional array that was partitioned column-wise, *i.e.*, each processor was assigned a set of consecutive columns. The communication utilities tested here are:

1. `iSR`: nearest neighbor individual element send and receive, using the EXPRESS calls exwrite() and exread().

2. `vSR`: nearest neighbor vector send and receive along one direction, using the EXPRESS calls exvwrite() and exvread().

3. `EXCH1`: nearest neighbor vector exchange along one direction, using the EXPRESS call exvchange().

4. `vSRSR`: nearest neighbor vector sends and receives along two directions, using the EXPRESS calls exvwrite() and exvread().

5. `EXCH2`: nearest neighbor vector exchange along two directions, using two calls to exvchange().

6. `COMBN`: combine operation over all processors, using the EXPRESS call excombine().

7. `BCAST`: one to all broadcast, using the EXPRESS call exbroadcast().

The table generated by the training set for the characteristics shown in Figure 14.6 is:

comm type	startup overhead (msec)	slope	step size (msec)
iSR	11.5	3.58	—
vSR	8.0	0.77	13.0
EXCH1	1.0	0.64	0.0
vSRSR	16.0	1.54	26.0
EXCH2	2.0	1.15	0.0
COMBN	5.2	2.05	—
BCAST	3.0	0.51	6.0

Figure 14.6: Communication cost characteristics of some EXPRESS utilities on the NCUBE

The communication cost estimate for a particular data size is then calculated using the formula:

$$T^{\text{comm}} = \text{startup overhead} + (x * \text{slope}) + (\text{step size} * \lfloor \text{msg length/pkt size} \rfloor)$$

where "pkt size" is the size of each message packet, which on the NCUBE is 1024 bytes (1 Kbyte).

The static performance model is meant primarily to help the programmer discriminate between different data partitioning schemes. Our approach is to provide the programmer with the necessary tools to experiment with several data partitioning strategies, until he can converge on the one that is likely to give him a satisfactory performance. The tool provides feedback information about performance estimates each time a partitioning is done by the programmer.

14.2.8 Conclusion

Our emphasis in this work has been to try to recognize *collective* communication patterns rather than generating sequences of individual element sends and receives. Algorithm COMM determines this in a very natural way. This is especially important for *loosely synchronous problems* which represent a large class of scientific computations [Fox:88a]. Several communication utilities have been developed that provide optimal message passing communication for such problems, provided the communication is of a regular nature and occurs collectively [Fox:88h].

We believe that our approach can be extended to derive partitioning schemes automatically. Data dependence and other information can be used to compute a fairly restricted set of reasonable data partitioning schemes for a selected program segment. The performance estimation module can then be applied in turn to each of the partitionings in the computed set.

The work described in this section was a joint effort between Caltech and Rice University, as part of the Center for Research on Parallel Computation (CRPC) research collaboration [Balasundaram:90a]. The principal researchers were Vasanth Balasundaram and Geoffrey Fox at Caltech, and Ken Kennedy and Ulrich Kremer at Rice. The data partitioning tool described here is being implemented as part of the ParaScope parallel programming environment under development at Rice University [Balasundaram:89c].

14.3 Fortran 90 Climate Experiment

*** Contribution needed from G. C. Fox

14.4 Optimizing Compilers by Neural Networks

The ability of neural networks to compute solutions to optimization problems has been widely appreciated since Hopfield and Tank's work on the traveling salesman problem [Hopfield:85b]. We have examined whether neural network optimization can be usefully applied to compiler optimizations. The problem is nontrivial because compiler optimizations usually involve intricate logical reasoning, but we were able to find an elegant formalism for turning a set of logical constraints into a neural network [Fox:89l]. However, our conclusions were that the method will only be viable if and when large hierarchically structured neural network can be built. The neural approach to compiler optimization is worth pursuing, because such a compiler would not be limited to a finite set of code transformations, and could handle unusual code routinely. Also, if the neural network were implemented in hardware, a processor could perform the optimizations at run time on small windows of code.

Figure Figure 14.7 shows how a simple computation of $A = A * B + C$ is scheduled by a neural network. The machine state is represented at five consecutive cycles by five sets of 20 neurons. The relevant portion of the machine comprises the three storage locations a, b, c and a register r, and the machine state is defined by showing which of the five quantities A, B,

Figure 14.7: A neural network can represent machine states (top) and generate correct machine code for simple computations (bottom).

C, $A*B$ and $A*B+C$ ocupies which location. A firing neuron (i.e., shaded block) in row $A*B$ and column r indicates that $A*B$ is in the register. The neural network is set up to "know" that computations can only be done in the register, and it produces a firing pattern representing a correct computation of $A = A*B+C$.

14.4.1 Credits and C3P References

The neural compiler was conceived by Geoffrey Fox and investigated by Jeff Koller.

14.4.2 C3P References

[Koller:88c], [Fox:89l], [Fox:89q]

14.5 ASPAR: Parallel Processing for a Wider Audience

It is now a widely accepted fact that parallel computing is a successful technology. It has been applied to problems in many fields and has achieved excellent results on projects ranging in scope from academic demonstrations up through complete commercial applications, as shown by other sections of this book.

Despite this success, however, parallel computing is still considered something of a "black-art" to be undertaken only by those with intimate knowledge of hardware, software, physics, computer science and a wealth of other complex areas. To the uninitiated there is something frightening about the strange incantations that abound in parallel processing circles—not just the "buzz-words" that come up in polite conversation but the complex operations carried out on a once elegant piece of sequential code in order for it to successfully run on a parallel processing system.

14.5.1 Degrees of Difficulty

It is easy to define various "degrees of difficulty" in parallel processing. One such taxonomy might be as follows:

i) Extremely difficult

In this category fall the complex, asynchronous, real-time applications. A good example of such a beast is "parallel chess" [Felten:88h] where AI heuristics must be combined with real-time constraints to solve the ill-posed problem of searching the "tree" of potential moves.

ii) Complex

In this area one might put the very large applications of fairly straightforward science. Often algorithms must be carefully constructed but the greatest single problem is the large scale of the overall system and the fact that different "modules" must be integrated into the complete system. An example might be the SDI simulation "Sim88" and its successors [Meier:90a]. The parallel processing issues in such a code require careful thought but pose no insurmountable problems.

iii) Hard

Problems such as large scale fluid dynamics or oceanography [Ghil:90a] often have complex physics but fairly straightforward and well-known numerical methods. In these cases the majority of the work involved in parallelization comes from analysis of the individual algorithms which can then often be parallelized separately. Each sub-module is then a simpler, tractable, problem which often has a "well-known" parallel implementation.

iv) Straightforward but Tedious

The simplest class of "interesting" parallel programs are partial differential equations [Brooks:82b], [Fox:88a]. In these cases the parallel processing issues are essentially trivial but the successful implementation of the algorithm still requires some care to get the details correct.

v) Trivial

The last class of problems are those with "embarrassing parallelism"—essentially uncoupled loop iterations or functional units. In these cases the parallel processing issues are again trivial but the code still requires care if it is to work correctly in all cases.

The "bottom line" from this type of analysis is that all but the hardest cases pose problems in parallelization which are, at least conceptually, straightforward. Unfortunately the actual practice of turning such concepts into working code is never trivial, and rarely easy. At best it is usually an error-prone and time consuming task.

This is the basic reason for ASPAR's existence. Experience has taught us that the complexities of parallel processing are really due not to any inherent problems but to the fact that human beings and parallel computers don't speak the same language. While a human can usually explain a parallel algorithm on a piece of paper with great ease it is often a significant task to convert that picture to functional code. It is our hope that the bulk of the work can be automated by the use of "parallelizing" technologies such as ASPAR. In particular we believe (and our results so far bear out this belief) that problems in all the previous categories, (except possibly a)), can be either completely or significantly automated.

14.5.2 Various Parallelizing Technologies

To understand the issues involved in parallelizing codes and the difference between ASPAR and other similar tools we must examine two basic issues involved in parallelizing code; the local and the global view.

The local view of a piece of code is restricted to one or more loops or similar constructs upon which particular optimizations are to be applied. In this case little attention is paid to the larger scale of the application.

The global view of the program is one in which the characteristics of a particular piece of data or a function are viewed as a part of the complete algorithm. The impact of operating on one item is then considered in the context of the entire application. We believe that ASPAR offers a completely new approach to both views.

Figure 14.8: Vectorizability Analysis

14.5.3 The Local View

"Local" optimization is a method which has been used in compilers for many years and whose principles are well understood. We can understand the evolutionary path to "parallelizing compilers" as follows.

a) Vectorizing Compilers

The goal of automatic parallelization is obviously not new, just as parallel processors are not new. In the past providing support for advanced technologies was in the realm of the compiler which assumed the onus, for example, of hiding vectorizing hardware from the innocent users.

To perform these tasks typically involves a fairly simple line of thought shown by the "flow diagram," Figure 14.8. Basically the simplest idea is to analyze the dependencies between data objects within loops. If there are no dependencies then "kick" the vectorizer into performing all, or as many as it can handle, of the loop iterations at once. Classic vector code, therefore has the appearance

```
   DO 10 I=1,10000
            A(I) = B(I) + C(I)*D(I)
10 CONTINUE
```

b) Parallelizing Compilers

Figure 14.9: A Parallelizing Compiler

We can easily derive parallelizing compilers from this type of technology by changing the box marked "vectorize" to one marked "parallelize". After all, if the loop iterations are independent, parallel operation is straightforward. Even better results can often be achieved by adding a set of "restructuring operations" to the analysis as shown in Figure 14.9.

The idea here is to perform complex "code transformations" on cases which fail to be independent during the first dependency analysis in an attempt to find a version of the same algorithm with fewer dependencies. This type of technique is similar to other compiler optimizations such as loop-unrolling and code-inlining [Zima:88a], [Whiteside:88a]. It's goal is to create new code which produces exactly the same result when executed but which allows for better optimization and, in this case, parallelization.

c) ASPAR

The emphasis of the two previous techniques is still on producing exactly the same result in both sequential and parallel codes. They also rely heavily on sophisticated compiler technology to reach their goals.

ASPAR takes a rather different approach. One of its first assumptions is that it may be OK for the sequential and parallel codes to give different answers!

In technical terms this assumption removes the requirement that loop iterations be independent before parallelization can occur. In practical terms

Figure 14.10: A Parallelizing Compiler

we can best understand this issue by considering a simple example: image analysis.

One of the fundamental operations of image analysis is "convolution". The basic idea is to take an image and replace each pixel value by an average of its neighbors. In the simplest case we end up with an algorithm that looks like

```
   DO 10 I = 2,N-1
   DO 20 J = 2,N-1
               A(I,J) = 0.25*(
                     A(I+1,J) + A(I-1,J) +
                     A(I,J+1) + A(I,J-1))
20 CONTINUE
10 CONTINUE
```

To make this example complete we show, in Figure 14.10, the results of applying this operation to an extremely small (integer valued) image.

It is crucial to note that the results of this operation are not as trivial as one might naively expect. Consider the value at the point ($I = 3$, $J = 3$) which has the original value 52. To compute this value we are instructed to add the values are locations $A(2,2)$, $A(2,4)$, $A(3,2)$ and $A(3,4)$. If we looked only at the original data, from the top of the figure, we might then conclude that the correct answer is $(46 + 87 + 44 + 53)/4 = 57$.

Note that the source code, however, modifies the array A while simultaneously using its values. As a result the above calculation accesses the

Figure 14.11: Data Distributed for Four Processors

correct array elements, but by the time we get around to computing the value at $(3,3)$ the values to the left and above have already been changed by previous loop iterations. As a result the correct value at $(3,3)$ is given by $(\underline{34} + 87 + \underline{61} + 53)/4 = 58$, where the underlined values are those which have been calculated on previous loop iterations.

Obviously this is no problem for a sequential program because the algorithm, as stated in the source code, is translated correctly to machine code by the compiler which has no trouble executing the correct sequence of operations; the problems with this code arise, however, when we consider its parallelization.

The most obvious parallelization strategy is to simply partition the values to be updated among the available processors. Consider, for example, a version of this algorithm parallelized for four nodes.

Initially we divide up the original array A by assigning a quadrant to each processor. This gives the situation shown in Figure 14.11. If we divide up the loop iterations in the same way we see that the process with the top-left corner of the array is to compute $(17 + 28 + 46 + 44)/4$ where the first two values are in its quadrant and the others lie to the right and below the processor boundary. This is not too much of a problem—on a shared memory machine we would merely access the value "44" directly while on a distributed memory machine a message might be needed to transfer the value to our node. In neither case is the procedure very complex: especially since we are having the compiler or parallelizer do the actual communication for us.

The first problem comes in the processor responsible for the data in the top-right quadrant. Here we have to compute $(??+81+80+52)/4$ where the values "80" and "81" are local and the value "52" is in another processors quadrant and therefore subject to the same issues are just described for the top-left processor.

The crucial issue surrounds the value "??" in the previous expression. According to the sequential algorithm this processor should wait for the top-left node to compute its value and then use this new result to compute the new data in the top-right quadrant. Of course this represents a serialization of the worst kind, especially when a few moments though shows that this delay propagates through the other processors too! The end result is that no benefit is gained from parallelizing the algorithm.

Of course this is not the way image analysis (or any of the other fields with similar underlying principles such as PDE's, Fluid mechanics, etc.) is done in parallel. The key fact which allows us to parallelize this type of code despite the dependencies is the observation that: *A large number of sequential algorithms contain data dependencies that are not crucial to the correct "physical" results of the application.*

In this case the data dependency that appears to prevent parallelization is also present in the sequential code but is typically irrelevant. This is not to say that it's effects are not present but merely that the large scale behavior of our application is unchanged by ignoring it. In this case, therefore, we allow the processor with the top-right quadrant of the image to use the "old" value of the cells to its left while computing new values, even though the processor with the top-left quadrant is actively engaged in updating them at the very same time that we are using them!

While this discussion has centered on a particular type of application and the intricacies of parallelizing it, the arguments and features are common to an enormous range of applications. For this reason ASPAR works from a very different point of view than "parallelizing" compilers—its most important role is to find data dependencies of the form just described - and break them! In doing this we apply methods that are often described as *stencil* techniques.

In this approach we try to identify a relationship between a new data value and the old values which it uses during computation. This method is much more general than simple dependency analysis and leads to a correspondingly higher success rate in parallelizing programs. The basic flow of ASPAR's deliberations might, therefore, be summarized in Figure 14.12.

Figure 14.12: ASPAR's Decision Structure

It is important to note that ASPAR provides options to enforce strict dependency checking as well as to override "stencil-like" dependencies. By adopting this philosophy of checking for simple types of dependencies ASPAR more nearly duplicates the way humans address the issue of parallelization and this leads to its greater success. The use of advanced compilation techniques could also be useful, however, and there is no reason why ASPAR should "give up" at the point labelled "Sequential" in Figure 14.12. A similar "loopback" via code restructuring as shown in Figure 14.9 would also be possible in this scenario and would probably yield good results.

14.5.4 The "Global" View

Up to now the discussion has rested mainly on the properties of small portions of code—often single or a single group of nested loops in practical cases. While this is generally sufficient for a "vectorizing" compiler it is too little for effective parallelization. To make the issues a little clearer consider the following piece of code:

```
   DO 10 I=1,100
            A(I) = B(I) + C(I)
10 CONTINUE
   DO 20 I=1,100
            D(I) = B(I) + C(100-I+1)
20 CONTINUE
```

Taken in isolation (The local view) both of these loop constructs are trivially parallelizable and have no dependencies. For the first loop we would

assign the first few values of the arrays A, B and C to the first processor, the next few to the second and so on until we had accounted for each loop iteration. For the second loop we would assign the first few elements of A and B and the last few of C to the first node, and so on. Unfortunately there is a conflict here in that one loop wants to assign values from array C in increasing order while the other wants them in decreasing order. This is the *global decomposition* problem.

The simplest solution, in this particular case, can be derived from the fact that array C only appears on the right hand side of the two sets of expressions. Thus we can avoid the problem altogether by not distributing array C at all. In this case we have to perform a few index calculations but we can still achieve good speed-up in parallel.

Unfortunately life is not usually as simple as presented in this case. In typical codes we would find the logic which led to the "non-distribution" of array C would gradually spread out to the other data structures with the end result that we end up distributing nothing and often failing to achieve any speed-up at all.

14.5.5 Global Strategies

Addressing the global decomposition problem poses problems of a much more serious nature than the previous dependency analysis and local stencil methods because while many clever compiler-related tricks are known to help the local problems there is little theoretical analysis of more global problems. It is very recent, for example, to find compilers that perform any kind of inter-procedural analysis at all.

As a result the resolution of this problem is really one which concerns the parallel programming model available to the parallelization tools. Again ASPAR is unique in this respect.

To understand some of the possibilities it is again useful to create a classification scheme for global decomposition strategies. It is interesting to note that, in some sense, the complexity of these strategies is closely related to our initial comments about the "degree of difficulty" of parallel processing.

a) Functional Decomposition

This style is the simplest of all. We have a situation in which there are no data dependencies among functional units other than initial input and final output. Furthermore each "function" can proceed independently of the others. The global decomposition problem is solved by

virtue of never having appeared at all.

In this type of situation the runtime requirements of the parallel processing system are quite small—typically a "send" and "receive" paradigm is adequate to implement a "master-slave" processing scenario. This is the approach typified by systems such as Linda [Padua:86a] and Strand [Foster:90a].

Of course there are occasional complexities involved in this style of programming such as the use of "broadcast" or data reduction techniques to simplify common operations. For this reason higher level systems such as Express are often easier to use than their "simpler" contemporaries since they include standard mechanisms for performing commonly occurring operations.

b) Global Static Decomposition

This type of application is typified by areas such as numerical integration or convolution operations similar to those previously described.

Their characteristic is that while there are data dependencies among program elements these can be analyzed symbolically at compile time and catered for by suitable insertion of calls to a message passing (for distributed memory) or locking/unlocking (for shared memory) library.

In the convolution case, for example, we provide calls which would arrange for the distribution of data values among processors and the communication of the "boundary strip" which is required for updates to the local elements.

In the integration example we would require routines to sum up contributions to the overall integral computed in each node. For this type of application only simple runtime primitives are required.

c) Global, "oscillating" Decompositions

Problems such as those encountered in large scale scientific applications typically have behavior in which the global decomposition schemes for data objects vary in some standard manner throughout the execution of the program, but in a deterministic way which can be analyzed at compile-time: one routine might require an array to be distributed row-by-row, for example, while another might require the same array to be partitioned by columns or perhaps not at all.

These issues can be dealt with during the parallelization process but require much more sophisticated runtime support than those previously described. Particularly if the resulting programs are to scale well on larger numbers of nodes it is essential that runtime routines be supplied to efficiently perform matrix transposition or boundary cell exchange or global concatenation operations. For ASPAR these operations are provided by the Express runtime system.

14.5.6 The Hard Case

The three categories of decomposition described so far can deal with a significant part of a large majority of "real" applications. By this we mean that good implementations of the various dependency analysis, dependency "breaking" and runtime support systems can correctly parallelize 90% of the code in any application that is amenable to automatic parallelization. Unfortunately this is not really good enough.

Our real goal in setting out to produce automatic parallelization tools is to relieve the user of the burden of performing tricky manipulations by hand. Almost by definition, the 10% of each application left over by the application of the techniques described so far is the most complex part and probably represents about 95% of the complexity in parallelizing the original code by hand! So, at this point, all we have achieved is the automatic conversion of the really easy parts of the code, probably at the expense of introducing messy computer generated code which makes [illegible] understanding of the remaining 10

The solution to this problem comes from the adoption of a much more sophisticated picture of the runtime environment.

14.5.7 Dynamic Data Distribution

The three decomposition methods already described suffer from the defect that they are all implemented, except in detail, during the compile-time "parallelization" of the original program. Thus, while the particular details of "which column to send to which other processor" and similar may be deferred to the runtime support the overall strategy is determined from static analysis of the sequential source code. ASPAR's method is entirely different.

Instead of enforcing global decomposition rules based on static evaluation of the code ASPAR leaves all the decisions about global decomposition to the runtime system and offers only hints as to possible optimizations, whenever

they can safely be determined from static analysis. As a result ASPAR's view of the previously troublesome code would be something along the lines of

```
C- I need B and C to be distributed in increasing order.
                DO 10 I=1,100
                A(I) = B(I) + C(I)
10      CONTINUE
C- I need B to increase and C to decrease.
        DO 20 I=1,100
                D(I) = B(I) + C(100-I)
20      CONTINUE
```

where the "comments" correspond to ASPAR's hints to the runtime support.

The advantages of such an approach are extraordinary. Instead of being stymied by complex, dynamically changing decomposition strategies ASPAR proceeds irrespective and merely expects that the runtime support will be smart enough to provide whatever data will be required for a particular operation.

As a result of this simplification in philosophy ASPAR is able to successfully parallelize practically 100parallelized at all, with no user intervention.

14.5.8 Conclusions

The success of ASPAR relies on two crucial pieces of technology

- The ability to recognize and work around "irrelevant" data dependencies by virtue of "stencils",
- Avoiding global decomposition issues by invoking a sophisticated runtime support.

It is interesting that neither of these is the result of any extensions to existing compiler technology but are derived from our experience with parallel computers. This is consistent with our underlying philosophy of having ASPAR duplicate the methods which real programmers used to successfully parallelize code by hand. Obviously not all problems are amenable to this type of automatic parallelization but we believe that of the cases discussed in the opening paragraphs of this document we can usefully address all but the "Extremely Difficult."

In the simpler cases we believe that the goal of eliminating the role of "human error" in generating correctly functioning parallel code has been accomplished.

The price that has been paid, of course, is the requirement for extremely smart runtime systems. The use of Express as the underlying mechanism for ASPAR has proved its value in addressing the simpler types of decomposition scheme.

The development of the dynamic data distribution mechanisms required to support the more complex applications has led to a completely new way of writing, debugging and optimizing parallel programs which we believe will become the cornerstone of the next generation of Express systems and may revolutionize the ways in which people think about parallel processing.

14.6 Coherent Parallel C

Coherent Parallel C (CPC) was originally motivated by the fact that for many parallel algorithms, the Connection Machine can be very easy to program.

The CPC language is not simply a C with parallel `for` loops; instead, a data parallel programming model is adopted. This means that one has an entire process for each data object. An example of an "object" is one mesh point in a finite element solver. How the processes are actually distributed on a parallel machine is transparent—the user is to imagine that an entire processor is dedicated to each process. This simplifies programming tremendously: complex `if` statements associated with domain boundaries disappear; problems which do not exactly match the machine size and irregular boundaries are all handled transparently. Figure 14.13 illustrates CPC by contrasting "normal" hypercube programming with CPC programming for a simple grid-update algorithm.

The usual communication calls are not seen at all at the user level. Variables of other processes (which may or may not be on another processor) are merely accessed, giving global memory. In our Ncube implementation, this was implemented using the efficient global communications system called the *crystal_router* (see Chapter 22 of [Fox:88a]).

An actual run-time system was developed for the Ncube and is described in [Felten:88a]. Much work remains to be done, of course. How to optimize so as to produce an efficient communications traffic is unexplored; a serious attempt to produce a fine-grained, MIMD machine really involves new types

Figure 14.13: Normal Hypercube Programming Model versus CPC Model for the Canonical Grid-based Problem. The upper part of the figure shows a two-dimensional grid upon which the variables of the problem live. The middle portion shows the usual hypercube model for this type of problem. There is one process per processor and it contains a subgrid. Some variables of the subgrid are on a process boundary, some are not. Drawn explicitly are communication buffers and the channels between them which must be managed by the programmer. The bottom portion of the figure shows the CPC view of the same problem. There is one data object (a grid point) for each process so that all variables are on a process boundary. The router provides a full interconnect between the processes.

of hardware, somewhat like Dally's J-machine.

14.6.1 Credits

Ed Felten and Steve Otto

14.7 Fortran D/C* — The Future?

*** Contribution needed from G. C. Fox

Chapter 15

Asynchronous Applications

15.1 Problem Architecture (heterogeneous)

*** Contribution needed from G. C. Fox

15.2 Melting in Two Dimensions

15.2.1 Problem Description

Although we live in a three-dimensional world, many important processes involve interactions on surfaces, which are effectively two-dimensional. While experimental studies of two-dimensional systems have been successful in probing some aspects of such systems, computer simulation is another powerful tool that can be used to measure their properties. We have used a computer simulation to study the melting transition of a two-dimensional system of interacting particles [Johnson:86a], [Johnson:86b]. One purpose of the study is to investigate whether melting in two dimensions occurs through a qualitatively different process than it does it three dimensions. In three dimensions, the melting transition is a first-order transition, which displays a characteristic latent heat. Halperin and Nelson [Halperin:78a], [Nelson:79a] and Young [Young:79a] have raised the possibility that melting in two dimensions could occur through a qualitatively different process. They have suggested that melting could consist of a pair of higher-order phase transitions, which lack a latent heat, that are driven by topological defects in the two-dimensional crystal lattice.

We studied a two-dimensional system of particles interacting through a

truncated Lennard-Jones potential. The Lennard-Jones potential is

$$\phi = 4\epsilon\left(\frac{\sigma^{12}}{r} - \frac{\sigma^{6}}{r}\right)$$

where ϵ is the energy parameter, σ is the length parameter, and r is the distance between two particles. The potential is attractive at distances larger than σ and is repulsive at smaller distances. The potential energy of the whole system is the sum of the potential energies of each pair of interacting particles. In order to ease the computational requirements of the simulation we have truncated the potential at a particle separation of 3σ.

15.2.2 Solution Method

We chose to use a Monte Carlo method to simulate the interaction of the particles. The method consists of generating a sequence of configurations in such a way that the probability of being in configuration r, denoted as P_r, is

$$P_r \propto e^{-\beta E_r}$$

where E_r is the potential energy of configuration r and $\beta = 1/kT$. A configuration refers collectively to the positions of all the particles in the simulation. The update procedure that we describe in the next section generates such a sequence of configurations by repeatedly updating the position of each of the particles in the system. Averaging the values of such quantities as potential energy and pressure over the configurations gives their expected values in such a system.

The process of moving from one configuration to another is known as a Monte Carlo update. The update procedure we used involves three steps that allow the position of one particle to change [Metropolis:53a]. The first step is to choose a new position for the particle with uniform probability in a region about its current position. Next, the update procedure calculates the difference in potential energy between the current configuration and the new one. Finally, the new position for the particle is either accepted or rejected based on the difference in potential energy and rules that generate configurations with the required probability distribution.

The two-dimensional system being studied has several characteristics that must be considered in designing an efficient algorithm for implementing the Monte Carlo simulation. One of the most important characteristics is that the interaction potential has a short range. The Lennard-Jones potential approaches zero quickly enough that the effect of distant particles can

be safely ignored. We made the short-range nature of the potential precise by truncating it at a distance of 3σ. We must use the short-range nature of the potential to organize the particle positions so that the update procedure can locate quickly the particles whose potential energy changes during an update.

Another feature of the system that complicates the simulation is that the particles are not confined to a grid that would structure the data. Such irregular data makes simultaneously updating multiple particles more difficult. One result of the irregular data is that the computational loads of the processors is unbalanced in a distributed-memory, MIMD processor. In order to minimize the effect of the load imbalance, the nodes of the concurrent processor must run asynchronously. We developed an interrupt-driven communication system [Johnson:85a] that allows the nodes to implement an asynchronous update procedure. The interrupt-driven communication system allows a node to send requests for contributions to the change in potential energy that moving its particle would cause. Nodes receiving such requests compute the contribution of their particles and send a response reporting their result.

15.2.3 Concurrent Update Procedure

Performing Monte Carlo updates in parallel requires careful attention to ensuring that simultaneous updates do not interfere with each other. Because the basic equations governing the Monte Carlo method remain unchanged, performing the updates in parallel requires that a consistent sequential ordering of the updates exist. No particular ordering is required; only the existence of such an ordering is critical. Particles that are farther apart than the range of the interaction potential cannot influence each other, so any arbitrary ordering of their updates is always consistent. However, if some of the particles being updated together are within the range of the potential, they cannot be updated as if they were independent because the result of one update affects the others. Fortunately, the symmetry of the potential guarantees that all of the affected particles are aware that their updates are interdependent.

Each node involved in the conflicting updates must act to resolve the situation by making one of only two possible decisions. For each request for contributions to the difference in potential energy of an update, a node can either send a response immediately or delay the response until its own update finishes. If the node sends the response immediately, it must use the

old position of the particle that it is updating. If the node instead delays the response while waiting for its own update to finish, it will use the new position of the particle when its update finishes. If all of the nodes involved in the conflicting updates make consistent decisions, a sequential ordering of the updates will exist, ensuring the correctness of the Monte Carlo procedure. However, if two nodes both decide to send responses to each other based on the current positions of the particles they are updating, no such ordering will exist. If two nodes both decide to delay sending responses to each other, neither will be able to complete their update, causing the simulation to deadlock.

Several features of the concurrent update procedure make resolving such interdependent updates difficult. Each node must make its decision regarding the resolution of the conflicting updates in isolation from the other nodes because all of the nodes are running asynchronously to minimize load imbalance. However, the nodes cannot run completely asynchronously because assigning a consistent sequential ordering to the updates requires that the update procedure impose a synchronizing condition on the updates. Still, the condition should be as weak as possible so that the decrease in processing efficiency is minimized.

One solution to the problem of correctly ordering interdependent updates requires that a clock exist in each of the nodes. The update procedure records the time at which it begins updating a particle and includes that time with each of its requests for contributions to the difference in potential energy. When a node determines that its update conflicts with that of another node, it uses the times of the conflicting updates to resolve the dependence. The node sends a response immediately if the request involves an update that precedes its own. The node delays sending a response if its own update precedes the one that generated the request. Should the times be exactly equal, the unique number associated with each node provides a means of consistently ordering the updates. When each of the processors involved in the conflicting updates use such a method to resolve the situation, a consistent sequential ordering must result. Using the time of each of the conflicting updates to determine their ordering allows the earliest updates to finish first, which achieves good load balance in the concurrent algorithm.

Although delaying a response to a conflicting update is a synchronizing condition, it is sufficiently weak that it does not seriously degrade the performance of the concurrent algorithm. A node can respond to other nodes' requests while waiting for responses to requests that it generated. The node that delays sending a response can perform most of the computation to

generate the response while it is waiting for responses to its own requests because the position of only one particle is in question. In fact, the current implementation simply generates the two possible responses so that it can send the correct response immediately after its own update completes.

An interesting feature of the concurrent update algorithm is that it produces results that are inherently irreproducible. If two simulations start with exactly the same initial data, including random number seeds, the simulations will differ eventually. The source of the irreproducible behavior is that all components of the concurrent processor are not driven by the same clock. For instance, the communication channels that connect the nodes contain an asynchronous loop that allows the arrival times of messages to differ by arbitrarily small amounts. Such differences can affect the order in which requests are received, which in turn determines the order in which a node generates responses. Once such differences change the outcome of a single update, the two simulations begin to evolve independently. Both simulations continue to generate configurations with the correct probability distribution, so the statistical properties of the simulations do not change. However, the irreproducible behavior of the concurrent update algorithm can make debugging somewhat more difficult.

15.2.4 Performance Analysis

Because a complete performance analysis of the Monte Carlo simulation is rather lengthy, we provide only a summary of the analysis here. Calculating the efficiency of the concurrent update algorithm is relatively simple because it requires only measurements of the time an update takes on one node and its time on multiple nodes. A more difficult parameter to calculate is the load balance of the update procedure. In order to calculate the load balance, we measured the time required to send each type of message that the update uses. The total communication overhead is the sum of the overheads for each type of message, which is the product of the time to send that type of message and the number of such messages. We calculated the number of each type of message by assuming a uniform distribution of particles. Because the update algorithm contains no significant serial components, we attributed to load imbalance the parallel overhead remaining after accounting for the communication overhead. The load balance is a factor that can range from $1/N$, where N is the number of nodes, to one, which occurs when the loads are balanced perfectly. We give the update time in seconds, the efficiency, and the load balance for several simulations on the 64-node Caltech hypercube

Particles	Update Time	Efficiency	Load Balance
1024	1.33	0.477	0.713
4096	4.00	0.634	0.808
16384	13.52	0.750	0.858
65536	48.04	0.844	0.907

Table 15.1: Simulations on the 64-node Caltech Hypercube

in Table 15.1 ([Johnson:86a] p. 73).

15.2.5 Credits and References

Mark A. Johnson wrote the Monte Carlo simulation of melting in two dimensions for his Ph.D. research at Caltech.

15.3 Computer Chess

As this book shows, distributed-memory, multiple-instruction stream (MIMD) computers are successful in performing a large class of scientific computations. The problems tend to have regular, homogeneous data sets and the algorithms are usually "crystalline" in nature. The second generation of algorithms now being explored tend towards an amorphous structure and asynchronous execution. In short, it is less obvious how well hypercubes or transputer arrays will do.

As an attempt to explore a part of this interesting region in algorithm space, we have implemented chess on an NCUBE hypercube. Besides being a fascinating field of study in its own right, computer chess is an interesting challenge for parallel computers because:

- It is not clear how much parallelism is actually available—the important method of alpha-beta pruning conflicts with parallelism.
- Some aspects of the algorithm require a globally shared data set.
- The parallel algorithm has dynamic load imbalance of an extreme nature.

One might also ask the question, "Why study computer chess at all?" We think the answer lies in the unusual position of computer chess within the artificial intelligence world. Like most AI problems, chess requires a program which will display seemingly intelligent behavior in a limited, artificial world. Unlike most of AI, the programmers do not get to make up the rules of this world. In addition, there is a very rigorous procedure to test the intelligence of a chess program—playing games against humans. Computer chess is one area where the usually disparate worlds of AI and high performance computing meet.

Before going on, let us state that our approach to parallelism (and hence speed) in computer chess is not the only one. Belle, Cray Blitz, Hitech, and Deep Thought have shown in spectacular fashion that fine-grained parallelism (pipelining, specialized hardware) leads to impressive speeds (see [Frey:83a], [Marsland:87a], [Ebeling:85a], [Welsh:85a]). Our coarse-grained approach to parallelism should be viewed as a complementary, not a conflicting, method. Clearly the two can be combined.

15.3.1 Sequential Computer Chess

In this section we will describe some basic aspects of what constitutes a good chess program on a sequential computer. Having done this, we will be able to intelligently discuss the parallel algorithm.

At present, all competitive chess programs work by searching a tree of possible moves and countermoves. A program starts with the current board position and generates all legal moves, all legal responses to these moves, and so on until a fixed depth is reached. At each leaf node, an evaluation function is applied which assigns a numerical score to that board position. These scores are then "backed up" by a process called minimaxing, which is simply the assumption that each side will choose the line of play most favorable to it at all times. If positive scores favor white, then white picks the move of maximum score and black picks the move of minimum score. These concepts are illustrated in Figure 15.1.

The evaluation function employed is a combination of simple material balance and several terms which represent positional factors. The positional terms are small in magnitude but are important since material balance rarely changes in tournament chess games.

The problem with this brute-force approach is that the size of the tree explodes exponentially. The "branching factor" or number of legal moves in a typical position is about 35. In order to play master-level chess a search

Figure 15.1: Game Playing by Tree Searching. The top half of the figure illustrates the general idea: Develop a full-width tree to some depth, then score the leaves with the evaluation function, f. The second half shows minimaxing—the reasonable supposition that white (black) chooses lines of play which maximizes (minimizes) the score.

of depth eight appears necessary, which would involve a tree of 35^8 or about 2×10^{12} leaf nodes.

Fortunately, there is a better way. Alpha-beta pruning is a technique which always gives the same answer as brute-force searching without looking at so many nodes of the tree. Intuitively, alpha-beta pruning works by ignoring subtrees which it knows cannot be reached by best play (on the part of both sides). This reduces the effective branching factor from 35 to about six, which makes strong play possible.

The idea of alpha-beta pruning is illustrated in Figure 15.2. Assume that all child nodes are searched in the order of left to right in the figure. On the left side of the tree (the first subtree searched), we have minimaxed and found a score of $+4$ at depth one. Now start to analyze the next subtree. The children report back scores of $+5$, -1,.... The pruning happens after the score of -1 is returned: since we are taking the minimum of the scores $+5$, -1, ... we immediately have a bound on the scores of this subtree—we know the score will be no larger than -1. Since we are taking the maximum at the next level up (the root of the tree) and we already have a line of play better than -1 (namely, the $+4$ subtree), we need not explore this second subtree any further. Pruning occurs, as denoted by the dashed branch of the second subtree. The process continues through the rest of the subtrees.

The amount of work saved in this small tree was insignificant but alpha-

Figure 15.2: Alpha-Beta Pruning for the Same Tree as Figure 15.1. The tree is generated in left-to-right order. As soon as the score -1 is computed, we immediately have a bound on the level above (≤ -1) which is below the score of the $+4$ subtree. A cutoff occurs, in that no more descents of the ≤ -1 node need to be searched.

beta becomes very important for large trees. From the nature of the pruning method, one sees that the tree is not evolved evenly downward. Instead, the algorithm pursues one branch all the way to the bottom, gets a "score to beat" (the alpha-beta bounds), and then sweeps across the tree sideways. How well the pruning works depends crucially on move ordering. If the best line of play is searched first, then all other branches will prune rapidly.

Actually, what we have discussed so far is not full alpha-beta pruning, but merely "pruning without deep cutoffs." Full alpha-beta pruning shows up only in trees of depth four or greater. A thorough discussion of alpha-beta with some interesting historical comments can be found in Knuth and Moore, [Knuth:75a].

The Evaluation Function

The evaluation function of our program is similar in style to that of the Cray Blitz program [Welsh:85a]. The most important term is material, which is a simple count of the number of pieces on each side, modified by a factor which encourages the side ahead in material to trade pieces but not pawns. The material evaluator also recognizes known draws such as king and two knights vs. king.

There are several types of positional terms, including pawn structure,

king safety, center control, king attack, and specialized bonuses for things like putting rooks on the seventh rank.

The pawn structure evaluator knows about doubled, isolated, backward, and passed pawns. It also has some notion of pawn chains and phalanxes. Pawn structure computation is very expensive, so a hashtable is used to store the scores of recently evaluated pawn structures. Since pawn structure changes slowly, this hashtable almost always saves us the work of pawn structure evaluation.

King safety is evaluated by considering the positions of all pawns on the file the king is occupying and both neighboring files. A penalty is assessed if any of the king's covering pawns are missing or if there are holes (squares which can never be attacked by a friendly pawn) in front of the king. Additional penalties are imposed if the opponent has open or half-open files near the king. The whole king safety score is multiplied by the amount of material on the board, so the program will want to trade pieces when its king is exposed, and avoid trades when the opponent's king is exposed. As in pawn structure, king safety uses a hashtable to avoid recomputing the same information.

The center control term rewards the program for posting its pieces safely in the center of the board. This term is crude since it does not consider pieces attacking the center from a distance, but it can be computed very quickly and it encourages the kind of straightforward play we want.

King attack gives a bonus for placing pieces near the opposing king. Like center control, this term is crude but tends to lead to positions in which attacking opportunities exist.

The evaluation function is rounded out by special bonuses to encourage particular types of moves. These include a bonus for castling, a penalty for giving up castling rights, rewards for placing rooks on open and half-open files or on the seventh rank, and a penalty for a king on the back rank with no air.

Quiescence Searching

Of course it only makes sense to apply a static evaluation function to a position which is quiescent, or tactically quiet. As a result, the tree is extended beyond leaf nodes until a quiescent position is reached, where the static evaluator is actually applied.

We can think of the quiescence search as a dynamic evaluation function, which takes into account tactical possibilities. At each leaf node, the side to

move has the choice of accepting the current static evaluation or of trying to improve its position by tactics. Tactical moves which can be tried include pawn promotions, most capture moves, some checks, and some pawn promotion threats. At each newly generated position the dynamic evaluator is applied again. At the nominal leaf nodes, therefore, a narrow (small branching factor), tactical search is done, with the static evaluator applied at all terminal points of this search (which end up being the true leaves).

Iterative Deepening

Tournament chess is played under a strict time control, and a program must make decisions about how much time to use for each move. Most chess programs do not set out to search to a fixed depth, but use a technique called iterative deepening. This means a program does a depth two search, then a depth three search, then a depth four search, and so on until the allotted time has run out. When the time is up, the program returns its current best guess at the move to make.

Iterative deepening has the additional advantage that it facilitates move ordering. The program knows which move was best at the previous level of iterative deepening, and it searches this principal variation first at each new level. The extra time spent searching early levels is more than repaid by the gain due to accurate move ordering.

The Hashtable

During the tree search, the same board position may occur several times. There are two reasons for this. The first is transposition, or the fact that the same board position can be reached by different sequences of moves. The second reason is iterative deepening—the same position will be reached in the depth two search, the depth three search, etc. The hashtable is a way of storing information about positions which have already been searched; if the same position is reached again the search can be sped up or eliminated entirely by using this information.

The hashtable plays a central role in a good chess program and so we will describe it in some detail. First of all, the hashtable is a form of content addressable memory—with each chess board (a node in the chess tree) we wish to associate some slot in the table. Therefore, a hashing function, h, is required, which maps chess boards to slots in the table. h is designed so as to scatter similar boards across the table. This is done because in any single

search the boards appearing in the tree differ by just a few moves and we wish to avoid collisions (different boards mapping to the same slot) as much as possible. Our hash function is taken from [Zobrist:70a]. Each slot in the table contains:

- the known bounds on the score of this position
- the depth to which these bounds are valid
- a suggested move to try
- a staleness flag
- 64 bit collision check

Instead of just blindly generating all legal moves at a position and then going down these lines of play, the hashtable is first queried about the position. Occasionally, the hashtable bounds are so well-determined as to cause an immediate alpha-beta cutoff. More often, the hashtable has a suggested move to try and this is searched first. The 64-bit collision check is employed to ensure that the slot has information about the same position that the program is currently considering (remember, more than one chess board can map to the same slot in the table).

Whenever the program completes the search of a subtree of substantial size (i.e., one of depth greater than some minimum), the knowledge gained is written into the hash table. The writing is not completely näive, however. The table contains only a finite number of slots so collisions occur; writeback acts so as to keep the most valuable information. The depth field of the slot helps in making the decision as to what is most valuable. The information coming from the subtree of greater depth (and hence, greater value) is kept.

The staleness flag allows us to keep information from one search to the next. When time runs out and a search is considered finished, the hash table is not simply cleared. Instead, the staleness flag is set in all slots. If, during the next search, a read is done on a stale slot the staleness flag is cleared, the idea being that this position again seems to be useful. On writeback, if the staleness flag is set the slot is simply overwritten, without checking the depths. This prevents the hashtable from becoming clogged with old information.

Proper use of an intelligent hashtable such as described in the above gives one, in effect, a "principal variation" throughout the chess tree. As discussed in [Ebeling:85a], a hashtable can effectively give near-perfect move ordering and hence, very efficient pruning.

The Opening

The opening is played by making use of an "opening book" of known positions. Our program knows the theoretically "best" move in about 18,000 common opening positions. This information is stored as a hashtable on disk and can be looked up quickly. This hashtable resolves collisions through the method of chaining [Knuth:73a].

The Endgame

Endgames are handled by using special evaluation functions which contain knowledge about endgame principles. For instance, an evaluator for king and pawn endgames may be able to directly recognize a passed pawn which can race to the last rank without being caught by the opposing king. Except for the change in evaluation functions, the endgame is played in the same fashion as the middlegame.

15.3.2 Parallel Computer Chess

The Hardware

Our program is implemented on an NCUBE/10 system. This is an MIMD (multiple instruction stream, multiple data stream) multicomputer, with each node consisting of a custom VLSI processor running at 7 MHz, 512 kbytes of memory, and on-chip communication channels. There is no shared memory—processors communicate by message-passing. The nodes are connected as a hypercube but the VERTEX message-passing software [NCUBE:87a] gives the illusion of full connectivity. The NCUBE system at Caltech has 512 processors, but systems exist with as many as 1024 processors. The program is written in C, with a small amount of assembly code.

15.3.3 Parallel Alpha-Beta Pruning

Some good chess programs do run in parallel (see [Finkel:82a], [Marsland:84a], [Newborn:85a], [Schaeffer:84a], [Schaeffer:86a]), but before our work nobody had tried more than about 15 processors. We were interested in using hundreds or thousands of processors. This forced us to squarely face all the issues of parallel chess—algorithms which work for a few processors do not necessarily scale up to hundreds of processors. An example of this is the occurrence of sequential bottlenecks in the control

Figure 15.3: Slaves Searching Sub-trees in a Self-scheduled Manner. Suppose one of the searches, in this case, search two, takes a long time. The advantage of the self-scheduling is that, while this search is proceeding in slave two, the other slaves will have done all the remaining work. This very general technique works as long as the dynamic range of the computation times is not too large.

structure of the program. We have been very careful to keep the control of the program de-centralized so as to avoid these bottlenecks.

The parallelism comes from searching different parts of the chess tree at the same time. Processors are organized in a hierarchy with one master processor controlling several teams, each submaster controlling several subteams, etc. The basic parallel operation consists of one master coming to a node in the chess tree, and assigning subtrees to his slaves in a self-scheduled way. Figure 15.3 shows a timeline of how this might happen with three subteams. Self-scheduling by the slaves helps to load-balance the computation, as can be seen in the figure.

So far, we have defined what happens when a master processor reaches a node of the chess tree. Clearly, this process can be repeated recursively. That is, each subteam can split into sub-subteams at some lower level in the tree. This recursive splitting process, illustrated in Figure 15.4, allows large numbers of processors to come into play.

In conflict with this is the inherent sequential model of the standard alpha-beta algorithm. Pruning depends on fully searching one subtree in order to establish bounds (on the score) for the search of the next subtree. If one adheres to the standard algorithm in an overly strict manner, there may be little opportunity for parallelism. On the other hand, if one is too näive

Figure 15.4: The splitting process of Figure 15.3 is now repeated, in a recursive fashion, down the chess tree to allow large numbers of processors to come into play. The top-most master has four slaves, which are each in turn an entire team of processors, and so on. This figure is only approximate, however. As explained in the text, the splitting into parallel threads of computation is not done at every opportunity but is tightly controlled by the global hashtable.

in the design of a parallel algorithm the situation is easily reached where the parallel program searches an impressive number of board positions per second, but still does not search much more deeply than a single processor running the alpha-beta algorithm. The point is that one should not simply split or "go parallel" at every opportunity—as we will see below it is sometimes better to leave processors idle for short periods of time and then do work at more effective points in the chess tree.

Analysis of Alpha-Beta Pruning

The standard source on mathematical analysis of the alpha-beta algorithm is the paper by Knuth and Moore [Knuth:75a]. This paper gives a complete analysis for perfectly ordered trees, and derives some results about randomly ordered trees. We will concern ourselves here with perfectly ordered trees, since real chess programs achieve almost-perfect ordering.

In this context, perfect move-ordering means that in any position, we always consider the best move first. Ordering of the rest of the moves does not matter. Knuth and Moore show that in a perfectly ordered tree, the nodes can be divided into three types, as illustrated by Figure 15.5. As in

Figure 15.5: Pruning of a Perfectly Ordered Tree. The tree of Figure 15.1 and Figure 15.2 has been extended another ply, and also the move ordering has been re-arranged so that the best move is always searched first. By classifying the nodes into types as described in the text, the following pattern emerges: all children of type one and three nodes are searched, while only the first child of a type two node is searched.

previous figures, nodes are assumed to be generated and searched in left to right order. The typing of the nodes is as follows. Type one nodes are on the "principal variation." The first child of a type one node is type one and the rest of the children are type two. Children of type two nodes are type three, and children of type three nodes are type two.

How much parallelism is available at each node? The pruning of the perfectly ordered tree of Figure 15.5 offers a clue. By thinking through the alpha-beta procedure one notices the following pattern:

- all children of type one nodes are searched,
- only the first child of a type two node is searched—the rest are pruned, and,
- all children of type three nodes must be searched.

This oscillating pattern between the node types is, of course, the reason for distinguishing them as different types in the first place.

The implications of this for a parallel search are important. To efficiently search a perfectly ordered tree in parallel, one should perform the following algorithm.

- At type one nodes, the first child must be searched sequentially (in order to initialize the alpha-beta bounds), then the rest can be searched in parallel.
- At type two nodes there is no parallelism since only one child will be searched (time spent searching other children will be wasted).
- Type three nodes, on the other hand, are fully parallel and all the children can be searched independently and simultaneously.

The key for parallel search of perfectly ordered chess trees, then, is to stay sequential at type two nodes, and go parallel at type three nodes. In the non-perfectly ordered case, the clean distinction between node types breaks down, but is still approximately correct. In our program, the hashtable plays a role in deciding upon the node type. The following strategy is used by a master processor when reaching a node of the chess tree:

> Make an inquiry to the hashtable regarding this position. If the hashtable suggests a move, search it first, sequentially. In this context, "sequentially" means that the master takes her slaves with her down this line of play. This is to allow possible parallelism lower down in the tree. If no move is suggested or the suggested move fails to cause an alpha-beta cutoff, search the remaining moves in parallel. That is, farm the work out to the slaves in a self-scheduled manner.

This parallel algorithm is intuitively reasonable and also reduces to the correct strategy in the perfectly ordered case. In actual searches, we have explicitly (on the NCUBE graphics monitor) observed the sharp classification of nodes into type two and type three at alternate levels of the chess tree.

Global Hashtable

The central role of the hashtable in providing refutations and telling the program when to go parallel makes it clear that the hashtable must be shared between all processors. Local hashtables would not work since the complex, dynamically-changing organization of processors makes it very unlikely that a processor will search the same region of the tree in two successive levels of iterative deepening. A shared table is expensive on a distributed-memory machine, but in this case it is worth it.

Each processor contributes an equal amount of memory to the shared hashtable. The global hashfunction maps each chess position to a global slot number consisting of a processor ID and a local slot number. Remote memory is accessed by sending a message to the processor in which the desired memory resides. To insure prompt service to remote memory requests, these messages must cause an interrupt on arrival. The VERTEX system does not support this feature, so we implemented a system called generalized signals [Felten:88b] which allows interrupt-time servicing of some messages without disturbing the running program.

When a processor wants to read a remote slot in the hashtable, it sends a message containing the local slot number and the 64-bit collision check to the appropriate processor. When this message arrives the receiving processor is interrupted; it updates the staleness flag and sends the contents of the desired slot back to the requesting processor. The processor which made the request waits until the answer comes back before proceeding.

Remote writing is a bit more complicated due to the possibility of collisions. As explained previously, collisions are resolved by a priority scheme; the decision of whether to overwrite the previous entry must be made by the processor which actually owns the relevant memory. Remote writing is accomplished by sending a message containing the new hashtable entry to the appropriate processor. This message causes an interrupt on arrival and the receiver examines the new data and the old contents of that hashtable slot and decides which one to keep.

Since hashtable data is shared between many processors, any access to the hashtable must be an atomic operation. This means we must guarantee that two accesses to the same slot cannot happen at the same time. The generalized signals system provides a critical-section protection feature which can be used to queue remote read and write requests while an access is in progress.

Experiments show that the overhead associated with the global hashtable is only a few percent, which is a small price to pay for accurate move ordering.

15.3.4 Load Balancing

As we explained in an earlier section, slaves get work from their masters in a self-scheduled way in order to achieve a simple type of load balancing. This turns out not to be enough, however. By the nature of alpha-beta, the time to search two different sub-trees of the same depth can vary quite dramatically. A factor of 100 variation in search times is not unreasonable.

Self-scheduling is somewhat helpless in such a situation. In these cases a single slave would have to grind out the long search, while the other slaves (and conceivably, the entire rest of the machine) would merely sit idle. Another problem, near the bottom of the chess tree, is the extremely rapid time scales involved. Not only do the search times vary by a large factor, but this all happens at millisecond time scales. Any load balancing procedure will therefore need to be quite fast and simple.

These "chess hot spots" must be explicitly taken care of. The master and submaster processors, besides just waiting for search answers, updating alpha-beta bounds, and so forth, also monitor what is going on with the slaves in terms of load balance. In particular, if some minimum number of slaves are idle and if there has been a search proceeding for some minimum amount of time, the master halts the search in the slave containing the hot spot, re-organizes all his idle slaves into a large team, and re-starts the search in this new team. This process is entirely local to this master and his slaves and happens recursively, at all levels of the processor tree.

This "shoot-down" procedure is governed by the two parameters: the minimum number of idle slaves, and the minimum time before calling a search a potential hot spot. These parameters are introduced to prevent the halting, processor re-arrangement and its associated overhead in cases which are not necessarily a hot spot. The parameters are tuned for maximum performance.

The payoff of dynamic load balancing has been quite large. Once the load balancing code was written, debugged, and tuned, the program was approximately three times faster than before load balancing. Through observations of the speed-up (to be discussed below) and also by looking directly at the execution of the program across the NCUBE (using the parallel graphics monitor, also to be discussed below) we have become convinced that the program is well load balanced and we are optimistic about the prospects for scaling to larger speed-ups on larger machines.

An interesting point regarding asynchronous parallel programming was brought forth by the dynamic load balancing procedure. It is concerned with the question, "Once we've re-arranged the teams and completed the search, how do we return to the original hierarchy so as to have a reasonable starting point for the next search?" Our first attempts at resetting the processor hierarchy met with disaster. It turned out that processors would, occasionally, not make it back into the hierarchy (that is, be the slave of someone) in time for another search to begin. This happened because of the asynchronous nature of the program and the variable amount of time mes-

sages take to travel through the machine. Once this happened, the processor would end up in the wrong place in the chess tree and the program would soon crash. A natural thing to try in this case is to demand that all processors be re-connected before beginning a new search but we rejected this as being tantamount to a global re-synchronization and hence very costly. We therefore took an alternate route whereby the code was written in a careful manner so that processors could actually stay disconnected from the processor tree, and the whole system could still function correctly. The disconnected processor would re-connect eventually—it would just miss one search. This solution seems to work quite well both in terms of conceptual simplicity and speed.

15.3.5 Speed-up Measurements

Speed-up is defined as the ratio of sequential running time to parallel running time. We measure the speed-up of our program by timing it directly with different numbers of processors on a standard suite of test searches. These searches are done from the even-numbered Bratko-Kopec positions citeBratko:82a, a well-known set of positions for testing chess programs. Our benchmark consists of doing two successive searches from each position and adding up the total search time for all twenty-four searches. By varying the depth of search we can control the average search time of each benchmark.

The speed-ups we measured are shown in Figure 15.6. Each curve corresponds to a different average search time. We find that speed-up is a strong function of the time of the search (or equivalently, its depth). This result is a reflection of the fact that deeper search trees have more potential parallelism and hence, more speed-up. Our main result is that at tournament speed (the uppermost curve of the figure), our program achieves a speed-up of 101 out of a possible 256. Not shown in this figure is our later result: a speed-up estimated to be 170 on a 512 node machine.

The "double hump" shape of the curves is also understood: the location of the first dip, at 16 processors, is the location at which the chess tree would like the processor hierarchy to be a one-level hierarchy sometimes, a two-level hierarchy at other times. We always use a one-level hierarchy for 16 processors so we are sub-optimal here. Perhaps this is an indication that a more flexible processor allocation scheme could do somewhat better.

Figure 15.6: The Speed-up of the Parallel Chess Program as a Function of Machine Size and Search Depth. The results are averaged over a representative test set of 24 chess positions. The speed-up increases dramatically with search depth, corresponding to the fact that there is more parallelism available in larger searches. The uppoermost curve corresponds to tournament play—the program runs more than 100 times faster on 256 nodes as on a single NCUBE node when playing at tournament speed.

15.3.6 Real-time Graphical Performance Monitoring

One tool we have found extremely valuable in program development and tuning is a real-time performance monitor with color-graphics display. Our NCUBE hardware has a high-resolution color graphics monitor driven by many parallel connections into the hypercube. This gives sufficient bandwidth to support a status display from the hypercube processors in real time. Our performance monitoring software was written by Rod Morison and is described in [Morison:88a].

The display shows us where in the chess tree each processor is, and it draws the processor hierarchy as it changes. By watching the graphics screen we can see load imbalance develop and observe dynamic load balancing as it tries to cope with the imbalance. The performance monitor gave us the first evidence that dynamic load balancing was necessary, and it was invaluable in debugging and tuning the load balancing code.

15.3.7 Speculation

The best computer chessplayer of today (Deep Thought) has reached grandmaster strength. How strong a player can be built within five years, using

today's techniques?

Deep Thought is a chess engine implemented in VLSI, and searches roughly 500,000 positions per second. The speed of Chiptest-type engines can probably be increased by about a factor of 30 through design refinements and improvements in fabrication technology. This factor comes from assuming a speed doubling every year, for five years. Our own results imply that an additional factor of 250 speed-up due to coarse-grain parallelism is plausible. This is assuming something like a 1000-processor machine with each processor being an updated version of Deep Thought. This means that a machine capable of searching 3.75 billion ($30 \times 250 \times 500,000$) positions per second is not out of the question within five years.

Communication times will also need to be improved dramatically over the current NCUBE. This will entail hardware specialization to the requirements of chess. How far communication speeds can be scaled and how well the algorithm can cope with proportionally slower communications are poorly understood issues.

The relationship between speed and playing strength is well-understood for ratings below 2500. A näive extrapolation of Thompson's results [Thompson:82a] indicates that a doubling in speed is worth about 40 rating points in the regime above 2500. Thus this machine would have a rating somewhere near 3000, which certainly indicates world-class playing strength.

Of course nobody really knows how such a powerful computer would do against the best grandmasters. The program would have an extremely unbalanced style and might well be stymied by the very deep positional play of the world's best humans. We must not fall prey to the overconfidence which led top computer scientists to lose consecutive bets to David Levy!

15.3.8 Credits

Steve Otto and Ed Felten were the leaders of the chess project and did the majority of the work. Eric Umland began the project and would have been a major contributor but for his untimely death. Rod Morison wrote the opening book code and also developed the parallel graphics software. Summer students Ken Barish and Rob Fätland contributed chess expertise and various peripheral programs.

15.4 Ray Tracing

*** Contribution needed from J. Salmon

15.5 Database

*** Contribution needed from G. C. Fox

Chapter 16

Asynchronous Message Passing

16.1 General Structure of Asynchronous Software

*** Contribution needed from G. C. Fox

16.2 Zipcode

*** Contribution needed from A. Skjellum

Chapter 17

High Level Synchronous Software Systems

17.1 MOOS II: An Operating System for Dynamic Load Balancing on the iPSC/1

Applications involving irregular time behavior or dynamically varying data structures are difficult to program using the crystalline model or its variants. Examples are dynamically adaptive grids for studying shock waves in fluid dynamics, N-body simulations of gravitating systems, and artificial intelligence applications, such as chess. The few applications in this class that have been written typically use custom designed operating systems and special techniques.

To support applications in this class, we developed a new, general purpose operating system called MOOS for the Mark II hypercube [Salmon:88a], and later wrote an extended version called MOOS II for the Intel iPSC/1 [Koller:88b]. While the MOOS system was fairly convenient, it became available at a time when the Mark II and iPSC/1 were falling into disuse because of uncompetitive performance. Only one real application was ever written (Ray tracing, Section 15.4 [Salmon:88c]), and only toy applications were written using MOOS II. Its main value was, therefore, as an experiment in operating system design and some of its features are now incorporated in *Express* (Section 5.2).

Figure 17.1: An executing MOOS program is a dynamic network (left) of tasks communicating through fifo buffers called pipes (right).

17.1.1 Design of MOOS

The user writes a MOOS program as a large collection of small tasks, that communicate with each other by sending messages through pipes, as shown in Figure 17.1. Each task controls a piece of data, so it can be viewed as an object in the object-oriented sense (hence the name Multitasking Object-oriented OS). The tasks and pipes can be created at any time by any task on any node, so the whole system is completely dynamic.

The MOOS II extensions allow one to form groups of tasks called teams that share access to a piece of data. Also, a novel feature of MOOS II is that teams are relocatable, *i.e.*, they can be moved from one node to another while they are running. This allows one to perform dynamic load balancing if necessary.

The various subsystems of MOOS II, which together form a complete operating system and programming environment, are shown in Figure 17.2. For convenience, we attempted to preserve a UNIX flavor in the design, and were also able to provide support for debugging and performance evaluation because the iPSC/1 hardware has built-in memory protection. Easy interaction with the host is achieved using ICubix, an asynchronous version of Cubix (Section 5.2) that gives each task access to the Unix system calls on the host. The normal C-compilers can be used for programming, and the only extra utility program required is a binder to link the user program to the operating system.

Despite the increased functionality, the performance of MOOS II on the

Figure 17.2: Subsystems of the MOOS II Operating System

iPSC/1 turned out to be slightly better than that of Intel's proprietary NX system.

17.1.2 Dynamic Load Balancing Support

Our plan was to use MOOS II to study dynamic load balancing, and eventually incorporate a dynamic load balancer in the MOOS system. However, our first implementation of a dynamic load balancer, along the lines of [Fox:86h], convinced us that dynamic load balancing is a difficult and many-faceted issue, so the net result was a better understanding of the complexities rather than a general-purpose balancer.

The prototype dynamic load balancer worked as shown in Figure 17.3, and is appropriate for applications where the number of MOOS teams in an application is constant, but the amount of work performed by individual teams changes slowly with time. A centralized load manager on node 0 keeps statistics on all the teams in the machine. At regular intervals, the teams report the amount of computation and communication they have done since the last report, and the central manager computes the new load balance. If the balance can be improved significantly, some teams are relocated to new nodes, and the cycle continues.

This centralized approach is easy, and good in that it relocates as few teams as possible to maintain the balance, but it is bad because computing which teams to move becomes a sequential bottleneck. For instance, for 256 teams on 16 processors, a simulated annealing optimization takes about 1.2 seconds on the iPSC/1, while the actual relocation process only takes about

Figure 17.3: One simple load balancing scheme implemented in MOOS II

0.3 seconds, so the method is limited to applications where load redistribution only needs to be done every 10 seconds or so. The lesson here is that to be viable, the load optimization step itself must be parallelized. The same conclusion will also hold for any other distributed memory machine, since the ratio of computation time to optimization time is fairly machine-independent.

17.1.3 What We Learned

Aside from finding some new reasons not to use old hardware, we were able to pinpoint issues worthy of further study, concerning parallel programming in general and load balancing in particular.

1. In the MOOS programming style, it is messy and expensive for user tasks to find out when other groups of tasks have terminated. In our defense, the same unsolved problem exists in ADA.

2. There are serious ambiguities in the meaning of parallel file IO for asynchronous systems like MOOS, which are exacerbated when tasks can move from node to node. When writing, does each task have a block in the file, and if so, how do we correlate tasks with blocks? There are hints that a hypertext-like system is more suitable than a linear file for parallel IO, but we were unable to obtain completely satisfactory semantics.

3. It is not clear that there is a general solution to the dynamic load balancing problem. The issue may be as resistant to classification as,

say, the general nonlinear PDE problem. A better solution may be to provide language support so that the programmer can control the load distribution as part of the program. Existing applications that load balance successfully (e.g. Felten *et al's* chess [Felten:86a] or Salmon *et al's* N-body solver [Salmon:89a]) compute and use load information on the fly in ways that a general-purpose method could not.

4. As the number of tasks grows, the amount of load information grows enormously, and we have to be selective about what we record. The best choice seems to be application- and machine-dependent.

5. In irregular problems, having a large number of tasks does not automatically solve the load balance problem. Unforeseen correlations between tasks tend to appear that confound naive balancing schemes. The load balancing problem must be tackled on many scales simultaneously.

Future work will, therefore, have to focus less on the mechanism of moving tasks around, and more on how to communicate load information between user and system.

17.1.4 Credits and C^{3}P References

MOOS was written by John Salmon, Sean Callahan, John Flower and Adam Kolawa. MOOS II was written by Jeff Koller.

C3P References: [Salmon:88a] [Koller:88a], [Koller:88b], [Koller:88d], [Koller:89a], [Fox:86h].

17.2 Time Warp

Discrete event simulations are among the most expensive of all computational tasks. With current technology, one sequential execution of a large simulation can take hours or days of sequential processor time. For example, many large military simulations take days to complete on standard single processors. If the model is probabilistic, many executions will be necessary to determine the output distributions. Nevertheless, many scientific, engineering, and military projects depend heavily on simulation because experiments on real systems are too expensive or too unsafe. Therefore, any technique that speeds up simulations is of great importance.

We designed the Time Warp Operating System (TWOS) to address this problem. TWOS is a multiprocessor operating system that runs parallel discrete event simulations. We developed TWOS on the Caltech/JPL Mark III Hypercube. We have since ported it to various other parallel architectures, including the Transputer and a BBN Butterfly Plus. TWOS is not intended as a general-purpose multiuser operating system, but rather as an environment for a single concurrent application (especially a simulation) in which synchronization is specified using virtual time [Jefferson:85c].

The innovation that distinguishes TWOS from other operating systems is its complete commitment to an optimistic style of execution and to process rollback for almost all synchronization. Most distributed operating systems either cannot handle process rollback at all, or implement it as a rarely used mechanism for special purposes such as exception handling, deadlock breaking, transaction abortion, or fault recovery. But the Time Warp Operating System embraces rollback as the normal mechanism for process synchronization, and uses it as often as process blocking is used in other systems. TWOS contains a simple, general distributed rollback mechanism capable of undoing or preventing any side-effect, direct or indirect, of an incorrect action. In particular, it is able to control or undo such troublesome side-effects as errors, infinite loops, I/O, creation and destruction of processes, asynchronous message communication, and termination.

TWOS uses an underlying kernel to provide basic message-passing capabilities, but it is not used for any other purpose. On the Caltech/JPL Mark III Hypercube, this role was played by *Cubix*, described in section 5.2. The other facilities of the underlying operating system are not used because rollback forces a rethinking of almost all operating system issues, including scheduling, synchronization, message queueing, flow control, memory management, error handling, I/O, and commitment. All of these are handled by TWOS. Only the extra work of implementing a correct message-passing facility prevents TWOS from being implemented on the bare hardware.

We have been developing TWOS since 1983. It is now an operational system, including many advanced features, such as dynamic creation and destruction of objects, dynamic memory management, and dynamic load management. TWOS is being used by the United States Army's Concept and Analysis Agency to develop a new generation of theater-level combat simulations. TWOS has also been used to model parallel processing hardware, computer networks, and biological systems.

Figure 17.4 shows the performance of TWOS on one simulation called STB88. This simulation models theater level combat in central Europe

[Wieland:89a]. The graph in Figure 17.4 shows how much version 2.3 of TWOS was able to speed up this simulation on varying numbers of nodes of a parallel processor. The speedup shown is relative to running a sequential simulator on a single node of the same machine used for the TWOS runs. The sequential simulator uses a splay tree for its event queue. It never performs rollback, and hence has a lower overhead than TWOS. The sequential simulator links with exactly the same application code as TWOS. It is intended to be the fastest possible general purpose discrete event simulator that can handle the same application code as TWOS.

Figure 17.4 demonstrates that TWOS can run this simulation more than 25 times faster than running it on the sequential simulator, given sufficient numbers of nodes. On other applications, even higher speedups are possible. In certain cases, TWOS has achieved within 30% of the maximum theoretical speedup, as determined by critical path analysis.

Research continues on TWOS. Currently, we are investigating dynamic load management [Reiher:90a]. Dynamic load management is important for TWOS because good speedups generally require careful mapping of a simulation's constituent objects to processor nodes. If the balance is bad, then the run is slow. But producing a good static balance takes approximately the same amount of work as running the simulation on a single node. Dynamic load management allows TWOS to achieve nearly the same speed with simple mappings as with careful mappings.

Dynamic load management is an interesting problem for TWOS because the utilizations of TWOS' nodes are almost always high. TWOS optimistically performs work whenever work is available, so nodes are rarely idle. On the other hand, much of the work done by a node may be rolled back, contributing nothing to completing the computation. Instead of balancing simple utilization, TWOS balances effective utilization, the proportion of a node's work that is not rolled back. Use of this metric has produced very promising preliminary results.

Future research directions for TWOS include database management, real-time user interaction with TWOS, and the application of virtual time synchronization to other types of parallel processing problems. [Jefferson:87a] contains a more complete description of TWOS.

17.2.1 Credits and C^{3}P References

David Jefferson, Peter Reiher, Brian Beckman, Frederick Wieland, Mike Di Loreto, Philip Hontalas, John Wedel, Leo Blume, Joseph Ruffles, Steven

Belenot, Jack Tupman, Herb Younger, Richard Fujimoto, Kathy Sturdevant, Lawrence Hawley, Abe Feinberg, Pierre LaRouche, Matt Presley, and Van Warren

Figure 17.4: Performance of TWOS on STB88

Chapter 18

Data Analysis

18.1 T10 and Other System Issues

*** Contribution needed from G. C. Fox

18.2 Pulsars

*** Contribution needed from T. Prince

18.3 Optical and Infrared Interferometry

*** Contribution needed from T. Prince

18.4 Processing of Voyager Images for Neptune

*** Contribution needed from J. Solomon

18.5 Sar

*** Contribution needed from G. C. Fox

Chapter 19

BMC^{3}IS Simulations

19.1 Introduction

*** Contribution needed from G. C. Fox

19.2 Concurrent Multi-Target Tracking

19.2.1 Introduction

Some sort of generic introductory material should go here. I have no idea yet just what it should be, or how long it should be.

19.2.2 Nature of the Problem

Sim89 is designed to process a so-called 'mass raid' scenario, in which a few hundred primary threats are launched within a one to two minute time window, together with about 40–60 secondary, anti-satellite launches. The primary targets boost through two stages of powered flight (total boost time is about 300 seconds), with each booster ultimately deploying a single Post Boost Vehicle (PBV). Over the next few hundred seconds, each PBV in turn deploys 10 Re-entry Vehicles (RV's). The Sim89 environment does not yet include the factor of 10–100 increase in object counts due to decoys, as would be expected in the 'real world'.

The data available for the tracking task consist, essentially, of line of sight measurements from various sensing platforms to individual objects in the target ensemble at (fairly) regular time intervals. At present, all sensing

Figure 19.1: I need a figure caption here

platforms are assumed to travel in circular orbits about a spherically symmetric earth (neither of these assumptions/simplifications is essential). The current program simulates two classes of sensors: GEO platforms, in geostationary, equatorial orbits, and MEO platforms in polar orbits at altitude 2000 km. The scan time for GEO sensors is typically taken to be $\Delta T = 5$ sec, with $\Delta T = 10$ sec for MEO platforms.

Figure 19.2.2 shows a small portion of the field of view of a MEO sensor at a time about half way through the RV deployment phase of a typical Sim89 scenario. The circles are the data seen by the sensor at one scan and the crosses are the data seen by the same sensor at a time $\Delta T = 10$ sec later. Given such data, the primary tasks of the tracking program are fairly simple to state:

1. Determine which data at one scan are associated with data from previous scans.

2. For associated data points over several scans, determine the trajectory of the underlying targets.

Moreover, these tasks must be done in an extremely timely fashion. For example, the ASAT's mentioned at the beginning of this subsection have burn times of order 100 seconds, and can hit a space-based asset in less that 250 seconds after launch. In order to allow self/mutual defense among the space-based assets, the tracker must provide good 3D tracks for ASAT's within about 60 seconds of launch.

In order to (in principle) process data from a wide variety of sensors, the Sim89 tracker adopts a simple unified sensing formalism. For each sensor,

the *Standard Reference Plane* is taken to be the plane passing through the center of the earth, normal to the vector from the center of the earth to the (instantaneous) satellite position. Note that these standard frames are *not* inertial. The 2D data used by the tracking algorithm are the coordinates of the intersection of the reference plane and the line of sight from the sensor to the target. The intersection coordinates are defined in terms of a standard cartesian basis in the reference frame, with one axis along the normal to the sensor's orbital plane, and the other parallel to the projection of the platform velocity onto the reference plane.

19.2.3 Tracking Techniques

The task of interpreting data such as those shown in Figure 19.2.2 is clearly rather challenging. The tracking algorithm described in the next section is based on a number of elementary building blocks, which are now briefly described.

Single Target Tracking

In order to associate observations from successive scans, a model for the expected motion of the underlying target is required. The system model used throughout the Sim89 tracker is based on a simple kinematic Kalman filter. Consider, for the moment, motion in one dimension. The model used to describe this motion is

$$\frac{d}{dt}\begin{pmatrix} x \\ v \\ a \\ j \end{pmatrix} = \begin{pmatrix} v \\ a \\ j \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ w_q \end{pmatrix} \tag{19.1}$$

where x, v, a and j are position, velocity, acceleration and jerk ($j = da/dt$) and w_q is a stochastic (noise) contribution to the jerk. The Kalman filter based on Equation 19.1 is completely straightforward, and ultimately depends on a single parameter

$$q \equiv \langle w_q^2 \rangle \approx \frac{10^{-8}\,\mathrm{km}^2}{sec^7} \tag{19.2}$$

The system model of Equation 19.1 is appropriate for describing targets travelling along trajectories with unknown but approximately smooth accelerations. The size of the noise term in Equation 19.2 determines the

magnitude of abrupt changes in the acceleration which can be accomodated by the model without loss of track. For the typical noise value quoted in Equation 19.2, scan-to-scan variations $|\Delta a| \leq 1\text{-}2\,g$ are easily accomodated.

During boost phase, the actual trajectories of the targets are, in principle, not known, and the substantial freedom for unanticipated meneuvering implicit in Equation 19.1–Equation19.2 is essential. On the other hand, the exact equation of motion for ballistic target (i.e., RV's) is completely known, so that the uncertainties in predicted positions according to the kinematic model are much larger than is necessary. Nonetheless, Equation 19.1 is maintained as the primary system model throughout *all* phases of the Sim89 tracker. This choice is based primarily on considerations of speed. Evaluations of predicted positions according to Equation 19.1 require only polynomial arithmetic and are much faster than predictions done using the exact equations of motion. Moreover, for the scan times under consideration, the differences between exact and polynomial predictions are certainly small compared to expected sensor measurement errors.

While 'exact' system models for target trajectories are not used in the tracker *per se*, they are used for the collection of tracking 'answers' which are exchanged between tracking systems or between trackers and other elements in the full Sim89 environment. (This 'Handover' issue is discussed in more detail in the next section.)

Multi-Target Tracking

Given the preceeding prescription for estimating the state of a single target from a sequence of 2D observations, the central issue in multi-target tracking is that of associating observations with tracks or observations on one scan with those of a subsequent scan (e.g., in Figure 19.2.2, which 'x' is paired with which 'o'). There are, in a sense, two extreme schemes for attempting this track$\leftrightarrow$hit association:

Track Splitting : All *plausible* associations of existing tracks with new data are maintained in the updated track file.

Optimal Associations : Only a single global association of tracks with incoming data is selected.

Each of these prescriptions has advantages and disadvantages.

The Track Splitting model is robust in the sense that the correct track$\leftrightarrow$hit association is very likely to be generated and maintained at any

step in track processing. The track extension task is also extemely 'localized', in the sense that splittings of any one track can be done independently of those for other tracks. This makes concurrent implementations of Track Splitting quite simple. The primary objections to track splitting are twofold:

1. Track Splitting does not provide an easily interperted 'answer' as to the actual nature of the underlying scenario.

2. Without (sometimes) elaborate mechanisms to identify and delete poor or duplicate entries in the track file, Track Splitting leads to an essentially exponential explosion in the track file size in dense target environments.

Track Splitting is particularly useful at early times in the evolution of a target scenario, when the available data are too sparse to determine the 'correctness' of any candidate track. As is discussed in more detail in Section 19.2.4. the primary role of Track Splitting in the Sim89 tracker is that of track initiation.

The Optimal Association prescription is orthogonal to Track Splitting in the sense that the single 'best' pairing is maintained in place of all plausible pairings. This best Track$\rightarrow$Hit association is determined by a global optimization procedure, as follows. Let $S_A = \{a_i\}$ and $S_B = \{b_j\}$ be two lists of items (e.g., actual data and predicted data values). Let

$$d_{ij} \equiv d[a_i, b_j] \tag{19.3}$$

be a cost for associating items a_i and b_j (e.g., the cartesian distance between predicted and actual data positions for the data coordinates defined above). The optimal association of the two lists is that particular permutation,

$$i \rightarrowtail j = \Pi(i) \tag{19.4}$$

such that the total association score,

$$S_{TOT} \equiv \sum_i d[a_i, b_{\Pi(i)}] \tag{19.5}$$

is minimized ovar all permutations Π of Equation 19.4.

Leaving aside, for now, the question of computational costs associated with the minimization of Equation 19.5, there are some fundamental difficulties associated with the use of Optimal Associators in multi-target tracking models. In particular

1. Optimal associators perform poorly if the two lists S_A and S_B do not correspond to the *same* sets of underlying targets.

2. Poor entries in the cost matrix $\{d_{ij}\}$ can lead to global distortions of the globally optimal association.

The purely mathematical solution to the problem of minimizing Equation 19.5 need not be a *reasonable* solution to the problem of finding the best pairings of the two lists, and the points just noted are canonical failure modes by which blind optimal associations yield miserable solutions to 'real' problems. Nonetheless, if one requires the ultimate output of the tracking function to be the single 'best guess' as to the actual nature of the underlying target scenario, then some suitably massaged form of Optimal Association is clearly required.

19.2.4 Algorithm Overview

The manner in which the elements of the preceeding section are combined into an overall tracking algorithm is governed by a few fundamental assumptions

A1. For substantial fractions of the scenarios under consideration, the actual trajectories of the targets of interest are not fully constrained.

A2. The densities of targets are not so large as to preclude the separation of individual targets over some/most of the time interval in question.

The first assumption requires *Stereo Tracking*. Target motion along the line of sight from any one sensor is assumed to be sufficiently 'unknowable' that cooperative tracking from pairs of sensors is required to determine the full 3D state of a target. The second assumption is, in essence, a statement of limitations of the entire Sim89 approach. The Sim89 tracker is ultimately a *Point Tracker* in the sense that the algorithm attempts to associate targets with individual data points provided by the sensors. This approach makes sense only if the sensor can actually resolve individual targets for most/much of the time. If the nature of the sensor and underlying targets is such that a cluster of real targets is seen only as an ill-defined 'clump' by the sensor, the overall Sim89 prescription is inappropriate, and more imaginative solutions to the tracking problem (e.g., track density estimation by neural network techniques) would be required.

Figure 19.2: I need a figure caption here

Given these assumptions on the nature of the tracking problem, the overall form of the Sim89 tracking model is as illustrated in Figure 19.2.4. The basic elements are a pair of 2D trackers, each receiving and processing data from its own sensor, a 3D tracking module which combines information from the two 2D systems, and a 'Handover' module. The Handover module controls both the manner in which the 3D tracker sends its answers to whomever is listening and the way in which tracks from other systems are entered into the existing 3D track files. The following subsections provide brief descriptions of the algorithms used in each of these component sub-tasks.

19.2.5 2D Mono Tracking

The primary function of the 2D tracking module is fairly straightforward: given 2D data sets arriving at reasonably regular time intervals (scans) from the sensors, construct a big set of 'all' plausible 2D tracks linking these observations from scan-to-scan. This is done by way of a simple Track-Splitting module. The tracks from the two 2D trackers in Figure 19.2.4 are the fundamental inputs to the 3D track initialization algorithm described in the next subsection.

The adoption of Track Splitting in place of Optimal Association for the 2D trackers is largely a consequence of assumption (A1) above. Without a restrictive model for the (unseen) motion along the sensor line of sight, the information available to the 2D tracker is not sufficient to differentiate among plausible global track sets through the data points. Instead, the 2D tracker attempts to form all plausible 'tracks' through its own 2D data set,

Figure 19.3: I need a figure caption here

with the distinction between real and phantom tracks deferred to the 3D track initiation and association modules described in the next section.

With the receipt of a new data set from the sensors, the action of the 2D tracker consists of several simple steps:

1. Extend existing tracks to new data, as possible
2. Redistribute the global track file among the nodes (concurrent execution)
3. Collect and sort 'good' 2D tracks into a global 2D report list
4. Initialize new entries for the track file.

This algorithm flow is illustrated in Figure 19.2.5. Discussions of the track file redistribution in Step 2 (as well as concurrent aspects of the other steps) are deferred. The following subsections describe track extensions, report collection and track initiation.

2D Track Extensions

An item in the 2D track file is described by an eight-component state vector

$$\vec{x} \equiv \begin{pmatrix} \vec{x}_y \\ \vec{x}_z \end{pmatrix} \tag{19.6}$$

where the component vectors on the RHS of Equation 19.6 are four-element kinematic state vectors as defined for Equation 19.1, referred to the standard measurement axes

$\hat{y}$: Unit vector along the projected sensor velocity.

$\hat{z}$: Unit vector along the normal to the orbital plane.

defined in Section 19.2.2. (Recall that the 'massaged' data used in the tracker are the projections of the true target positions onto these axes.) The axis $\hat{y}$ is non-inertial, so that the state in Equation 19.6 has substantial 'contaminations' from motion of the sensor.

In principle, each track described by Equation 19.6 has an associated covariance matrix with 36 independent elements. In order to reduce the storage and CPU resource requirements of 2D tracking, a simplyfying assumption is made. The measurement error matrix for a 2D datum

$$\vec{z} \equiv \begin{pmatrix} \rho_y \\ \rho_z \end{pmatrix} \tag{19.7}$$

is taken to have the simple form

$$M \equiv E[\vec{z}\vec{z}^T] \equiv \begin{bmatrix} \sigma_M^2 & 0 \\ 0 & \sigma_M^2 \end{bmatrix} \tag{19.8}$$

with the *same* effective value σ_M^2 used to describe the measurement variance for each projection, and no correlation of the measurement errors. The assumption in Equation 19.7–Equation 19.8 is reasonable, provided the effective measurement error σ_M is made large enough, and reduces the number of independent components in the covariance matrix from 36 to 10.

The central task of the 2D track extension module is to find all plausible track$\leftrightarrow$hit associations, subject to a set of criteria which define 'plausible'. The primary association criterion is based on the track association score

$$S \equiv |\vec{y}|^2/[2(P_{xx} + \sigma_M^2)] \tag{19.9}$$

where P_{xx} is the variance of the prediced data position along a reference axis and

$$\vec{y} \equiv \vec{z}_{DATA} - \vec{z}_{PREDICTED} \tag{19.10}$$

is the diference between the actual data value and that predicted by Equatihon 19.6 for the time of the datum. Equation 19.9 is simply a dimensionless measure of the size of the mismatch in Equation 19.10, normlized by the expected prediction error.

The first step in limiting Track$\leftrightarrow$Hit associations is a simple cut $S < S_{MAX}$ on the association score of Equation 19.9. For the dense, multi-target

environments used in Sim89, this simple cut is not sufficiently restrictive, and a variety of additional heuristic cuts are made. The most important of these are

1. Approximate Data Linearity : The data point of a proposed association must represent 'forward' motion relative to the last two data included in the track.

2. Vertical Motion Cuts (Optional) : The projected 2D motion must be consistent with underlying 3D motion away from the earth.

These cuts are particulary important at early stages in Boost-Phase tracking, when scan-to-scan target motion is not large compared to measurement errors, and the sizes of the prediction gates according to the tracking filter are large.

The actual track scoring cut is a bit more complicated than the preceeding paragraph implies. Let S_{EXT} denote the nominal extension score of Equation 19.9. In addition, define a cumulative association score S_{TOT} which is updated on associations in a fading memory fashion

$$S_{TOT} \mapsto \alpha S_{TOT} + (1-\alpha) S_{EXT} \tag{19.11}$$

with (typically) $\alpha \approx 0.5$. An extension is accepted only if S_{EXT} is below some nominal cutoff (typically $3\text{–}4\sigma$) *and* S_{TOT} is below a more restrictive cut ($2\text{–}3\sigma$). This second cut prevents creation of poor tracks with barely acceptable extension scores at each step.

The preceeding rules for Track↔Hit associations define the basic 2D track extension formalism. There are, however, two additional problems which must be addressed:

- The prescription can generate duplicate tracks (meaning identical associated data sets over some number of scans).

- The size of the track file can increase without bounds.

These problems are particulary acute in dense target environments.

In regards to the first problem, two entries in the track file are said to be *equivalent* if they involve the same associated data points over the past four scans. If an equivalent track pair is found in the track file, the track with a higher cumulative score S_{TOT} is simply deleted. The natural mechanism for track deletion in a track-splitting model is based on the track's data

association history. If no data items give acceptable association scores over some preset number of scans (typically 0–2), the track is simply discarded.

The equivalent-track merging and poor track deletion mechanisms are not sufficient to prevent track file 'explosions' in truly dense environments. A final track-limiting mechanism is simply a hard cutoff on the number of tracks maintained for any item in the data set (this cut is typically $N_{MAX} = 8$). If more than N_{MAX} tracks give acceptable association scores to a particular datum, only the N_{MAX} pairings with the lowest total association scores S_{TOT} are kept.

The complexity of the track extension algorithm is nominally $N_{DATA} \times N_{TRACKS}$ for N_{DATA} new data items and N_{TRACKS} existing tracks. This $O(N^2)$ computational burden is easily reduced to something closer to $O(NlogN)$ by sorting both the incoming data and the predicted data values for existing tracks.

2D Report Formation

The *Report Formation* subtask of the 2D tracker collects/organizes established 2D tracks into a list to be used as input for 3D track initiations, where 'established' simply means tracks older than some minimum cutoff age (typically seven hits). The task of initiating 3D tracks from lists of 2D tracks consists of two parts:

1. Determining which items from the two lists are to be associated.

2. Constructing 3D state information for associated pairs.

The second task is straightforward geometry. The Report function for 2D tracking is intended to aid in the more difficult association task.

The essential element in 2D + 2D $\rightarrow$ 3D associations is the so-called *Hinge Angle* illustrated in Figure 19.2.5. Consider a single target viewed *simultaneously* by two different sensors. Assuming that each 2D tracker knows the orbits of the other tracker's sensor, each tracker can independently reconstruct two reference planes in 3D inertial space:

SSE : The plane containing the two sensors and the center of the earth.

SST : The plane containint the two sensors and the target.

The hinge angle

$$\chi = \chi(t) \equiv \Theta(SSE, SST) \tag{19.12}$$

Figure 19.4: I need a figure caption here

is simply the angle between these two planes.

Once the time for the 2D report has been specified, the steps involved in the Report function are relatively straightforward:

1. Select all tracks satisfying the minimum age requirement.
2. Use the state model in Equation 19.6 to propagate these tracks to the reference time.
3. Evaluate both χ *and* its time derivative $d\chi/dt$ at the reference time
4. Sort the list of reported tracks by χ values.

The state model in Equation 19.6 not only provides the mechanism for synchronizing the report items but also the additional variable $d\chi/dt$ which ultimately aids in the associations of report lists from two 2D systems.

Track Initialization

The algorithm described in Section 19.2.5 is only applicable for extending tracks which already exist in the track file. The creation of new entries is done by a separate track initiation function.

The track initiator involves little more than searches for nearly colinear triples of data over the last three scans. A triplet

$\vec{z}_1$: 2D Datum, Current Scan

$\vec{z}_2$: 2D Datum, Prior Scan

$\vec{z}_3$: 2D Datum, Two Scans Back

is a candidate new track if

$$S_{INIT} \equiv \left(\frac{1}{12\sigma_M^2}\right)\left[\frac{\vec{z}_1 - \vec{z}_2}{t_1 - t_2} - \frac{\vec{z}_2 - \vec{z}_3}{t_2 - t_3}\right] \leq S_{CUT} \qquad (19.13)$$

where the cutoff is generally fairly loose (e.g., $S_{CUT} \approx 16$). In addition, an number of simple heuristic cuts (maximum speed) are applied.

The initiator searches for all approximately linear triples over the last three scans, subject to the important additional restriction that *no* initiations to a particular item $\vec{z}_1$ of the current data set are attempted if *any* established track (minimum age cut) already exists ending at that datum. The nominal $O(N^3)$ complexity of the initiator is reduced to approximately $O(N \log N)$ by exploiting the sorted dature on the incoming data sets.

19.2.6 3D Tracking

Unlike the 2D tracking module, the 3D stereo tracker attempts to construct a single track for each (perceived) underlying target. The fundamental algorithm element for this type of tracking is the optimal associator described in Section 19.2.3. A single pass through the 3D tracker utilizes optimal associations for two distinct sub-tasks:

Track Extensions Data from a sensor are associated with predicted data positions for existing 3D tracks. This task is performed twice per scan, once for each 2D substsyem of Figure 19.2.4.

Track Initiation 2D report lists are associated and new 3D tracks are initiated for correlations to data points not used in the preceeding track extension step.

As was noted in Section 19.2.3, a canonical problem with optimal associators is the possibility of globally poor associations due to incompatible lists or poor distance information for some of the entries in either lists. These problems are addressed as follows:

1. Evaluations of individual distances for the cost matrix include relatively restrictive cuts which prohibit poor associations. The nominal cost for such associations is set to an 'infinite' token and the association is simply ignored if selected in the course of minimization of Equation 19.5.

2. Additional 'Quality Control' modules are used to assess feasibility of proposed associations and tracks failing the quality constraints are deleted from the system.

The associators for track extensions and initiations and the quality control modules are described in the following Subsections.

Track Extension Associations

Given a list of existing 3D tracks and a set of observations from a particular sensor, the track extension task nominally consists of three basic steps:

1. Evaluate the cost matrix for associating individual tracks with entries from the data set.

2. Find the optimal association by minimization tof the global score of Equation 19.5.

3. Update each track according to its associated data item, using a full 3D kinematic Kalman filter.

This nominal algorithm is hopelessly slow. If the track file and data set each have a characteristic size N, Step 1 requires N^2 operations and Step 2 requires N^3. The reduction of the unacceptable polynomial complexities of Step 2 and Step 3 to something approaching $O(N \log N)$ is done as follows.

A list of predicted data values for existing tracks is evaluated and is sorted using the same key as was used sorting the data set. The union of the sorted prediction and data sets is then broken into some number of gross sub-blocks, defined by appropriately large gaps in values of the sorting key. This reduces the single large association problem into a number of smaller sub-problems.

For each subproblem, a pruned distance matrix is evaluated, subject to two primary constraints:

- Individual associations are considered only if the association score is less than some maximum allowed score value.

- The number of data value to be associated with any one track prediction is limited to some preset maximum.

The score for an individual association is the distance between prediction and datum, weighted by the prediction uncertainty:

$$d[\text{Track,Datum}] \equiv \frac{\Delta z^2}{\hat{P}_{zz}} + \frac{\Delta y^2}{\hat{P}_{yy}} \tag{19.14}$$

where, for $i = y, z$,

$$\Delta x_i \equiv x_i[\text{Datum}] - x_i[\text{Prediction}] \tag{19.15}$$

and $\hat{P}_{ii}$ is the predicted variance for Equation 19.15 according to the 3D tracking filter. The score is essentially a χ^2 for the proposed association, and the cutoff value is typically of order $\chi^2 \leq 16$. The maximum allowed number of associations for any single prediction is typically $N_{MAX} = 6$–8. If more than N_{MAX} data give acceptable association scores, the possible pairings are sorted by the association score and only the N_{MAX} best fits (lowest scores) are kept.

The preceeding scoring algorithm leads to a (generally) sparse distance matrix for the large subblocks defined through gaps in the sorting keys. The next step in the algortihm is a quick block diagonalization of the distance matrix through appropriate reorderings of the rows and columns. By this point, the original large association problem has been reduced to a large number of modest sized subproblems and Munkres algorithm for minimizing the global cost in Equation 19.5 is (finally) used to find the optimal pairings.

19.3 SIM 8X

*** Contribution needed from G. C. Fox

Chapter 20

MOVIE

20.1 System

*** Contribution needed from W. Furmanski

20.2 Map Separates

*** Contribution needed from W. Furmanski

20.3 Virtual Reality

*** Contribution needed from W. Furmanski

Chapter 21

Conclusions

21.1 Lessons, Research and Education

*** Contribution needed from G. C. Fox

Chapter 22

Appendices

22.1 Reports

*** Contribution needed from G. C. Fox

22.2 An Introduction to the CITLIB File-server

CITLIB (also known as C3PLIB) is an implementation of a form of technology transfer electronically. To be specific, the transfer takes place over the network by electronic mail.

CITLIB runs the Argonne National Laboratory NETLIB software which is an electronic mail-response program. It contains an electronic archive which is structured in a hierarchical fashion, as a tree of directories and sub-directories. There is an index describing the content of each node in the directory tree.

CITLIB contains information relevant to computational science and parallel processing. Currently, CITLIB contains abstracts and information for the journal *Concurrency, Practice and Experience*, C^3P bibliographies and abstracts, C^3P reports, news, and codes (such as programs for the books *Solving Problems on Concurrent Processors, Vol. 1* [Fox:88a] and *Vol. 2* [Angus:90a] (see Section **??**) and the benchmark suite for the Caltech Performance Evaluation Project (see Section 13).

To access CITLIB, just send a one-line mail message "send index" (without the quotation marks) to {citlib@caltech.edu} or {citlib@caltech.bitnet}. CITLIB, in response to your e-mail, will send to you a detailed description of what you can get from the file server and how.

22.2.1 Credits

Alex W. Ho is the manager and maintainer of the CITLIB archive, while Mark F. Beauchamp is responsible for implementing the NETLIB software for CITLIB.

22.3 Biographic Information

*** Contribution needed from G. C. Fox

22.4 Sponsors

*** Contribution needed from G. C. Fox

List of Tables

List of Figures

Bibliography

[Wilson:74a] Wilson, K. G. "Confinement of quarks," *Phys. Rev. D*, 10:2445, 1974.

[Brooks:83a] Brooks, E., Fox, G., Otto, S., Randeria, M., Athas, W., DeBenedictis, E., Newton, N., and Seitz, C. "Glueball mass calculations on an array of computers," *Nucl. Phys. B.*, 220(FS8):383, 1983. Caltech Report C3P-027.

[Otto:83a] Otto, S., and Randeria, M. "Modified action glueballs," *Nucl. Phys. B.*, 225(FS9):579, 1983. CALT-68-1040. Caltech Report C3P-330.

[Otto:84a] Otto, S. W., and Stack, J. D. "SU(3) heavy-quark potential with high statistics," *Phys. Rev. Lett.*, 52:2328, 1984. Caltech Report C3P-067.

[Otto:85b] Otto, S. W., and Stolorz, P. "An improvement for glueball mass calculations on a lattice," *Physics Letters B*, 151(5,6):428, February 1985. Caltech Report C3P-343.

[Patel:85a] Patel, A., Otto, S., and Gupta, R. "The non-perturbative β-function for SU(2) lattice gauge theory," *Phys. Lett. B.*, 159:143, 1985. CALT-68-1261, Calculation on the Mark I 64 Node. Caltech Report C3P-216.

[Fucito:84a] Fucito, F., Kinney, R., and Solomon, S. "On the phase diagram of finite temperature QCD in the presence of dynamical quarks," *Nucl. Phys. B.*, 248:615, 1984. CALT-68-1189. Caltech Report C3P-333.

[Fucito:85b] Fucito, F., and Solomon, S. "The chiral symmetry restoration transition in the presence of dynamical quarks," *Phys. Rev. Lett.*, 55:2641, 1985. CALT-68-1124. Caltech Report C3P-331.

[Fucito:85c] Fucito, F., and Solomon, S. "Finite temperature QCD in the presence of dynamical quarks," *Nucl. Phys. B.*, 253:727, 1985. BNL 34784 CALT-68-1127. Caltech Report C3P-332.

[Fucito:85d] Fucito, F., and Solomon, S. "On the order of the deconfining transition for finite temperature QCD in the presence of dynamical quarks," *Phys. Rev. D*, 31:1460, 1985. CALT-68-1285. Caltech Report C3P-334.

[Fucito:86a] Fucito, F., Moriarty, K. J. M., Rebbi, C., and Solomon, S. "The hadronic spectrum with dynamical Fermions," *Physics Letters B*, 172(235), May 1986. Caltech Report C3P-341.

[Flower:85a] Flower, J. W., and Otto, S. "The field distribution in SU(3) lattice gauge theory," *Phys. Lett. B*, 160:128, 1985. Caltech Report C3P-178.

[Flower:86b] Flower, J., and Otto, S. W. "Scaling violations in the heavy quark potential," *Phys. Rev.*, 34:1649, 1986. CALT-68-1340 DOE Research and Development Report - High Energy Physics calculations on the Hypercube. Caltech Report C3P-262.

[Furmanski:87c] Furmanski, W., and Kolawa, A. "Yang-Mills vacuum - an attempt of lattice loop calculus," *Nucl. Phys. B*, 291:594, 1987. CALT-68-1330. Caltech Report C3P-335.

[Stolorz:86b] Stolorz, P. "Microcanonical renormalization group for SU(2)," *Phys. Lett. B*, 172:77, 1986. Caltech Report C3P-741.

[Flower:87e] Flower, J. "Baryons on the lattice," *Nucl Phys B*, 289(2):484–504, 1987. Caltech Report C3P-319b.

[Chiu:88a] Chiu, T. W. "Schwinger model on the random block lattice." Technical Report C3P-647, California Institute of Technology, July 1988. To be published in Physics Letters B, 1989.

[Chiu:88c] Chiu, T. W. "Fermion propagators on a four dimensional random-block lattice," *Physics Letters B*, 206(3):510–516, January 1988. Caltech Report C3P-507.

[Chiu:88e] Chiu, T. W. "Vacuum polarization on 4-d random block lattice." Technical Report C3P-693, California Institute of Technology, 1988.

[Chiu:88f] Chiu, T. W. "Field theory on the random block lattice." Technical Report C3P-694, California Institute of Technology, 1988.

[Chiu:89a] Chiu, T. W. "Random coupling models of lattice Fermion." Technical Report C3P-813, California Institute of Technology, August 1989.

[Ding:89a] Ding, H. Q., Baillie, C. F., and Fox, G. C. "Calculation of the heavy quark potential at large separation on a hypercube parallel computer." Technical Report C3P-779, California Institute of Technology, August 1989. For publication information, see Ding:90b.

[Baillie:89e] Baillie, C. F., Brickner, R. G., Gupta, R., and Johnsson, L. "QCD with dynamical Fermions on the Connection Machine," in *Proceedings of Supercomputing '89*, pages 2–9. ACM Press, November 1989. IEEE Computer Society and ACM SIGARCH, Reno, Nevada. Caltech Report C3P-786.

[Metropolis:53a] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. "Equation of state calculations by fast computing machines," *J. Chem. Phys.*, 21:1087–1091, 1953.

[Weingarten:81a] Weingarten, D. H., and Petcher, D. N. "Monte Carlo integration for lattice gauge theories with Fermions," *Phys. Lett. B*, 99:333, 1981.

[Fucito:81a] Fucito, F., Marinari, E., Parisi, G., and Rebbi, C. "A proposal for Monte Carlo simulations of Fermionic systems," *Nucl. Phys. B*, 180([FS2]):369, 1981.

[Duane:87a] Duane, S., Kennedy, A. D., Pendleton, B. J., and Roweth, D. "Hybrid Monte Carlo," *Physics Letters B*, 195:216–220, 1987.

[Duane:85a] Duane, S. "Stochastic quantization versus the microcanonical ensemble: Getting the best of both worlds," *Nucl. Phys. B*, 257:652–662, 1985.

[Callaway:83a] Callaway, D., and Rahman, A. "Lattice gauge theory in the microcanonical ensemble," *Phys. Rev. D*, 28:1506–1514, 1983.

[Polonyi:83a] Polonyi, J., and Wyld, H. W. "Microcanonical simulation of Fermionic systems," *Phys. Rev. Lett.*, 51:2257–2260, 1983.

[Parisi:81a] Parisi, G., and Wu, Y. "Perturbation theory without gauge fixing," *Sci. Sin.*, 24:483–496, 1981.

[Batrouni:85a] Batrouni, G. G., Katz, G. R., Kronfeld, A. S., Lepage, G. P., Svetitsky, B., and Wilson, K. G. "Langevin simulations of lattice field theories," *Phys. Rev. D*, 32:2736–2747, 1985.

[Gupta:88a] Gupta, R., DeLapp, J., Batrouni, G., Fox, G. C., Baillie, C., and Apostolakis, J. "The phase transition in the 2-d XY model," *Phys. Rev. Lett.*, 61:1996, 1988. Caltech Report C3P-643.

[Fox:89n] Fox, G. C. "Parallel computing comes of age: Supercomputer level parallel computations at Caltech," *Concurrency: Practice and Experience*, 1(1):63–103, September 1989. Caltech Report C3P-795.

[Eichten:80a] Eichten, E., Gottfried, K., Kinoshita, T., Lane, K. D., and Yan, T. M. "Charmonium: Comparison with experiment," *Phys. Rev. D*, 21:203, 1980.

[Ding:89b] Ding, H.-Q. "The Mark IIIfp Hypercube: Performance of a quantum chromo-dynamics code." Technical Report C3P-799, California Institute of Technology, June 1989. For publication information, see Ding:90c.

[Hillis:85a] Hillis, W. D. *The Connection Machine.* MIT Press, Cambridge, Mass., 1985.

[Hillis:87a] Hillis, W. D. "The Connection Machine," *Scientific American*, page 108, June 1987.

[Lenz:20a] Lenz, W. *Z. Phys.*, 21:613, 1920.

[Ising:25a] Ising, E. *Z. Phys.*, 31:253, 1925.

[Onsager:44a] Onsager, L. "A 2d model with an order-disorder transition," *Phys. Rev.*, 65:177, 1944.

[Wilson:80a] Wilson, K. G. "Monte Carlo renormalization group on the 3d Ising Model," in G. T. Hooft, editor, *Recent Developments in Gauge Theories.* Plenum Press, New York, 1980. Cargese, 1979.

[Swendsen:79a] Swendsen, R. H. "Monte Carlo renormalization group," *Phys. Rev. Lett.*, 42:859, 1979.

[Pawley:84a] Pawley, G. S., Swendsen, R. H., Wallace, D. J., and Wilson, K. G. "Monte Carlo renormalization group calculation of the critical behavior in the simple cubix Ising Model," *Phys. Rev. B*, 29:4030, 1984.

[Swendsen:87a] Swendsen, R. H., and Wang, J. "Nonuniversal critical dynamics in Monte Carlo simulations," *Phys. Rev. Lett.*, 58(2):86–88, 1987.

[Wolff:89b] Wolff, U. "Continuum behavior in the lattice O(3) nonlinear Sigma model," *Phys. Lett. B*, 222:473, 1989.

[Baillie:88h] Baillie, C. F., and Barish, K. "Spin operators for the 3-d Ising model." Technical Report C3P-648, California Institute of Technology, July 1988.

[Potts:52a] Potts, R. B. "Some generalized order-disorder transformations," *Proc. Camb. Phil. Soc.*, 48:106, 1952.

[Wu:82a] Wu, F. Y. "The Potts model," *Reviews of Modern Physics*, 54:235–275, 1982.

[Baillie:89l] Baillie, C. F., and Coddington, P. D. "A comparison of cluster algorithms for Potts models." Technical Report C3P-835, California Institute of Technology, October 1989. For publication information, see Baillie:90m.

[Heermann:89a] Heermann, D. W., and Burkitt, A. N., "System size dependence of the autocorrelation time for the Swendsen-Wang Ising Model," 1989. Wuppertal preprint WU B 89-25.

[Creutz:87a] Creutz, M. "Over-relaxation and Monte Carlo simulation," *Phys. Rev. D*, 36:515, 1987.

[Brown:87a] Brown, F., and Woch, T. J. "Over-relaxed heat bath and Metropolis algorithms for accelerating pure gauge Monte Carlo calculations," *Phys. Rev. Lett.*, 58:2894, 1987.

[Polyakov:75a] Polyakov, A. M. "Interaction of goldstone particles in 2d," *Phys. Lett. B*, 59:79, 1975.

[Brezin:76a] Brezin, E., and Zinn Justin, J. "Spontaneous breakdown of continuous symmetries near 2d," *Phys. Rev. B*, page 3110, 1976.

[Wolff:90a] Wolff, U. "Asymptotic freedom and mass generation in the O(3) nonlinear Sigma model," *Nucl. Phys. B*, 334:581, 1990.

[Johnson:87a] Johnson, P. C., and Jackson, R. "Frictional-collisional constitutive relations for granular materials, with application to plane shearing," *Journal of Fluid Mechanics*, 176:67–93, 1987.

[Gutt:89a] Gutt, G. M. *The Physics of Granular Systems.* PhD thesis, California Institute of Technology, May 1989. Caltech Report C3P-785.

[Cundall:79a] Cundall, P. A., and Strack, O. D. L. "A discrete numerical model for granular assemblies," *Geotechnique*, 29(1):47–65, 1979.

[Haff:87a] Haff, P. K. "Micromechanical aspects of sound waves in granular materials," in *Proceedings of Solids Transport Contractor's Review*, pages 46–67. Department of Energy, September 1987. Held in Pittsburgh, Pennsylvania (September 17–18, 1987).

[Haff:87b] Haff, P. K., and Werner, B. T. "Particulate and multiphase processes," in T. Ariman and T. N. Verziroglu, editors, *Colloidal and Interfacial Phenomena,* **3**, page 483. Washington, D. C., 1987.

[Walton:84a] Walton, O. R. "Computer simulation of particulate flow," *Energy and Technology Review*, pages 24–36, May 1984. (Lawrence Livermore National Laboratory).

[Werner:87a] Werner, B. T. *A Physical Model of Wind-Blown Sand Transport.* PhD thesis, California Institute of Technology, 1987. Caltech Report C3P-425.

[Frisch:86a] Frisch, U., Hasslacher, B., and Pomeau, Y. "Lattice-gas automata for the Navier-Stokes equations," *Phys. Rev. Lett.*, 56:1505–1508, 1986.

[Margolis:86a] Margolis, N., Tommaso, T., and Vichniac, G. "Cellular-automata supercomputers for fluid-dynamics modeling," *Physical Review Letters*, 56(16):1694–1696, 1986.

[Haff:83a] Haff, P. K. "Grain flow as a fluid-mechanical phenomena," *Journal of Fluid Mechanics*, 134:401–430, 1983.

[Hui:84a] Hui, K., Haff, P. K., Ungar, J. E., and Jackson, R. "Boundary conditions for high shear rate grain flows," *Journal of Fluid Mechanics*, 145:223–233, 1984.

[Tuzun:82a] Tuzun, U., and Nedderman, R. M. "An investigation of the flow boundary during steady-state discharge from a funnel-flow bunker," *Powder Technology*, 31:27–43, 1982.

[Seitz:85a] Seitz, C. L. "The Cosmic Cube," *Communications of the ACM*, 28(1):22, 1985.

[Tuazon:85a] Tuazon, J. O., Peterson, J. C., Pniel, M., and Lieberman, M. "Caltech/JPL hypercube concurrent processor," in *IEEE 1985 Conference on Parallel Processing*, August 1985. St. Charles, Illinois. Caltech Report C3P-160.

[Brooks:82b] Brooks, E. "The Laplace equation on node NNCP." Technical Report C3P-010, California Institute of Technology, September 1982.

[Newton:82a] Newton, M. "An analysis of a parallel implementation of the fast Fourier transform." Technical report, California Institute of Technology, November 1982. Caltech Computer Science Document 5057:DF:82.

[Salmon:86b] Salmon, J. "The FFT and the N-body problem on the hypercube." Technical Report C3P-297.7, California Institute of Technology, November 1986. Support material for Second Hypercube class.

[Salmon:84b] Salmon, J. C. "Binary gray codes and the mapping of a physical lattice into a hypercube." Technical Report C3P-051, California Institute of Technology, January 1984.

[Johnson:86a] Johnson, M. A. *Concurrent Computation and its Application to the Study of Melting in Two Dimensions.* PhD thesis, California Institute of Technology, 1986. Caltech Report C3P-268.

[Johnson:85a] Johnson, M. A. "The interrupt-driven communication system." Technical Report C3P-137, California Institute of Technology, 1985. Unpublished.

[Peterson:85a] Peterson, J. C., Tuazon, J. T., Lieberman, D., and Pniel, M. "The Mark III hypercube ensemble concurrent computer," in *IEEE*

1985 Conference on Parallel Processing, August 1985. St. Charles, IL. Caltech Report C3P-151.

[Johnson:86c] Johnson, M. A. "The specification of CrOS III." Technical Report C3P-253, California Institute of Technology, February 1986.

[Kolawa:86d] Kolawa, A., and Zimmerman, B. "CrOS III Manual." Technical Report C3P-253b, California Institute of Technology, September 1986.

[Meier:84b] Meier, D. L. "C3PO: a proposed general purpose intermediate host program." Technical Report C3P-087, California Institute of Technology, July 1984.

[Salmon:87a] Salmon, J. "CUBIX: Programming hypercubes without programming hosts," in M. T. Heath, editor, *Hypercube Multiprocessors*, pages 3–9. SIAM, Philadelphia, 1987. Caltech Report C3P-378.

[Flower:86c] Flower, J., and Williams, R. "PLOTIX — a graphical system to run CUBIX and UNIX." Technical Report C3P-285, California Institute of Technology, May 1986.

[Salmon:86a] Salmon, J., and Hogan, C. "Correlation of QSO absorption lines in universes dominated by cold dark matter," *Monthly Notices of the Royal Astronomical Society*, 221:93, 1986. Caltech Report C3P-211.

[Goldsmith:86a] Goldsmith, J., and Salmon, J. "Static and dynamic database distribution for graphics ray tracing on the hypercube." Technical Report C3P-360, California Institute of Technology, 1986.

[Goldsmith:87a] Goldsmith, J., and Salmon, J. "Automatic creation of object hierarchies for ray tracing," *IEEE CG and A*, 14:14–20, 1987. Caltech Report C3P-295.

[Koller:88b] Koller, J. *The MOOS II Manual.* California Institute of Technology, 1988. Caltech Report C3P-662.

[Flower:87a] Flower, J., Otto, S., and Salama, M. "Optimal mapping of irregular finite element domains to parallel processors." Technical Report C3P-292b, California Institute of Technology, August 1987. In Proceedings, Symposium on Parallel Computations and Their Impact on Mechanics, ASME Winter Meeting, Dec. 14–16, Boston, Mass.

[Fox:86i] Fox, G., and Furmanski, W. "Load balancing by a neural network." Technical Report C3P-363, California Institute of Technology, 1986.

[Fox:86b] Fox, G., and Furmanski, W. "Optimal communication algorithms on the hypercube." Technical Report C3P-314, California Institute of Technology, 1986.

[Williams:88a] Williams, R. D. "DIME: A programming environment for unstructured triangular meshes on a distributed-memory parallel processor," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1770–1787. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-502.

[ParaSoft:88a] ParaSoft, "EXPRESS: A communication environment for parallel computers." 27415 Trabuco Circle, Mission Viejo, CA 92692, 1988.

[Flower:87c] Flower, J. "A guide to debugging with NDB." Technical Report C3P-489, California Institute of Technology, December 1987.

[Parasoft:88f] ParaSoft, "PM: performance analysis for parallel computers," 1988. 27415 Trabuco Circle, Mission Viejo, CA 92692.

[Ikudome:90a] Ikudome, K., Fox, G. C., Kolawa, A., and Flower, J. W. "An automatic and symbolic parallelization system for a distributed memory parallel computer," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference, Volume II*, pages 1105–1114, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina. Caltech Report C3P-877.

[Lee:86a] Lee, R. "Mercury I/O library users' guide, C language edition." Technical Report C3P-301, California Institute of Technology, 1986.

[Seitz:88b] Seitz, C. L., Athas, W. C., Flaig, C. M., Martin, A. J., Seizovic, J., Steele, C. S., and Su, W.-K. "The architecture and programming of the Ametek series 2010 multicomputer," in G. C. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, pages 33–36. ACM Press, 11 West 42nd Street, New York, NY 10036, 1988.

[Accetta:86a] Accetta, M., and et al. "Mach: a new kernal foundation for UNIX development," in *Atlanta Usenix Conference*, June 1986.

[DSL:89a] *The Helios Operating System.* Distribute Software Limited, 670 Aztec West, Almondsbury, Bristol, England, 1989.

[Ahuja:86a] Ahuja, S., Carriero, N., and Gelernter, D. "Linda and friends," *Computer*, 19(8), August 1986.

[Gelernter:89a] Gelernter, D. "Multiple tuple spaces in Linda," in *Proceedings of Parallel Architectures and Languages Europes*, volume 2, page 366. Springer-Verlag, LNCS, June 1989.

[Boris:73a] Boris, J. P., and Book, D. L. "Flux-corrected transport, I. SHASTA, a fluid transport code that works," *J. Comp. Phys.*, 11:38, 1973.

[Oran:90a] Oran, E. S., Boris, J. P., Whaley, R. O., and Brown, E. F. "Exploring fluid dynamics on a Connection Machine," *Supercomputing Review*, pages 52–60, 1990.

[Gustafson:88a] Gustafson, J. L., Montry, G. R., and Benner, R. E. "Development of parallel methods for a 1024-processor hypercube," *SIAM J. Sci. Stat. Comput.*, 9(4):609–638, July 1988.

[Anderson:87a] Anderson, P. W. *Science*, 235:1196, 1987.

[Ding:90g] Ding, H.-Q., and Makivic, M. "Spin correlations of 2d quantum antiferromagnet at low temperatures and a direct comparison with neutron scattering experiments," *Physics Review Letters*, 64:1449, 1990. Caltech Report C3P-844.

[Maddox:90a] Maddox, J. "Towards explaining super conductivity," *Nature*, 344:485, 1990.

[Fox:88a] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., and Walker, D. W. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1988.

[Fox:84a] Fox, G., and Otto, S. "Algorithms for concurrent processors," *Physics Today*, 37(5):50, 1984. Caltech Report C3P-071.

[Ding:88d] Ding, H. Q. "A fast random number generator for Mark IIIfp." Technical Report C3P-629, California Institute of Technology, May 1988.

[Ding:89d] Ding, H.-Q., and Makivic, M. "Quantum spin calculation on the hypercube." Technical Report C3P-845, California Institute of Technology, November 1989. For publication information, see Ding:90k.

[Chakravarty:88a] Chakravarty, S., Halperin, B. I., and Nelson, D. "Low-temperature behavior of two-dimensional quantum antiferromagnet," *Phys. Rev. Lett.*, 60:1057, 1988.

[Auerbach:88a] Auerbach, A., and Arovas, D. P. "Spin dynamics in the square-lattice antiferromagnet," *Phys. Rev. Lett.*, 61:617, 1988.

[Endoh:88a] Endoh, Y., Yamada, K., Birgeneau, R. J., Gabbe, D. R., Jenssen, H. P., Kastner, M. A., Peters, C. J., Picone, P. J., Thurston, T. R., Tranquada, J. M., Shirane, G., Hidaka, Y., Oda, M., Enomoto, Y., Suzuki, M., and Murakami, T. "Static and dynamic spin correlations in pure and doped La_2 Cu O_4," *Phys. Rev. B*, 37:7443, 1988.

[Singh:89a] Singh, R. R. P., Fleury, P. A., Lyons, K. B., and Sulewski, P. E. "Quantitative determination of quantum fluctuations in the spin-1/2 planar antiferromagnet," *Phys. Rev. Lett.*, 62:2736, 1989.

[Birgeneau:71a] Birgeneau, J., Skalyo, Jr., J., and Shirane, G. "Critical magnetic scattering in K_2 Ni F_4," *Phys. Rev. B*, 3:1736, 1971.

[Birgeneau:90a] Birgeneau, R. J. "Spin correlations in the two dimensional $S = 1$ Heisenberg antiferromagnet," *Phys. Rev. B.*, 41:2514, 1990.

[Higgins:88a] Higgins, S., and Cowley, R. A. "The phase transition of a disordered antiferromagnet with competing interactions," *J. Phys. C*, 21:2215, 1988.

[Ding:90b] Ding, H.-Q., Baillie, C. F., and Fox, G. C. "Calculation of the heavy quark potential at large separation on a hypercube parallel computer," *Physical Review D*, 41(9):2912–2916, May 1990. Caltech Report C3P-779b.

[Mermin:66a] Mermin, N. D., and Wagner, H. "Absence of ferromagnetism or antiferromagnetism in one or two dimensional isotropic Heisenberg models," *Phys. Rev. Lett.*, 77:1133, 1966.

[DeJongh:74a] De Jongh, L. J., and Miedema, A. R. "Experiments on simple magnetic model systems," *Adv. Phys.*, 23:1, 1974.

[Landau:76a] Landau, D. P. "Finite-size behavior of the Ising square lattice," *Phys. Rev. B*, 13:2997, 1976.

[Riedel:69a] Riedel, E., and Wegner, F. "Scaling approach to anisotropic magnetic systems statics," *Z. Physik*, 225:195, 1969.

[Kosterlitz:73a] Kosterlitz, J. M., and Thouless, D. J. "Ordering, metastability and phase transitions in two-dimensional systems," *J. Phys. C*, 6:1181, 1973.

[Matsubara:56a] Matsubara, T., and Katsuda, K. "A lattice model of liquid Helium, I," *Prog. Theo. Phys.*, 16:569, 1956.

[Rogiers:79a] Rogiers, J., Grundke, E. W., and Betts, D. D. "The spin 1/2 XY model, III, analysis of high temperature series expansions of some thermodynamic quantities in two dimensions," *Can. J. Phys.*, 57:1719, 1979.

[DeRaedt:84a] De Raedt, H., De Raedt, B., and Lagendijk, A. "Thermodynamics of the two-dimensional spin-1/2 XY model," *Z Phys. B*, 57:209, 1984.

[Loh:85a] Loh, Jr., E., Scalapino, D. J., and Grant, P. M. "Monte Carlo studies of the quantum XY model in two dimensions," *Phys. Rev. B*, 31:4712, 1985.

[Battiti:89g] Battiti, R. *Multiscale Methods, Parallel Computation, and Neural Networks for Computer Vision.* PhD thesis, California Institute of Technology, November 1989. Caltech Report C3P-850.

[Furmanski:88c] Furmanski, W., and Fox, G. C. "Integrated vision project on the computer network," in E. Clementi and S. Chin, editors, *Biological and Artificial Intelligence Systems*, pages 509–527. ESCOM Science Publishers B.V., P. O. Box 214, 2300 AE Leiden, The Netherlands, 1988. Caltech Report C3P-623.

[Marr:76a] Marr, D., and T., P. "Cooperative computation of stereo disparity," *Science*, 195:283–287, 1976.

[Poggio:85a] Poggio, T., Torre, V., and Koch, C. "Computational vision and regularization theory," *Nature*, 317:314–319, 1985.

[Brandt:77a] Brandt, A. "Multilevel adaptive solutions to boundary value problems," *Mathematics of Computation*, 31:333–390, 1977.

[Stuben:82a] Stüben, K., and Trottenberg, U. "Multigrid methods: Fundamental algorithms, model problem analysis and applications," in *Multigrid Methods Proc.*, pages 1–176, Berlin, 1982. Springer-Verlag.

[Terzopoulos:86a] Terzopoulos, D. "Image analysis using multigrid relaxation methods," *IEEE Trans. Pattern Analysis Machine Intelligence*, 8:129, 1986.

[Marroquin:84a] Marroquin, J. L. "Surface reconstruction preserving discontinuities," *MIT A. I. Memo*, 792, 1984.

[Battiti:89d] Battiti, R. "A multiscale approach to surface reconstruction with discontinuity detection." Technical Report C3P-676b, California Institute of Technology, May 1989.

[Chan:86b] Chan, T. F., and Saad, Y. "Multigrid algorithms on the hypercube multiprocessor," *IEEE Trans. on Computers*, C-35(11), 1986.

[Battiti:90b] Battiti, R. "Real-time multiscale vision on a two-dimensional mesh of processors." Technical Report C3P-932, California Institute of Technology, June 1990. For publication information, see Battiti:91a.

[Horn:85a] Horn, B. K. P., and Brooks, M. J. "The variational approach to shape from shading," *MIT A. T. Memo*, 813, 1985.

[Denker:86a] Denker, J. S., editor. *Neural Networks for Computing.* AIP, New York, 1986. AIP Conference Proceedings 151.

[Parker:82a] Parker, D. B. "Learning logic." Technical Report S81-64, Stanford University, 1982. Invention Report, File 1, Office of Technology Licensing.

[Rumelhart:86a] Rumelhart, D., Hinton, G., and Williams, R. "Learning internal representations by error propagation," in *Parallel Distributed Processing: Vol 1: Foundations.* MIT Press, 1986.

[Seinowski:87a] Seinowski, T. J., and Rosenberg, C. R. "Parallel networks that learn to pronounce English text," *Complex Systems*, 1(1):145–168, 1987.

[Denker:87a] Denker, J. S., Schwartz, D., Wittner, B., Solla, S., Howard, R., Jackel, L., and Hopfield, J. *Complex Systems*, 1:877, 1987.

[Kirkpatrick:83a] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. "Optimization by simulated annealing," *Science*, 220:671–680, May 1983.

[Dahl:87a] Dahl, E. D. "Accelerated learning using the generalized delta rule," in *International Conference on Neural Networks (IEEE)*, 1987.

[Parker:87a] Parker, D. B. "Optimal algorithms for adaptive networks: Second order back propagation, second order direct propagation, and second order hebbian learning," in *International Conference on Neural Networks (IEEE)*, 1987.

[Gullichsen:87a] Gullichsen, E., and Chang, E. "Pattern classificaiton by neural network: An experimental system for icon recognition," in *International Conference on Neural Networks (IEEE)*, 1987.

[Horn:81a] Horn, B. K. P., and Schunck, G. "Determining optical flow," *Artificial Intelligence*, 17:185–203, 1981.

[Enkelmann:88a] Enkelmann, W. "Investigations of multigrid algorithms for the estimation of optical flow fields in image sequences," *Computer Vision, Graphics and Image Processing*, 43:150–177, 1988.

[Glazer:84a] Glaser, F. "Multilevel relaxation in low-level computer vision," in A. Rosenfeld, editor, *Multiresolution Image Processing and Analysis*, pages 312–330. Springer-Verlag, 1984.

[Burt:84a] Burt, P. J. "The pyramid as a structure for efficient computation," in A. Rosenfeld, editor, *Multiresolution Image Processing and Analysis*, pages 6–35. Springer-Verlag, 1984.

[Battiti:88a] Battiti, R. "Collective stereopsis on the hypercube," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1000–1006. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-583.

[Baillie:89h] Baillie, C. F., Johnston, D. A., and Williams, R. D. "Crumpling in dynamically triangulated random surfaces with extrinsic curvature." Technical Report C3P-807, California Institute of Technology, July 1989. For publication information, see Baillie:90j.

[Baillie:90c] Baillie, C. F., Williams, R. D., and Johnston, D. A. "Crumpling dynamically triangulated random surfaces in higher dimensions," *Physics Letters B*, 243(4):358–364, January 1990. Caltech Report C3P-867.

[Baillie:90d] Baillie, C. F., Williams, R. D., and Johnston, D. A. "Non-universality in dynamically triangulated random surfaces with extrinsic curvature." Technical Report C3P-868, California Institute of Technology, March 1990. Accepted for publication in Modern Physics Letters A.

[Baillie:90e] Baillie, C. F., Johnston, D. A., and Williams, R. D. "Computational aspects of simulating dynamically triangulated random surfaces," *Computer Physics Communications*, 58(1/2):105–117, 1990. February/March 1990. Caltech Report C3P-808.

[Nambu:70a] Nambu, Y. "Quark model and factorization of veneziano amplitude," in R. Chand, editor, *Symmetries and Quark Models*. Gordon and Breach, 1970.

[Nielsen:70a] Nielsen, H. B. "An almost physical interpretation of the dual n-point function." Technical report, Nordita, 1970. Unpublished Nordita Report.

[Susskind:70a] Susskind, L. "Structure of hadrons implied by duality," *Phys. Rev. D.*, 1:1182–1186, 1970.

[Lovelace:68a] Lovelace, C. "A novel application of regge trajectories," *Phys. Lett.*, 28B:264–268, 1968.

[Neveu:71a] Neveu, A., and Schwarz, J. H. "Factorizable dual models of pions," *Nucl. Phys. B.*, B31:86–112, 1971.

[Scherk:74a] Scherk, J., and Schwarz, J. H. "Dual models for non-hadrons," *Nucl. Phys. B*, 81:118–144, 1974.

[Gliozzi:77a] Gliozzi, F., Scherk, J., and Olive, D. "Supersymmetry, supergravity theories and the dual spinor model," *Nucl. Phys. B.*, 122:253–290, 1977.

[Green:84a] Green, M. B., and Schwarz, J. H. "Anomaly cancellations in supersymmetric d=10 gauge theory and superstring theory," *Phys. Lett.*, 149B:117–122, 1984.

[Gross:85a] Gross, D., Harvey, J., Martinec, E., and Rohm, R. "Heterotic string theory," *Nucl. Phys. B.*, 256:253–284, 1985.

[Schwarz:85a] Schwarz, J. H. *Superstrings: The First 15 Years of Superstring Theory*, volume I and II. World Scientific, Singapore, 1985.

[Ambjorn:89a] Ambjorn, J., B., D., Frohlich, J., and Jonsson, T. "A renormalization group analysis of lattice models of two-dimensional membranes," *J. Stat. Phys.*, 55:29–85, 1989.

[Ambjorn:89b] Ambjorn, J., Durhuus, B., and Jonsson, T. "Kinematical and numerical study of the crumpling transition in crystalline surfaces," *Nucl. Phys. B*, 316:526–558, 1989.

[Baig:89a] Baig, M., Espriu, D., and Wheater, J. F. "Phase transition in random surfaces," *Nucl. Phys. B*, 314:587, 1989.

[Polyakov:81a] Polyakov, A. M. "Quantum geometry of bosonic strings," *Phys. Lett.*, 103B:207–210, 1981.

[Ambjorn:85a] Ambjorn, J., Durhuus, B., and Frohlich, J. "Diseases of triangulated random surface models and possible cures," *Nucl. Phys. B*, 257:433–449, 1985.

[David:85a] David, F. "A model of random surfaces with non-trivial critical behavior," *Nucl. Phys. B.*, 257:543–576, 1985.

[Kazakov:85a] Kazakov, V. A., Kostov, I. K., and Migdal, A. A. "Critical properties of randomly triangulated planar random surfaces," *Phys. Lett.*, 157B:295–300, 1985.

[Billoire:86a] Billoire, A., and David, F. "A model of random surfaces with non-trivial critical behavior," *Nucl. Phys. B*, 275:548–552, 1986.

[Boulatov:86a] Boulatov, D. V., Kazakov, V. A., Kostov, I. K., and Migdal, A. A. "Analytical and numerical study of a model of dynamically triangulated random surfaces," *Nucl. Phys. B*, 275:641, 1986.

[Jurkiewicz:86a] Jurkiewicz, J., Krzywicki, A., and Petersson, B. "A numerical study of discrete euclidean polyakov surfaces," *Phys. Lett.*, 168B:273–278, 1986.

[Ambjorn:87b] Ambjorn, J., de Forcrand, P., Koukiou, F., and Petritis, D. "Monte Carlo simulations of regularized bosonic strings," *Phys. Lett.*, 197B:548–552, 1987.

[David:87a] David, F., Jurkiewicz, J., Krzywicki, A., and Petersson, B. "Critical exponents in a model of dynamically triangulated random surfaces," *Nucl. Phys. B.*, 290:218–230, 1987.

[Jurkiewicz:86b] Jurkiewicz, J., Krzywicki, A., and Petersson, B. "A grand-canonical ensemble of randomly triangulated surfaces," *Phys. Lett.*, 177B:89–92, 1986.

[Durhuus:84a] Durhuus, B., Frohlich, J., and Jonsson, T. "Critical behavior in a model of planar random surfaces," *Nucl. Phys. B.*, 240:453–480, 1984.

[Espriu:87a] Espriu, D. "Triangulated random surfaces," *Phys. Lett.*, 194B:271–276, 1987.

[Polyakov:86a] Polyakov, A. M. "Fine structure of strings," *Nucl. Phys. B*, 268:406–412, 1986.

[Kleinert:86a] Kleinert, H. "The membrane properties of condensing strings," *Phys. Lett.*, 174B:335–338, 1986.

[Ambjorn:87a] Ambjorn, J. *et al.*. "Regularized strings with extrinsic curvature," *Nucl. Phys. B*, 290:480, 1987.

[Catterall:89a] Catterall, S. M. "Extrinsic curvature in dynamically triangulated random surface models," *Phys. Lett. B*, 220(1–2):207–214, 1989.

[Williams:90b] Williams, R. D. "DIME: Distributed Irregular Mesh Environment." Technical Report C3P-861, California Institute of Technology, February 1990. Users Manual.

[Barnes:90a] Barnes, T. "Numerical studies in high temperature superconductivity: The two dimensional Heisenberg antiferromagnet." Technical Report C3P-873, University of Toronto, 1990. In preparation.

[Barnes:89c] Barnes, T., Kotchan, D., and Swanson, E. S. "Evidence for a phase transition in the zero temperature anistrophic 2D Heisenberg antiferromagnet," *Phys. Rev. B.*, 39:4357, 1989. Caltech Report C3P-653.

[Barnes:89a] Barnes, T., Cappon, K. J., Dagotto, E., Kotchan, D., and Swanson, E. S. "Critical behavior of the 2D anisotropic Heisenberg antiferromagnet: A numerical test of spin-wave theory," *Phys. Rev. B.*, 40:8945, 1989. Toronto preprint UTPT-89-01. Caltech Report C3P-722.

[Barnes:88c] Barnes, T., and Daniell, G. J. "Numerical solution of spin systems and the s = 1/2 Heisenberg antiferromagnet using guided random walks," *Phys. Rev. B.*, 37:3637, 1988.

[Barnes:89b] Barnes, T. "Numerical solution of high temperature superconductor spin systems," in C. Bottcher, M. R. Strayer, and J. B. McGrory, editors, *Nuclear and Atomic Physics at one Gigaflop*, volume 10, pages 83–106. Nuclear Science Research Conference Series, Harwood Academic, 1989. Toronto preprint UTPT-88-07. Caltech Report C3P-638.

[Kochanek:88a] Kochanek, C. S., and Apostolakis, J. "The two screen gravitational lens," *Mon. Not. R. astr. Soc.*, 235:1073–1109, 1988. Caltech Report C3P-644.

[Apostolakis:88d] Apostolakis, J., and Kochanek, C. S. "Statistical gravitational lensing on the Mark III hypercube," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 963–970. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-581.

[Fox:82a] Fox, G. C. "Matrix operations on the homogeneous machine." Technical Report C3P-305, California Institute of Technology, July 1982.

[Ipsen:87b] Ipsen, I. C. F., and Jessup, E. R. "Two methods for solving the symmetric tridiagonal Eigenvalue problem on the hypercube," in

M. T. Heath, editor, *Hypercube Multiprocessors*, pages 627–638. SIAM, Philadelphia, 1987.

[Johnsson:87b] Johnsson, S. L., Saad, Y., and Schultz, M. H. "Alternating direct methods on multiprocessors," *SIAM J. Sci. Stat. Comput.*, 8:686, 1987.

[Johnsson:89a] Johnsson, S. L., and Ho, C. "Multiplication of arbitrarily shaped matrices on Boolean cubes using the full communications bandwidth." Technical Report Yale Research Report YALEU/DCS/TR-721, Yale University, July 1989.

[Saad:85a] Saad, Y., and Schultz, M. H. "Parallel direct methods for solving banded linear systems." Technical Report Yale Research Report YALEU/DCS/RR-387, Yale University, 1985.

[Geist:86a] Geist, G. A., and Heath, M. T. "Matrix factorization on a hypercube multiprocessor," in M. T. Heath, editor, *Hypercube Multiprocessors*, pages 161–180. SIAM, Philadelphia, 1986.

[Geist:89a] Geist, G. A. "Reduction of a general matrix to tridiagonal form using a hypercube multiprocessor," in J. L. Gustafson, editor, *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, page 665, P. O. Box 428, Los Altos, California 94023, March 1989. Golden Gate Enterprises.

[Romine:87a] Romine, C. H. "The parallel solution of triangular systems on a hypercube," in M. T. Heath, editor, *Hypercube Multiprocessors*. SIAM, Philadelphia, 1987.

[Romine:90a] Romine, C. H., and Sigmon, K. "Reducing inner product computation in the one-sided Jacobi algorithm," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference, Volume I*, pages 301–310, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina.

[Fox:85b] Fox, G., Hey, A. J. G., and Otto, S. "Matrix algorithms on the hypercube I: Matrix multiplication," *Parallel Computing*, 4:17, 1987. Caltech Report C3P-206.

[Hipes:89b] Hipes, P. G. "Matrix multiplication on the JPL/Caltech Mark IIIfp Hypercube — preliminary draft." Technical Report C3P-746, California Institute of Technology, March 1989. Submitted to Computer Physics Communications.

[Chu:87a] Chu, E., and George, A. "Gaussian elimination with partial pivoting and load balancing on a microprocessor," *Parallel Computing*, 5:65, 1987.

[Li:87a] Li, G., and Coleman, T. F. "A parallel triangular solver for a hypercube multiprocessor," in M. T. Heath, editor, *Hypercube Multiprocessors*. SIAM, Philadelphia, 1987.

[Moler:86a] Moler, C. "Matrix computation on distributed memory multiprocessors," in M. T. Heath, editor, *Hypercube Multiprocessors*, pages 181–195. SIAM, Philadelphia, 1986.

[Velde:87a] Van de Velde, E. F., and Keller, H. B. "The design of a parallel multigrid algorithm," in L. Kartashev and S. Kartashev, editors, *Proceedings of the Second International Conference on Supercomputing at Santa Clara*, pages 76–83, St. Petersburg, Florida, May 1987. International Supercomputing Institute, Inc. Caltech Report C3P-406.

[Aldcroft:88a] Aldcroft, T., Cisneros, A., Fox, G. C., Furmanski, W., and Walker, D. W. "LU decomposition of banded matrices and the solution of linear systems on hypercubes," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1635–1655. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-348b.

[Hipes:88c] Hipes, P. G. "A Gauss-Jordan decomposition and system solver for distributed memory multicomputers." Technical Report C3P-652, California Institute of Technology, August 1988.

[Gerasoulis:88a] Gerasoulis, A., Missirlis, n., Nelken, I., and Peskin, R. "Implementing Gauss Jordan on a hypercube multicomputer," in G. C. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1569–1576. ACM Press, 11 West 42nd Street, New York, NY 10036, 1988.

[Hipes:89d] Hipes, P. G. "Comparison of LU and Gauss-Jordan system solvers for distributed memory multicomputers." Technical Report

C3P-652c, California Institute of Technology, September 1989. Submitted to Concurrency: Practice and Experience.

[Hipes:88a] Hipes, P. G., and Kuppermann, A. "Gauss-Jordan inversion with pivoting on the Caltech Mark II hypercube," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1621–1634. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-578.

[Hipes:87a] Hipes, P., and Kuppermann, A. "Lifetime analysis of high energy resonances in three-dimensional reactive scattering," *Chem P Lett*, 133(1):1–7, 1987. Caltech Report C3P-382.

[Kuppermann:86a] Kuppermann, A., and Hipes, P. G. "Three-dimensional quantum mechanical reactive scattering using symmetrized hyperspherical coordinates," *J. Chem. Phys.*, 84(10):5962–5964, 1986. Caltech Report C3P-343.

[Furmanski:88b] Furmanski, W., Fox, G. C., and Walker, D. "Optimal matrix algorithms on homogeneous hypercubes," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1656–1673. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-386b.

[Angus:90a] Angus, I. G., Fox, G. C., Kim, J. S., and Walker, D. W. *Solving Problems on Concurrent Processors: Software for Concurrent Processors*, volume 2. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1990.

[Fox:88h] Fox, G. C., and Furmanski, W. "Optimal communication algorithms for regular decompositions on the hypercube," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, pages 648–713. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-314b.

[Velde:89b] Van de Velde, E. F. "Multicomputer matrix computations: Theory and practice," in J. L. Gustafson, editor, *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, pages 1303–1308, P. O. Box 428, Los Altos, California 94023, March 1989. Golden Gate Enterprises. CRPC-TR89002 Technical Report. Caltech Report C3P-766.

[Velde:88d] Van de Velde, E. "Data redistribution and concurrency." Technical Report C3P-635, California Institute of Technology, May 1988.

[Lorenz:89a] Lorenz, J., and Van de Velde, E. F. "Concurrent computations of invariant manifolds," in J. L. Gustafson, editor, *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, pages 1315–1320, P. O. Box 428, Los Altos, California 94023, March 1989. Golden Gate Enterprises. Caltech Report C3P-759.

[Velde:90b] Van de Velde, E. F., and Lorenz, J. "Applications of adaptive data distributions," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference, Volume I*, pages 249–253, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina. Caltech Report C3P-902.

[Schatz:75a] Schatz, G. C., and Kuppermann, A. "Quantum mechanical reactive scattering: An accurate three-dimensional calculation," *J. Chem. Phys.*, 62:2502, 1975.

[Schatz:76a] Schatz, G. C., and Kuppermann, A. "Quantum mechanical reactive scattering for three-dimensional atom plus diatom systems: I. theory," *J. Chem. Phys.*, 65:4642, 1976.

[Schatz:76b] Schatz, G. C., and Kuppermann, A. "Quantum mechanical reactive scattering for three-dimensional atom plus diatom systems: II. accurate cross sections for $H + H_2$," *J. Chem. Phys.*, 65:4668, 1976.

[Kuppermann:75a] Kuppermann, A. "A useful mapping of the triatomic potential energy surface," *Chem. Phys. Letters*, 32:374, 1975.

[Ling:75a] Ling, R. T., and Kuppermann, A. "Electronic and atomic collisions," in J. S. Rusley and R. Gabelle, editors, *9th International Conference on the Physics of Electronic and Atomic Collisions*, volume 1, pages 353–354. University of Washington Press, July 1975. Abstracts of papers presented in Seattle, Washington, July 24–30.

[Cuccaro:89a] Cuccaro, S. A., G., H. P., and Kuppermann, A. "Hyperspherical coordinate reactive scattering using variational surface functions," *Chemical Physics Letters*, 154(2):155–164, January 1989. Caltech Report C3P-720.

[Cuccaro:89b] Cuccaro, S. A., Hipes, P. G., and Kuppermann, A. "Symmetry analysis of accurate $H + H_2$ resonances," *Chemical Physics Letters*, 157(5):440–446, May 1989. Caltech Report C3P-821.

[Hipes:88b] Hipes, P., Mattson, T., Wu, M., and Kuppermann, A. "Chemical reaction dynamics: Integration of coupled sets of ordinary differential equations on the Caltech hypercube," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1051–1061. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-570.

[Johnson:73a] Johnson, B. R. "The multi-channel log-derivative method for scattering calculations," *Journal of Computational Physics*, 13:445, 1973.

[Johnson:77a] Johnson, B. R. "New numerical methods applied to solving the one-dimensional Eigenvalue problem," *J. Chem. Phys.*, 67:4086, 1977.

[Johnson:79a] Johnson, B. R. "The log derivative and renoramlized Numerov algorithms." Technical Report LBL 9501, Lawrence Berkeley Laboratory, 1979. Presented at the NNCC Workshop.

[Manolopoulos:86a] Manolopoulos, D. E. "An improved log derivative method for inelastic scatterieng," *Journal of Chemical Physics*, 85:6425, 1986.

[Liu:73a] Liu, B. "Ab initio potential energy surface for linear h_3," *Chemical Physics*, 58:1925, 1973.

[Siegbahn:78a] Siegbahn, P., and Liu, B. "An accurate three-dimensional potential energy surface for H_3," *Journal of Chemical Physics*, 68:2457, 1978.

[Truhlar:78a] Truhlar, D. G., and Horowitz, C. J. "Functional representation of Liu and Siegbahn's accurate ab initio potential energy calculations for $H + H_2$," *Journal of Chemical Physics*, 68:2466, 1978.

[Truhlar:79a] Truhlar, D. G., and Horowitz, C. J. "Erratum," *Journal of Chemical Physics*, 71:1514(E), 1979.

[Delves:59a] Delves, L. M. "Tertiary and general-order collisions," *Nuclear Physics*, 9:391, 1959.

[Delves:62a] Delves, L. M. "Three-particle photo-disintegration of the Triton," *Nuclear Physics*, 29:268, 1962.

[Lepetit:90a] Lepetit, B., Peng, Z., and Kuppermann, A. "Calculation of bound rovibrational states on the first electronically excited state of the H_3 system," *Chem. Phys. Lett.*, 166:572, 1990.

[Lepetit:90b] Lepetit, B., and Kuppermann, A. "Numerical study of the geometric phase in the $H + H_2$ reaction," *Chem. Phys. Lett.*, 166:581, 1990.

[Hood:86a] Hood, D., and Kuppermann, A. "Hyperspherical coordinate formulation of the electron — hydrogen atom scattering problems," in D. C. Clary, editor, *Theory of Chemical Reaction Dynamics*, pages 193–214. D. Reidel, Boston, 1986. Chemistry formalism related to work in C3P-94b. Caltech Report C3P-189.

[Fox:85h] Fox, G. C., Lyzenga, G., Rogstad, D., and Otto, S. "The Caltech Concurrent Computation Program — project description," in *ASME Conference on International Computers in Engineering*. ASME, August 1985. Caltech Report C3P-157.

[Fox:85d] Fox, G. C. "Annual report 1983–1984 and recent documentation." Caltech/JPL Concurrent Computation Project, Collection of Reports C3P-166, California Institute of Technology, 1985. Volume 1 - Tutorial and System Documentation; Volume 2 - Applications.

[Gilbert:58a] Gilbert, E. N. "Gray codes and paths on the N-Cube," *Bell System Technical Journal*, 37:815, May 1958.

[Fox:84g] Fox, G. C. "Eigenvalues of symmetric tridiagonal matrices." Technical Report C3P-095, California Institute of Technology, July 1984.

[Ipsen:87a] Ipsen, I. C. F., and Jessup, E. R. "Solving the symmetric tridiagonal Eigenvalue problem on the hypercube." Technical Report YALEU/DCS/RB-548, Yale University, 1987. Yale Internal Report.

[Ipsen:87c] Ipsen, I. C. F., and Jessup, E. R. "Solving the symmetric tridiagonal Eigenvalue problem on the hypercube," in *Proceedings of the Second Conference on Hypercube Multiprocessors*, 1987. Held in Knoxville, Tennessee.

[Smith:76a] Smith, B. T. "Matrix eigensystem routine — EISPACK guide, second edition," in *Lecture Notes in Computer Science, Vol. 6*. Springer-Verlag, New York, 1976.

[Fox:84h] Fox, G. C. "Householder's tridiagonalization technique." Technical Report C3P-098, California Institute of Technology, August 1984.

[Patterson:86a] Patterson, J. "Householder transformation, decomposition, results, some observations." Technical Report C3P-297.5, California Institute of Technology, 1986. Support material for Second Hypercube class.

[Wilkinson:71a] Wilkinson, J. H., and Reinsch, C. "Linear algebra," in *Handbook for Automatic Computation, Volume II*, pages 227–240. Springer-Verlag, New York, 1971.

[Pfeiffer:88a] Pfeiffer, W., Alagar, A., Kamrath, A., Leary, R., and Rogers, J. "Benchmarking and optimization of scientific codes on the CRAY X-MP, CRAY-2, and SCS-40 vector computers." Technical Report C3P-699, California Institute of Technology and San Diego Supercomputer Center, November 1988. Submitted for publication in The Journal of Supercomputing.

[Messina:90a] Messina, P., Baillie, C. F., Felten, E. W., Hipes, P. G., Walker, D. W., Williams, R. D., Pfeiffer, W., Alagar, A., Kamrath, A., Leary, R. H., and Rogers, J. "Benchmarking advanced architecture computers," *Concurrency: Practice and Experience*, 2(3):195–256, 1990. Caltech Report C3P-712b.

[Calalo:89b] Calalo, R., Cwik, T., Ferraro, R. D., Imbriale, W. A., Jacobi, N., Liewer, P. C., Lockhart, T. G., Lyzenga, G. A., Mulligan, S., Parker, J. W., and Patterson, J. E. "Hypercube matrix computation task—research in parallel computational electromagnetics." Technical report, California Institute of Technology/Jet Propulsion Laboratory, 1989. Report for 1988–1989.

[Bayliss:80a] Bayliss, A., and Turkel, E. "Radiation boundary conditions for wave-like equations," *Commun. Pure and Appl. Math*, 33:707–725, 1980.

[Peterson:85c] Peterson, A. F., and Mittra, R. "Method of conjugate gradients for the numerical solution of large-body electromagnetic scattering problems," *J. Opt. Soc. Am.*, 2A:971–977, 1985.

[Peterson:86a] Peterson, A. F., and Mittra, R. "Convergence of the conjugate gradient method when applied to matrix equation representing electromagnetic scattering problems," *IEEE Trans. Antennas and Propagation*, AP-34:1447–1454, 1986.

[Nour-Omid:87b] Nour-Omid, B., Raefsky, A., and Lyzenga, G. "Solving finite element equations on concurrent computers," in *Proceedings of the Symposium on Parallel Computations and Their Impact on Mechanics.* ASME, December 1987. Caltech Report C3P-463.

[Jenkins:83a] Jenkins, J. T., and Savage, S. B. "A theory for the rapid flow of identical, smooth, nearly inelastic, spherical particles," *Journal of Fluid Mechanics*, 130:187–202, 1983.

[Walton:83a] Walton, O. R. "Particle dynamics calculations of shear flow," in J. T. Jenkins and M. Satake, editors, *Mechanics of Granular Materials: New Models and Constitutive Relations*, pages 327–338. Elsevier, Amsterdam, 1983.

[Bagnold:41a] Bagnold, R. A. *The Physics of Blown Sand and Desert Dunes.* Methuen, London, 1941.

[Werner:88a] Werner, B. T., and Haff, P. K. "Dynamical simulations of granular materials using the Caltech hypercube," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1313–1318. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-612.

[Werner:88b] Werner, B. T., and Haff, P. K. "The impact process in Eolian saltation: Two-dimensional simulations," *Sedimentology*, 35:189–196, 1988.

[Werner:90a] Werner, B. T. "A steady-state model of wind-blown sand transport," *J. Geology*, 98:1–17, 1990. Caltech Report C3P-971.

[Werner:90b] Werner, B. T. "Computer simulation of sand surface self-organization in wind-blown sand transport," *Sedimentology*, 1990. Submitted for publication. Caltech Report C3P-972.

[Ahnert:87a] Ahnert, F. "Approaches to dynamic equilibrium in theoretical simulations of slope development," *Earth Surface Processes and Landforms*, 12:3–15, 1987.

[Liewer:89c] Liewer, P. C., and Decyk, V. K. "A general concurrent algorithm for plasma particle-in-cell simulation codes," *Journal of Computational Physics*, 85(2):302–322, 1989. Caltech Report C3P-649b.

[Decyk:88a] Decyk, V. K. *Supercomputer*, 27:33, 1988.

[Liewer:90a] Liewer, P. C., Leaver, E. F., Decyk, V. K., and Dawson, J. M. "Dynamic load balancing in a concurrent plasma PIC code on the JPL/Caltech Mark III Hypercube," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference, Volume II*, pages 939–942, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina. Caltech Report C3P-894.

[Ferraro:90b] Ferraro, R. D., Liewer, P. C., and Decyk, V. K. "A 2D electrosttic PIC code for the Mark III hypercube," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference, Volume I*, pages 440–445, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina. Caltech Report C3P-905.

[Athas:88a] Athas, W. C., and Seitz, C. L. "Multicomputers: Message-passing concurrent computers," *IEEE Computer*, pages 9–24, August 1988.

[Skjellum:90a] Skjellum, A., Leung, A. P., and Morari, M. "Zipcode: A portable multicomputer communication library atop the reactive kernel," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference, Volume II*, pages 767–776, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina. Caltech Report C3P-870.

[Skjellum:90c] Skjellum, A. *Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Ordinary Differential-Algebraic*

Process Systems in Chemical Engineering. PhD thesis, California Institute of Technology, July 1990. Caltech Report C3P-940.

[Brenan:89a] Brenan, K. E., Campbell, S. L., and Petzold, L. R. *Numerical Solution of Initial-Value Problemss in Differential-Algebraic Equations.* North Holland, Elsevier, 1989.

[Velde:90a] Van de Velde, E. "Experiments with multicomputer LU decomposition," *Concurrency: Practice and Experience*, 2(1):1–26, 1990. Caltech Report C3P-725.

[Golub:89a] Golub, G. H., and van Loan, C. F. *Matrix Computations.* Johns Hopkins University Press, Baltimore, Maryland, 1989. 2nd Edition.

[Velde:88a] Van de Velde, E. F. "A concurrent solver for sparse unstructured systems." Technical Report C3P-604, California Institute of Technology, March 1988.

[Duff:86a] Duff, I. S., Erisman, A. M., and Reid, J. K. *Direct Methods for Sparse Matrices.* Oxford University Press, 1986.

[Alaghband:89a] Alaghband, G. "Parallel pivoting combined with parallel reduction and fill-in control," *Parallel Computing*, 11:201–221, 1989.

[Skjellum:89a] Skjellum, A., Morari, M., Mattisson, S., and Peterson, L. "Concurrent DASSL: structure, application and performance," in J. L. Gustafson, editor, *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, pages 1321–1328, P. O. Box 428, Los Altos, California 94023, March 1989. Golden Gate Enterprises. Caltech Report C3P-733.

[Petzold:83a] Petzold, L. R., "DASSL: Differential algebraic system solver." Sandia National Laboratories, Livermore, California, Category #D2A2, 1983.

[Hoare:78a] Hoare, C. A. R. "Communicating sequential processes," *Communications of the ACM*, 21(8):666–677, August 1978.

[Skjellum:90d] Skjellum, A., and Leung, A. P. "LU factorization of sparse, unsymmetric Jacobian matrices on multicomputer: Experience, strategies, performance," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Comput[illegible]g Conference, Volume I*, pages 328–337,

10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina. Caltech Report C3P-839b.

[Ashby:90a] 1990. Private Communication on *Iterative DASSL.*

[Brown:91a] Brown, P. N., and Hindmarsh, A. C. "Reduced storage matrix methods in stiff ODE systems," *J. Appl. Math. and Comp.*, 1991. To be published.

[Skjellum:88a] Skjellum, A., Morari, M., and Mattisson, S. "Waveform relaxation for concurrent dynamic simulation of distillation columns," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1062–1071. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-588.

[Kuru:81a] Kuru, S. *Dynamic Simulation with an Equation Based Flowsheeting System.* PhD thesis, Carnegie Mellon University, 1981. Chemical Engineering Department.

[Westerberg:79a] Westerberg, A. W., Hutchison, H. P., Motard, R. L., and Winter, P. *Process Flowsheeting.* Cambridge University Press, 1979.

[Cook:80a] Cook, W. J. "A modular dynamic simulator for distillation systems," Master's thesis, Case Western Reserve University, 1980. Chemical Engineering.

[Andersen:88a] Andersen, H. W., and Laroche, L. F. Private Communications on *Chemsim*, 1988–1990.

[Duff:77a] Duff, I. S. "MA28 — a set of Fortran subroutines for sparse unsymmetric linear equations." Technical Report Technical Report R8730, AERE, HMSO, 1977. London.

[Lorenz:89b] Lorenz, J., and Van de Velde, E. F. "Adaptive data distributions for concurrent continuation." Technical Report CRPC-TR89013, Center for Research on Parallel Computation, 1989.

[Stone:87a] Stone, H. S. *High-Performance Computer Architecture.* Addison-Wesley, 1987.

[Hackbusch:85a] Hackbusch, W. "Multi-grid methods and applications," in *Springer Series in Computational Mathematics.* Springer-Verlag, Berlin, 1985.

[Hackbusch:82a] Hackbusch, W., and Trottenberg, U., editors. *Multigrid Methods.* Springer-Verlag, New York, 1982.

[Velde:87b] Van de Velde, E. F., and Keller, H. B. "The parallel solution of nonlinear elliptic equations," in A. K. Noor, editor, *Parallel Computations and Their Impact on Mechanics*, pages 127–153. ASME, 1987. Caltech Report C3P-447.

[Bolstadt:86a] Bolstadt, J. H., and Keller, H. "A multigrid continuation method for elliptic problems with folds," *SIAM Journal on Scientific and Statistical Computing*, 7:1081–1104, 1986.

[Chan:82a] Chan, T. F. C., and Keller, H. B. "Arc-length continuation and multi-grid techniques for nonlinear elliptic Eigenvalue problems," *SIAM Journal on Scientific and Statistical Computing*, 3:173–193, June 1982.

[Dinar:85a] Dinar, N., and Keller, H. B. "Computations of Taylor vortex flows using multigrid continuation methods." Technical report, California Institute of Technology, October 1985.

[Blackman:86a] Blackman, S. S. *Multiple-Target Tracking with Radar Applications.* Artech House, Dedham, MA, 1986.

[Burgeios:71a] Burgeios, F., and Lassalle, J. C. "An extension of munkres algorithm for the assignment problem to rectangular matrices," *Comm. of the ACM*, 14:802, 1971.

[Kuhn:55a] Kuhn, H. W. "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, 2:83, 1955.

[Borsellino:61a] Borsellino, A., and Gamba, A. "An outline of a mathematical theory of PAPA," *Nuovo Cimento Suppl. 2*, 20:221–231, 1961.

[Broomhead:88a] Broomhead, D. S., and Lowe, D. "Multivariable functional interpolation and adaptive networks," *Complex Systems*, 2:321–355, 1988.

[Gorman:88a] Gorman, R. P., and Seinowski, T. J. "Analysis of hidden units in a layered network trained to classify sonar targets," *Neural Networks*, 1:75–89, 1988.

[Rumelhart:86b] Rumelhart, D. E., and McClelland, J. L. "Explorations in the microstructure of cognition," in *Parallel Distributed Processing: Vol 1: Foundations*. MIT Press, 1986.

[Lapedes:87a] Lapedes, A., and Farber, R. "Nonlinear signal processing using neural networks: Prediction and system modeling." Technical Report LA-UR-87-1662, Los Alamos National Laboratory, 1987. Los Alamos Preprint.

[Gill:81a] Gill, P. E., Murray, W., and Wright, M. H. *Practical Optimization.* Academic Press, 1981.

[Williams:87b] Williams, R. D. "Finite elements for 2D elliptic equations with moving nodes." Technical Report C3P-423, California Institute of Technology, April 1987.

[Battiti:89a] Battiti, R. "Accelerated back-propagation learning: Two optimization methods," *Complex Systems*, 3(4):331–342, 1989. Caltech Report C3P-714.

[Shanno:78a] Shanno, D. F. "Conjugate gradient methods with inexact searches," *Mathematics of Operations Research*, 3(3):244–256, 1978.

[Bowyer:81a] Bowyer, A. "Computing Direchlet tesselations," *Comp. J.*, 24:162, 1981.

[Young:71a] Young, D. M. *Iterataive Solution of Large Linear Systems.* Academic Press, New York, 1971.

[Williams:89c] Williams, R. D., and Felten, E. W. "Distributed processing of an irregular tetrahedral mesh." Technical Report C3P-793, California Institute of Technology, May 1989.

[Rivara:84a] Rivara, M. "Design and data structure of fully adaptive multigrid, finite-element software," *ACM Trans. in Math. Software*, 10:242, 1984.

[Williams:89a] Williams, R., and Glowinski, R. "Distributed irregular finite elements," in C. Taylor, editor, *Numerical Methods in Laminar and*

Turbulent Flow, volume 6, pages 3–13, Swansea, United Kingdom, 1989. Pineridge Press. Caltech Report C3P-715.

[Bristeau:87a] Bristeau, M. O., Glowinski, R., and Periaux, J. "Numerical methods for the Navier-Stokes equations: Applications to the simulation of compressible and incompressible viscous flows," *Comp. Phys. Rep.*, 6:73, 1987.

[Glowinski:84a] Glowinski, R. *Numerical Methods for Nonlinear Variational Problems.* Springer-Verlag, New York, 1984.

[Schreiber:83a] Schreiber, R., and Keller, H. B. "Driven cavity problems by efficient numerical techniques," *J. Comp. Phys.*, 49:310, 1983.

[Williams:89b] Williams, R. D. "Supersonic flow in parallel with an unstructured mesh," *Concurrency: Practice and Experience*, 1(1):51–62, 1989. (See manual for this code in Caltech Report, C^3P-861 (1990)). Caltech Report C3P-636b.

[Fox:88mm] Fox, G. C. "A review of automatic load balancing and decomposition methods for the hypercube," in M. Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, pages 63–76. Springer-Verlag, 1988. Caltech Report C3P-385.

[Hajek:88a] Hajek, B. "Cooling schedules for optimal annealing," *Mathematics of Operational Research*, 13:311–329, 1988.

[Otten:89a] Otten, R. H. J. M., and van Ginneken, L. P. P. P. *The Annealing Algorithm.* Kluwer Academic, Boston, Massachusetts, 1989.

[Baiardi:89a] Baiardi, F., and Orlando, S. *Strategies for a Massively Parallel Implementation of Simulated Annealing*, volume 366 of *Lecture Notes in Computer Science*, pages 273–277. Springer-Verlag, 1989.

[Barajas:87a] Barajas, F., and Williams, R. "Optimization with a distributed-memory parallel processor." Technical Report C3P-465, California Institute of Technology, September 1987.

[Braschi:90a] Braschi, B., Ferreira, A. G., and Zerovnik, J. "On the behavior of parallel simulated annealing," in D. J. Evans, G. R. Joubert, and F. J. Peters, editors, *Supercomputing 90*, pages 17–26. Elsevier, Amsterdam, 1990.

[Williams:86b] Williams, R. D. "Minimization by simulated annealing: Is detailed balance necessary?." Technical Report C3P-354, California Institute of Technology, September 1986. CALT-68-1407.

[Coddington:90a] Coddington, P. D., and Baillie, C. F. "Cluster algorithms for spin models on MIMD parallel computers," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference, Volume I*, pages 384–387, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina. Caltech Report C3P-862.

[Barnes:82a] Barnes, E. R. "An algorithm for partitioning the nodes of a graph," *SIAM Journal for Algorithms and Discrete Methods*, 3:541–550, 1982.

[Boppana:87a] Boppana, R. B. "Eigenvalues and graph bisection: An average case analysis," in *28th Annual Symposium on the Foundations of Computer Science*, pages 67–75, New York, 1987. Institute of Electrical and Electronics Engineers.

[Pothen:89a] Pothen, A., Simon, H. D., and Liu, K. P. "Partitioning sparse matrices with eigenvectors of graphs." Technical Report RNR-89-009, NASA Ames Research Center, July 1989.

[Fox:88tt] Fox, G. C., and Furmanski, W. "The physical structure of concurrent problems and concurrent computers," *Phil. Trans. R. Soc. Lond. A*, 326:411–444, 1988. Caltech Report C3P-493.

[Hopfield:85b] Hopfield, J., and Tank, D. ""Neural" computation of decisions in optimization problems," *Biol. Cybern.*, 52:141–152, 1985.

[Fiedler:75a] Fiedler, M. "Algebraic connectivity of graphs," *Czechoslovak Mathematics Journal*, 23(2/3):298–307, 1975.

[Fiedler:75b] Fiedler, M. "A property of eigenvectors of non-negative symmetric matrics and its application to graph theory," *Czechoslovak Mathematics Journal*, 25:619–627, 1975.

[Golub:83a] Golub, G. H., and van Loan, C. F. *Matrix Computations*. Johns Hopkins University Press, Baltimore, Maryland, 1983.

[Parlett:80a] Parlett, B. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[Martin:89a] Martin, O., Otto, S. W., and Felten, E. W. "Large-step Markov chains for the traveling salesman problem." Technical Report C3P-836, California Institute of Technology, August 1989. Submitted to Operations Research.

[Lin:65a] Lin, S. "Computer solutions of the traveling salesman problem," *The Bell System Technical Journal*, 44:2245, December 1965.

[Lin:73a] Lin, S., and Kernighan, B. W. "An effective heuristic algorithm for the traveling salseman problem," *Operations Res.*, 21:498, 1973.

[Johnson:91a] Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C., "Optimization by simulated annealing: An experimental evaluation, Part III (the traveling salesman problem)," 1991. In preparation.

[Johnson:90b] Johnson, D. S. "Local optimization and the traveling salesman problem," in *Proceedings of the 17th Colloquium on Automata, Languages, and Programming.* Springer-Verlag, 1990.

[Cook:90b] Cook, W., Chvatal, V., and Applegate, D., 1990. TSP Workshop, Rice University, April 22–24.

[TSP:90a] 1990. TSP Workshop, Rice University, April 22–24.

[Williams:90c] Williams, R. D., Rasnow, B., and Assad, C. "Hypercube simulation of electric fish potentials," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference, Volume I*, pages 470–477, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina. Caltech Report C3P-869.

[Bullock:86a] Bullock, T. H., and Heiligenberg, W., editors. *Electroreception.* John Wiley and Sons, Ltd., New York, 1986.

[Lissman:58a] Lissman, H. W. "On the function and evolution of electric organs in fish," *J. Exp. Biol.*, 35:156–160, 1958.

[Bacher:83a] Bacher, M. "A new method for the simulation of electric fields, generated by electric fish, and their distortions by objects," *Biol. Cybern.*, 47:51–54, 1983.

[Heiligenberg:75a] Heiligenberg, W. "Theoretical and experimental approaches to spatial aspects of electrolocation," *J. Comp. Physiol.*, 103:66–72, 1975.

[Brebbia:83a] Brebbia, C. A., editor. *Boundary Elements.* Springer-Verlag, Berlin, 1983.

[Cruse:75a] Cruse, T. A., and Rizzo, F. J., editors. *Boundary Integral Equation Method: Computational Applications in Applied Mechanics.* Applied Mechanics Division (ASME, Vol. 11), American Society of Mechanical Engineers, June 1975. Rensselaer Polytechnic Institute, Troy, New York, June 23–25.

[Jameson:86a] Jameson, A., and Baker, T. J. *Euler Calculations for a Complete Aircraft*, volume 264 of *Lecture Notes in Physics*, pages 334–344. Springer-Verlag, 1986. 10th International Conference on Numerical Methods in Fluid Mechanics, ed. F. G. Zhang and Y. L. Zhu.

[Jameson:86b] Jameson, A., Baker, T. J., and Weatherill, N. P. "Calculation of inviscid transonic flow over a complete aircraft." Technical Report AIAA Paper 86-0103, American Institute of Aeronautics and Astronautics, 1986.

[Dannenhoffer:89a] Dannenhoffer, D. J., and Davis, R. L. "Adaptive grid computations for complex flows," in *Proceedings of the Fourth International Conference on Supercomputing at Santa Clara, Volume II*, page 206, St. Petersburg, Florida, 1989. International Supercomputing Institute.

[Holmes:86a] Holmes, D. G., and Lamson, S. H. "Adaptive triangular meshes for compressible flow solutions," in J. Hauser and C. Taylor, editors. *Proceedings of the International Conference on Numerical Grid Generation, Landshut*, pages 413–423, Swansea, United Kingdom, 1986. Pineridge Press.

[Lohner:84a] Löhner, R., Morgan, K., and Zienkowicz, O. C. "The solution of non-linear hyperbolic equation systems by the finite element method," *International Journal of Numerical Methodology in Engineering*, 4:1043–1053, 1984.

[Lohner:85a] Löhner, R., Morgan, K., and Zienkowicz, O. C. "An adaptive finite element procedure for compressible high speed flows," *Comp. Meth. in Appl. Mech. and Eng.*, 51:441, 1985.

[Lohner:86a] Lohner, R., Morgan, K., Peraire, J., Zienkowicz, O. C., and Kong, L. "Finite element methods for compressible flow," in K. W. Morton and M. J. Baines, editors, *Numerical Methods for Fluid Mechanics, II*, pages 28–53. Clarendon Press, Oxford, 1986.

[Jameson:87a] Jameson, A., and Baker, T. J. "Improvements to the Aircraft Euler Method." Technical Report AIAA Paper 87-0452, American Institute of Aeronautics and Astronautics, 1987.

[Jameson:87b] Jameson, A. "Successes and challenges in computational aerodynamics." Technical Report AIAA Paper 87-1184, American Institute of Aeronautics and Astronautics, 1987.

[Mavriplis:88a] Mavriplis, D. J. "Accurate multigrid solution of the Euler equations on unstructured and adaptive meshes." Technical Report NASA-CR 181679/ICASE 88-40, NASA/ICASE, 1988.

[Perez:86a] Perez, E., Periaux, J., Rosenblum, J. P., Stoufflet, B., Derviaux, A., and Lallemand, M. H. *Adaptive Full-Multigrid Finite Element Methods for Solving the Two-Dimensional Euler Equations*, volume 264 of *Lecture Notes in Physics*, pages 523–533. Springer-Verlag, 1986. 10th International Conference on Numerical Methods in Fluid Mechanics, ed. F. G. Zhang and Y. L. Zhu.

[Usab:83a] Usab, W. J., and Murman, E. M. "Embedded mesh solution of the Euler equations using a multiple grid method." Technical Report AIAA Paper 83-1946-CP, American Institute of Aeronautics and Astronautics, 1983.

[AGARD:83a] for Aerospace Research or Development, A. G. *Test Cases for Steady Inviscid Transonic or Supersonic Flows.* North Atlantic Treaty Organization, January 1983. AGARD FDP WG-07.

[Williams:90a] Williams, R. D. "Performance of a distributed unstructured-mesh code for transonic flow." Technical Report C3P-856, California Institute of Technology, January 1990. To be presented at 1990 ACM International Conference on Supercomputing, June 1990 in The Netherlands.

[Leonard:80a] Leonard, A. "Vortex methods for flow simulation," *J. Computational Physics*, 37:289, 1980.

[Appel:85a] Appel, A. W. "An efficient program for many-body simulation," *Sci. Stat. Comput.*, 6:85, 1985.

[Barnes:86a] Barnes, J., and Hut, P. "A hierarchical $O(N \log N)$ force calculation algorithm," *Nature*, 324:446, 1986.

[Greengard:87b] Greengard, L., and Rokhlin, V. "A fast algorithm for particle simulations," *J. Comput. Phys.*, 73:325, 1987.

[Fox:80a] Fox, G. C. "Jet production in high-energy Hadron-Proton collisions," *Nuclear PHysics B*, 171:38, 1980. (with experimental groups at Caltech, UCLA, Chicago Circle, Fermilab and Indiana).

[Bouard:80a] Bouard, R., and Coutanceau, M. "The early stage of development of the wake behind an impulsively started cylinger for $40 \leq \mathrm{Re} \leq 10^4$," *J. Fluid Mech.*, 101:583, 1980.

[Stauffer:78a] Stauffer, D. "Scaling theory of percolation clusters," *Phys. Rep.*, 54:1, 1978.

[Essam:80a] Essam, J. W. "Percolation theory," *Rep. Prog. Phys.*, 43:833, 1980.

[Wolff:89a] Wolff, U. "Collective Monte Carlo updating for spin systems," *Phys. Rev. Lett.*, 62:361, 1989.

[Dewar:87a] Dewar, R., and Harris, C. K. "Parallel computation of cluster properties: Application to 2-D percolation," *J. Phys. A*, 20:985, 1987.

[Knuth:68a] Knuth, D. E. "Fundamental algorithms," in *The Art of Computer Programming, Vol. 1.* Addison-Wesley, Reading, 1968.

[Press:86a] Press, W., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. *Numerical Recipes, The Art of Scientific Computing.* Cambridge: New York, 1986.

[Burkitt:89a] Burkitt, A. N., and Heermann, D. W. "Parallelization of a cluster algorithm," *Comp. Phys. Comm.*, 54:201–209, 1989.

[Cypher:89a] Cypher, R., Sanz, J. L. C., and Snyder, L. "Hypercube and shuffle-exchange algorithms for image component labeling," *J. Algorithms*, 10:140–150, 1989.

[Embrechts:89a] Embrechts, H., Roose, D., and Wambacq, P. "Component labeling on a distributed memory multiprocessor," in F. Andre and J. P. Verjus, editors, *Proc. First European Workshop on Hypercube and Distributed Computers*, pages 5–17. North-Holland, Amsterdam, 1989.

[Lim:87a] Lim, W., Agrawal, A., and Nekludova, L. "A fast parallel algorithm for labeling connected components in image arrays." Technical Report NA86-2, Thinking Machines Corporation, 1987.

[Knuth:73a] Knuth, D. E. *Sorting and Searching, The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Mass., 1973.

[Batcher:68a] Batcher, K. E. "Sorting networks and their applications," in *AFIPS Conference Proceedings 32*, page 307. AFIPS Press, Montvale, NJ, 1968.

[Shell:59a] Shell, D. L. "A high-speed sorting procedure," *Communications of the ACM*, 2:30, 1959.

[Hoare:62a] Hoare, C. A. R. "Quicksort," *Computer J.*, 5:10, October 1962.

[Callahan:88d] Callahan, D., and Kennedy, K. "Compiling programs for distributed-memory multiprocessors," in *1988 Workshop on Programming Languages and Compilers for Parallel Computing*, Cornell, August 1988.

[Chen:88b] Chen, M., Li, J., and Choo, Y. "Compiling parallel programs by optimizing performance," *Journal of Supercomputing*, 2:171–207, 1988.

[Koelbel:87a] Koelbel, C., Mehrotra, P., and Van Rosendale, J. "Semi-automatic process partitioning for parallel computation," *International Journal of Parallel Computing*, 16, 1987.

[Koelbel:90a] Koelbel, C., Mehrotra, P., and Rosendale, J. V. "Supporting shared data structures on distributed memory machines," in *Principles and Practice of Parallel Programming (PPoPP)*, March 1990. Held in Seattle, Washington.

[Rogers:89b] Rogers, A., and Pingali, K. "Process decomposition through locality of reference," in *SIGPLAN 89 Conference on Programming Language Design and Implementation*, June 1989.

[Zima:88a] Zima, H. P., Bast, H.-J., and Gerndt, M. "SUPERB: a tool for semi-automatic MIMD/SIMD parallelization," *Parallel Computing*, 6:1, 1988.

[Wolfe:89a] Wolfe, M. *Optimizing Supercompilers for Supercomputers.* MIT Press, Boston, 1989.

[Saltz:87b] Saltz, J. H., Naik, V. K., and Nicol, D. M. "Reduction of the effects of the communication delays in scientific algorithms on message passing MIMD architectures," *SIAM Journal of Sci. and Stat. Computing*, 8(1):118–134, January 1987.

[Balasundaram:90d] Balasundaram, V., Fox, G., Kennedy, K., and Kremer, U. "Estimating communication costs from data layout specifications in an interactive data partitioning tool." Technical Report C3P-886, California Institute of Technology, April 1990. Invited presentation at the Workshop on Programming Distributed Memory Machines: Language Constructs, Compilers and Run-time Environments, NASA Langley Research Center, Hampton, Virginia, May 14–16, 1990.

[Balasundaram:90a] Balasundaram, V., Fox, G., Kennedy, K., and Kremer, U. "An interactive environment for data parititioning and distribution," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference, Volume II*, pages 1160–1170, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina; CRPC-TR90047. Caltech Report C3P-883.

[Balasundaram:89c] Balasundaram, V., Kennedy, K., Kremer, U., and McKinley, K. "The ParaScope Editor: An interactive parallel programming tool," in *Supercomputing 89*, November 1989. Held in Reno, Nevada.

[Fox:89l] Fox, G. C., and Koller, J. G. "Code generation by a generalized neural network: General principles and elementary examples," *Journal of Parallel and Distributed Computing*, 6(2):388–410, 1989. Caltech Report C3P-650b.

[Koller:88c] Koller, J. "Neural compiler talk." Technical Report C3P-663, California Institute of Technology, August 1988. foils.

[Fox:89q] Fox, G. C., Furmanski, W., and Koller, J. "The use of neural networks in parallel software systems," *Mathematics and Computers in Simulation*, 31(6):485–495, 1989. Elsevier Science Publishers B. V. (North-Holland). Caltech Report C3P-642b.

[Felten:88h] Felten, E. W., and Otto, S. W. "Chess on a hypercube," in *Proceedings of IEEE Symposium on the Design and Application of Parallel Digital Processors*, pages 30–42. IEEE Press, April 1988. Held in Lisbon. Caltech Report C3P-579b.

[Meier:90a] Meier, D. L., Cloud, K. L., Horvath, J. C., Allan, L. D., Hammond, W. H., and Maxfield, H. A. "A general framework for complex time-driven simulations on hypercubes," in D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference, Volume I*, pages 117–121, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina. Caltech Report C3P-960.

[Ghil:90a] Ghil, M., Keppenne, C. L., and *et al.*, "Parallel processing applied to climate modeling," 1990. UCLA Atmospheric Sciences preprint.

[Whiteside:88a] Whiteside, R. A., and Leichter, J. S. "Using Linda for supercomputing on a local area network," in *Proceedings of Supercomputing '88*, 1730 Massachusetts Avenue NW, Washington, D.C. 20036-1903, 1988. IEEE Computer Society Press. held November 14–18, Orlando, Florida.

[Padua:86a] Padua, D. A., and Wolfe, M. J. "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, 29(12):1185, 1986.

[Foster:90a] Foster, I., and Taylor, S. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Clifs, New Jersey 07632, 1990.

[Felten:88a] Felten, E. W., and Otto, S. W. "Coherent parallel C," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, pages 440–450. ACM Press, 11 West

42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-527.

[Johnson:86b] Johnson, M. "Melting in two dimensions." Technical Report C3P-297.10, California Institute of Technology, 1986. Support material for Second Hypercube class.

[Halperin:78a] Halperin, B. I., and Nelson, D. R. *Phys. Rev. Lett.*, 41:121, 1978.

[Nelson:79a] Nelson, D. R., and Halperin, B. I. *Phys. Rev. B.*, 19:2457, 1979.

[Young:79a] Young, A. P. *Phys. Rev. B*, 19:2457, 1979.

[Frey:83a] Frey, P. W., editor. *Chess Skill in Man and Machine.* Springer-Verlag, New York, 1983.

[Marsland:87a] Marsland, T. A. "Computer chess methods," in Shapiro, editor, *Encyclopedia of Artificial Intelligence.* John Wiley and Sons, New York, 1987.

[Ebeling:85a] Ebeling, C. *All the Right Moves: A VLSI Architecture for Chess.* MIT Press, Cambridge, 1985.

[Welsh:85a] Welsh, D. E., and Baczynskyj, B. "Computer chess II,". 1985. Dubuque, Iowa.

[Knuth:75a] Knuth, D. E., and Moore, R. W. "An analysis of Alpha-Beta pruning," *Artificial Intelligence*, 6:293–326, 1975.

[Zobrist:70a] Zobrist, A. L. "A hashing method with applications for game playing." Technical Report 88, University of Wisconsin, 1970. Computer Sciences Department, Madison, Wisconsin.

[NCUBE:87a] Corporation, N. *NCUBE Users' Handbook.* NCUBE Corporation, Beaverton, Oregon, October 1987.

[Finkel:82a] Finkel, R. A., and Fishburn, J. P. "Parallelism in Alpha-Beta search," *Artificial Intelligence*, 19:89–106, 1982.

[Marsland:84a] Marsland, T. A., and Popowich, F. "Parallel game-tree search." Technical Report TR-85-1, University of Alberta, 1984. Department of Computing Science.

[Newborn:85a] Newborn, M. "A parallel search chess program," in *Proceedings of the ACM Annual Conference*, pages 272–277. ACM, New York, 1985.

[Schaeffer:84a] Schaeffer, J., Olafsson, M., and Marsland, T. A. "Experiments in distributed tree-search." Technical Report TR-84-4, University of Alberta, 1984. Department of Computing Science.

[Schaeffer:86a] Schaeffer, J. "A multiprocessor chess program," in *Proceedings of ACM-IEEE Fall Joint Computer Conference*, 1986.

[Felten:88b] Felten, E. W. "Generalized signals: An interrupt-based communication system for hypercubes," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, pages 563–568. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-433b.

[Morison:88a] Morison, R. "Interactive performance display and debugging using the NCUBE real-time graphics system," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, pages 760–765. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-576.

[Thompson:82a] Thompson, K. "Computer chess strength," in M. Clarke, editor, *Advances in Computer Chess*, pages 55–56. Pergamon Press, Oxford, 1982. Volume 3.

[Salmon:88a] Salmon, J., Callahan, S., Flower, J., and Kolawa, A. "MOOSE: A multi-tasking operating system for hypercubes," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, pages 391–396. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-586.

[Salmon:88c] Salmon, J., "Ray tracing," January 1988. Poster session presentation at The Third Conference on Hypercube Concurrent Computers and Applications, Pasadena. Caltech Report C3P-565.

[Fox:86h] Fox, G., Kolawa, A., and Williams, R. "The implementation of a dynamic load balancer," in M. T. Heath, editor, *Hypercube Multiprocessors*, pages 114–121. SIAM, Philadelphia, 1987. Caltech Report C3P-328.

[Felten:86a] Felten, E., Morison, R., Otto, S., Barish, K., Fätland, R., and Ho, F. "Chess on a hypercube." Technical Report C3P-383, California Institute of Technology, 1986. For publication information, see Felten:87a.

[Salmon:89a] Salmon, J., Quinn, P., and Warren, M. "Using parallel computers for very large N-body simulations: Shell formation using 180K particles." Technical Report C3P-780, California Institute of Technology, April 1989.

[Jefferson:85c] Jefferson, D. "Virtual time," *ACM Transactions on Programming Languages and Systems*, 7(3), July 1985.

[Wieland:89a] Wieland, F., Hawley, L., Feinberg, A., DiLoreto, M., Blume, L., Ruffles, J., Reiher, P., Beckman, B., Hontalas, P., Bellenot, S., and Jefferson, D. "The performance of a distributed combat simulation with the time warp operating system," *Concurrency: Practice and Experience*, 1(1):35–50, 1989. Caltech Report C3P-798.

[Reiher:90a] Reiher, P. L., and Jefferson, D. J. "Virtual time based dynamic load management in the time warp operating system," in *Proceedings of Society for Computer Simulation Multiconference on Distributed Simulation*, pages 103–111, 1990.

[Jefferson:87a] Jefferson, D., Beckman, B., Wieland, F., Blume, L., Di Loreto, M., Hontalas, P., LaRouche, P., Sturdevant, k., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. "Distributed simulation and the time warp operating system," in *Proceedings of the 11th Annual ACM Symposium on Operating System Principles*, 1987.

Index

END

DATE FILMED

12/11/91