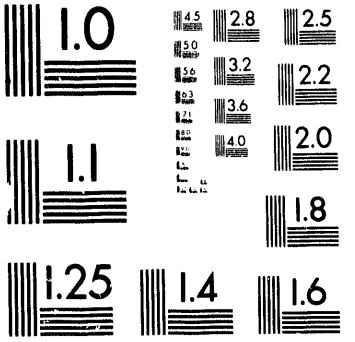

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

1 of 3

JPL Publication 93-15

System Software and Tools for High Performance Computing Environments

A Report on the Findings of the Pasadena Workshop: April 14–16, 1992

Thomas Sterling, USRA CESDIS
Paul Messina, California Institute of Technology, Jet Propulsion Laboratory
Marina Chen, Yale University
Frederica Darema, IBM T. J. Watson Research Center
Geoffrey Fox, Syracuse University
Michael Heath, National Center for Supercomputing Applications
Ken Kennedy, Rice University
Robert Knighten, Intel Supercomputer Systems Division
Reagan Moore, San Diego Supercomputer Center
Sanjay Ranka, Syracuse University
Joel Saltz, University of Maryland
Lew Tucker, Thinking Machines Corporation
Paul Woodward, University of Minnesota

April 1, 1993

Prepared for

National Aeronautics and Space Administration

Defense Advanced Research Projects Agency

Department of Energy

National Science Foundation

National Institute of Standards and Technology

National Oceanic and Atmospheric Administration

National Institutes of Health

Environmental Protection Agency

National Security Agency

by

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

Acknowledgments

The workshop on which this report is based was ably supported by many people. We would like to give particular thanks and praise to the following people:

Debby Kramer (JPL) and Terri Canzian (Caltech) provided general support before, during and after the workshop, including compiling and reproducing the working group presentations and white papers during the workshop. Kim Dunn (USRA) helped prepare and distribute materials to workshop attendees prior to the event and was involved in registration. Pat McLane (JPL) was in charge of local arrangements and on-site registration. She secured excellent facilities for this large yet highly interactive workshop. Chip Chapman and Paul Angelino (both of Caltech) were responsible for providing connections to the internet and to video projection equipment so that live demonstrations of the use of software repositories could be carried out.

We would also like to recognize the efforts of those who helped create this report. Stephen Lundstrom (PARSA) wrote extensive comments and insightful suggestions for the position papers of all the working groups. Terri Canzian formatted the many drafts of the report and typed some of the chapters and appendices. Mike MacDonald (USRA) provided careful editing of much of the material in this report. Debby Kramer oversaw the production and dissemination of several intermediate drafts as well as the final draft.

Conference Credits

The Workshop on System Software and Tools for High-Performance Computing Environments was held in Pasadena, California on April 14–16, 1992. The workshop was conceived, organized, and carried out by

Lee Holcomb (NASA) Workshop Chair

Organizing Committee

Paul Smith, Chair (NASA)

Mel Ciment (NSF)
George Cotter (NSA)
Fred Johnson (NIST)
Gary Johnson (DOE)
Carl Kukkonen (JPL)
Fred Long (NOAA)
Jacob Maizel (NIH)
Joan Novak (EPA)
William Scherlis (DARPA)

Program Committee

Paul Messina, Chair (Caltech/Jet Propulsion Laboratory)
Jack Dongarra (University of Tennessee/Oak Ridge National Laboratory)
John Dorband (NASA Goddard Space Flight Center)
Brian Ford (Numerical Algorithms Group, Ltd.)
Geoffrey Fox (Syracuse University)
Mark Furtney (Cray Research)
Mike Heath (NCSA/University of Illinois)
Ken Kennedy (Rice University)
Bob Knighten (Intel Supercomputer Systems Division)
H. T. Kung (Harvard University)
Steve Lundstrom (PARSA)
Robert Malone (Los Alamos National Laboratory)
David Mizell (Boeing Computer Services)
Reagan Moore (San Diego Supercomputer Center)
John Riganati (Supercomputing Research Center)
Joel Saltz (University of Maryland)
Thomas Sterling (USRA CESDIS)
Rick Stevens (Argonne National Laboratory)
William Tompkins (United Technologies Research Center)
Lew Tucker (Thinking Machines Corporation)
Paul Woodward (University of Minnesota)
Jerry Yan (NASA Ames Research Center)

Sponsors

The Workshop on System Software and Tools for High-Performance Computing Environments was sponsored by

National Aeronautics and Space Administration
Defense Advanced Research Projects Agency
Department of Energy
National Science Foundation
National Institute of Standards and Technology
National Oceanic and Atmospheric Administration
National Institutes of Health
Environmental Protection Agency
National Security Agency

We gratefully acknowledge their support for the workshop and the preparation of this report.

This proceedings was prepared by the California Institute of Technology, supported by funding from these sponsoring agencies. The Jet Propulsion Laboratory, California Institute of Technology oversaw the printing of this proceedings.

Abstract

The Pasadena Workshop on System Software and Tools for High Performance Computing Environments was held at the Jet Propulsion Laboratory from April 14 through April 16, 1992. The workshop was sponsored by a number of Federal agencies committed to the advancement of high performance computing (HPC) both as a means to advance their respective missions and as a national resource to enhance American productivity and competitiveness. Over a hundred experts in related fields from industry, academia, and government were invited to participate in this effort to assess the current status of software technology in support of HPC systems. The overall objectives of the workshop were to understand the requirements and current limitations of HPC software technology and to contribute to a basis for establishing new directions in research and development for software technology in HPC environments. This report includes reports written by the participants of the workshop's seven working groups. Materials presented at the workshop are reproduced in appendices. Additional chapters summarize the findings and analyze their implications for future directions in HPC software technology development.

Foreword

I am pleased to have the opportunity to introduce this report on the Pasadena Workshop on System Software and Tools for High Performance Computing (HPC) Environments. With the advent of emerging scalable parallel processing systems and the computational demands of the Grand Challenge problems, the field of HPC is at a defining moment in its evolution. This workshop was formulated to set a new course for the near term and future development of high performance computing software technology. This was accomplished by inviting over a hundred experts in related disciplines to meet and consider critical aspects of the problems and opportunities confronting this exciting field. Few occasions such as this have afforded a better chance to fundamentally influence the direction of the field of software technology for HPC.

Key areas in science and engineering must now rely on computer modeling, simulation, and analysis to enable advances to be made rapidly and at low risk. U.S. competitiveness and productivity in the global high technology markets hinge on the ability to bring large computing capabilities to bear on a wide range of industrial and scientific problems. Parallel processing is key to orders of magnitude improvement in performance, but must leverage the simultaneous advances in device technology, processor architecture, scalable parallel systems, and parallel algorithms. Massively parallel computing systems have only recently exceeded vector processing supercomputers in peak performance and memory capacity. This moves massively parallel processing from the intellectual curiosity of a few years ago to a high technology driven imperative of immediate importance.

Software technology is the critical area requiring focus for near term progress in HPC. With the computational problems defined by user needs, and the computing architectures constrained by commodity workstation processor designs, new software technology and commercial products must be developed that address both. Programming environments for facilitating the development of portable highly parallel applications are needed.

Nine Federal agencies (DARPA, DOE, EPA, NASA, NIH, NIST, NOAA, NSF, and NSA) contributing to the High Performance Computing and Communications (HPCC) program are proud to have sponsored this workshop to set new directions in this rapidly progressing field. NASA is particularly pleased to have played a lead role in the formation of this workshop. All of us hope that the findings described in this document will be an important contribution to the research scientists and engineers engaged in advancing

the capabilities of HPC. Rapid progress in these areas is essential. Harnessing diverse talents and resources in a coherent and coordinated manner to establish directions and carry out activities is key to this success. We encourage and look forward to the reactions and responses to this report, so as to continue the process of developing a community consensus.

Please, now read with interest and enjoyment the thoughts of your colleagues as they delve into the complex issues confronting this exciting field.

Paul H. Smith
Chairman, Organizing Committee

Contents

APPENDICES

Executive Summary

Introduction

This report summarizes the results of the Pasadena Workshop on System Software and Tools for High Performance Computing Environments at the Jet Propulsion Laboratory from April 14 through April 16, 1992. The workshop was sponsored by nine Federal agencies committed to the advancement of high performance computing (HPC), both as a means to advance their respective missions and as a national resource to enhance American productivity and competitiveness.

The impetus for the workshop was the recognition among the agencies that significant progress in system software and tools would be required to improve the usability of HPC systems and to achieve the sustained, scalable TeraFLOPS (10^{12} floating point operations per second) computational performance needed to support the investigation of Grand Challenge problems. The organizing committee designed the workshop to draw together participants from industry, academia and government to review current system software and tools and identify needed developments, including software priorities and mechanisms for creating that software.

Workshop Structure and Scope

System software and tools both include many subjects and interactions among disciplines. To provide a manageable structure for the workshop, seven topic areas were identified and assigned to working groups for study: Applications, Mathematical Software, Compilers and Languages, Software Tools, Operating Systems, Computing Environments, and Visualization. These diverse topic areas also ensured that key concepts and technologies would be addressed. The Applications Working Group played a unique role by identifying the requirements HPCC applications have for the software

addressed by the other working groups.

Over 120 experts from a wide spectrum of industrial, academic and government organizations contributed to the workshop and its working groups. Each working group prepared a position paper on its topic before the workshop, discussed it with the other groups during the workshop, and revised it based on the interactions. In their position papers the working groups were asked to address the following four issues:

1. Identify the important system software problems that need to be solved if the HPCC program is to be successful.

2. Categorize the software problems as (a) those that are best left to the vendors of HPC equipment, (b) those that are primarily the responsibility of the users, (c) those that could benefit from joint government agency attention, and (d) those whose solution should involve collaboration with the vendors.

3. Establish priorities for the software problems.

4. Provide a vision for the HPCC software component.

This report includes the working group position papers, summaries of those papers, and detailed analyses of issues, and implications.

A long-term objective of HPC software technology research and development is to bring massively parallel processing (MPP) system technology to a state of applicability and ease-of-use comparable to that of conventional supercomputers while attaining sustained TeraFLOPS-scale capability. The challenges involved in attaining this objective, however, have forced the community to adopt a more pragmatic, near-term objective: to provide immediate, practical means for application of MPP capability to Grand Challenge problems. It was this second objective on which the Pasadena workshop participants focused their efforts.

Three goals for system software and tools development were considered essential for achieving the near-term objective: performance, portability, and usability. Increasing computational performance to levels that render tractable new classes of engineering and scientific applications is the driving motivation of the HPC community. Portability of software among computers of different brands and numbers of processors is critical to achieving program (i.e., code) longevity and protecting investments in software development. Portability also is an important factor in scalability to large numbers of

processors. Interoperability, the ability to combine software modules developed under independent circumstances to achieve the desired functionality, contributes to portability as well as to reusability. The third goal, usability, determines the ultimate ease-of-use of the total HPC systems. While these three complementary goals are crucial to the viability of the emerging MPP systems as an enabling technology for science and engineering advancement, they often lead to conflicting design criteria that demand difficult trade-off decisions.

In keeping with the emphasis on studying the needs for the first few years of the HPCC program (i.e., the near-term objective of the HPC community), workshop participants focused primarily on a single class of machine. The target MPP was assumed to comprise upwards of a thousand conventional processors, each associated with large local memory, and interacting through rapid message-passing communication networks. This is the prevalent architecture of high-end commercial MPPs and is likely to remain so during most of the 1990s. Such distributed-memory MIMD multiprocessors are connected via local area networks to other systems such as file servers, graphical visualization workstations, and SIMD and vector processors. MPPs are invariably embedded in such heterogeneous environments.

Wide area networks will eventually integrate most such environments into a single national file system, providing the opportunity and means for shared computation. Such an environment will present challenges for resource allocation and management strategies and methods. In all cases, a balance between efficiency and ease-of-use has to be achieved that takes into account both system and programming resources. This balance will be realized by reordering the responsibilities of the programmer, compiler, runtime and operating systems. In managing concurrency, synchronization and scheduling are particularly important. Synchronization mechanisms must be achieved with low overhead costs. Scheduling of processes must be provided to support load balancing. Means for monitoring system behavior will become more important, both to aid the programmer and to provide feedback to the compile and runtime automated resource management tools. Many acknowledge that a logically consistent view of the application name space is required across the physically distributed MPP systems. At the machine level, this will be manifest as a shared address space.

Issues

No interaction among over 120 experts in a field as dynamic as ours will

achieve total unanimity of opinion. Indeed, the richness and power of such a workshop is achieved primarily through a juxtaposition of complementing and conflicting views. The Pasadena workshop benefited from a wealth of ideas and differing perspectives. For example, there was no clear picture as to the ultimate programming paradigm for parallel computation, let alone the language that should represent it. While data parallel programming and message-passing techniques are expected to play an important role, it was recognized that other forms of parallelism also are important and means of making them available to the programmer are necessary. In establishing priorities, workshop participants expressed strongly differing views about when the problem of managing heterogeneous systems should be addressed. Some considered the homogeneous system a difficult enough challenge and the heterogeneous system problem to be even harder. Others stressed that heterogeneous systems already are an important part of the HPC environment and means for their manipulation are required now. There was considerable debate about the degree of success already achieved.

It was not clear to what extent architectural considerations should be incorporated in mapping the future of HPC environments. One school advocated that economics dictated architectures derived from workstation requirements. However, it was also realized that specific needs of the HPC community might best be satisfied by modest perturbations to contemporary processors. In making tradeoffs between advanced tools and raw performance, there was much discussion of where the knee of the curve was. Some insisted that a sacrifice of even a 10 percent performance reduction was unacceptable, independent of the improvement achieved in other factors. Others were more than willing to see as much as a factor of two performance degradation if it would result in a programming environment as easy to use as today's conventional computers. Finally, there was much talk about the potential value of templates (i.e., pseudo-code representations of general algorithms that are language and architecture independent) in facilitating early portability. While many encouraged this approach, others contended that the potential effort required to develop templates would be too great to warrant their use.

Immediate Needs

Some system software and tool inadequacies must be addressed immediately. The requirements are immediate either because they are fundamental to work on the near and long range objectives or because they are sufficiently

difficult that if work is not started now and supported over the next several years, solutions are not likely in time to contribute to the HPCC Program. These immediate needs are shown below (no specific order):

- A key area of compiler technology requiring more research and development is exploitation of data locality. Data locality is important because MPPs have complicated memory hierarchies that can limit performance severely.

- Many MPPs use cross-compilers that run on workstations instead of on the MPP. Native compilation would be better because it will increase programmer productivity by reducing the wall-clock time required for development.

- One of the hardest challenges facing HPC programmers is debugging of parallel programs. Both logical and timing considerations lead to opportunities for error. Debuggers of at least the quality and capability of uniprocessor workstations should be made available on a per node basis on MPP systems. Beyond that, means of isolating and correcting timing-related, non-deterministic faults need to be devised and supported.

- Performance profiling techniques are required to reveal the behavior of the total system and its components so that the programmer can evaluate the effects of changes to program mapping. Such tools should be at least as capable as those found on conventional uniprocessor workstations despite the additional complexity of parallel flow control and distributed resources.

- Most Grand Challenge problems require many hours of execution on even the fastest computers. Therefore, a single problem typically is split into many runs of a few hours each. To facilitate restarting programs after a previous partial computation (or after system interruption), tools for checkpointing intermediate program state are needed.

- Visualization system needs in high-performance computing environments are primarily infrastructure issues: parallel I/O, availability of fast and large file systems, and common frameworks for workstations and high performance systems. Also, the development of scalable algorithms for graphics operations such as volume and polygon rendering, and grid generation will be useful.

- System programming for MPPs, especially for portable code, requires a common version of C derived for use with MPPs. A standard C library of service calls in support of MPP operation is needed to facilitate system programming and allow sharing of code across systems.

Key Results

The seven working groups were comprised of some of the best intellects and most experienced people in the HPC community. Their interactions illuminated a number of issues that are having, and will continue to have, a significant impact on the development of system software and tools for massively parallel HPC systems. For many of these issues no clear path to their resolution is now evident. Yet resolution in some form will be required if the goals and objectives of the HPCC Program are to be achieved. The key results that follow are a synthesis of those issues:

- There is insufficient experience with different programming models and languages to reach a consensus on their applicability and suitability for the various kinds of applications and high-performance systems.
- Realizing balanced TeraFLOPS-scale systems is hindered by insufficient understanding of resources required to support Grand Challenge applications. Extrapolation of previous balance points may not be feasible. Even on today's systems, I/O and file handling capability have not kept pace with processor performance, memory size, and interprocessor communication speed.
- The use of MPP systems is severely limited by the lack of mathematical software libraries found on more conventional systems. Rapid development is unlikely because of difficult new issues such as algorithm scalability and more complicated and complex data structures. Collections of templates are a possible alternative to libraries.
- Debugging and optimization of parallel programs on MPP systems are poorly supported by existing tools, in part because certain aspects of the system state that reflect system behavior are currently inaccessible by the user. Debuggers and performance measurement tools are the most urgently needed software tools.
- Programming languages and execution environments have substantial shortcomings. Any given one tends to support only one of the three

prevalent types of parallelism (data, task, and object). Yet, frequently all three are needed. Language design issues include deciding the proper semantics for parallel I/O and devising ways to specify data layout and concurrency.

- The diversity of user interfaces and execution models across the range of contemporary MPP systems limits portability of application programs and system software. Furthermore, there is little information on how to choose one over the other for a given computation.
- Resource management, now left almost entirely to the application programmer, should be achieved by an appropriate balance of responsibilities among the user, compiler, runtime system, and operating system.
- HPC systems lack mechanisms for fault-recovery, checkpointing of results, and re-establishment of network connections; yet, because their reliability is lower, recoverability for HPC systems is more important than for conventional systems.
- High-performance computing environments increasingly consist of networks of heterogeneous computing systems. Practical methods for explicit management of these distributed resources are needed in the near term, even if seamless operation is not achievable. Among the challenges are the development of scalable, distributed file systems, support of different data representations, management of remote execution, and uniform naming.
- Mechanisms for electronic dissemination of software and documentation developed in the HPCC program are needed to enable and foster practical software sharing among HPC researchers. Software sharing will substantially accelerate progress in utilization of high-performance computing environments.

Suggested Approach

The immediate needs and key results identified in the workshop indicate that much must be done just to provide acceptable support for HPC researchers. A strong consensus developed that an incremental approach to system software development is preferable to projects that attempt to provide revolutionary new capabilities. This conservative approach is more likely to achieve its more modest goals and to do so quickly, in a few years.

The workshop clearly illuminated the need for cooperation and sharing. Indeed, sharing results, tools, and program modules will be crucial to achieving the necessary capabilities and environment with the limited resources available. This is especially true because although many new elements of HPC software technology have to be built, a relatively small proportion of the worldwide software development effort is being dedicated to HPC related problems.

Software portability and interoperability are necessary if sharing is to prove effective. Key to this is the adoption of standards for functionality and interface protocols so software designers can work with confidence and their products will merge with the environments of others. Libraries of mathematical routines and repositories of source codes and test suites can and will be disseminated by means of a national software exchange over the network. The feasibility, effectiveness and efficiency of templates to transfer software modules for early portability need to be explored and developed.

Interdisciplinary efforts will be necessary to create the integrated environments demanded by HPC system operation. Industry, academic institutions and government funding agencies all must work together to set an agenda and order priorities to realize the necessary software technology.

Specific Strategy

The heart of the specific five-point strategy recommended here is sharing: shared goals, shared resources and shared results. In the scenario most likely to yield positive results, each organization emphasizes those aspects of the total problem appropriate to its own mission and shares the results with others. The five points of the strategy are shown below:

1. Stimulate commercial delivery and support of all necessary system hardware and software components comprising complete MPP computing environments, once the critical software elements have been developed by research organizations and individuals.

2. Encourage active community dialogue to define technical challenges, identify approaches, and share results.

3. Foster coordinated multi-agency support for research and advanced development in areas of importance to shared goals.

4. Fund many small research projects, a number of advanced development projects, and a few commercialization efforts.

5. Establish criteria and methods for evaluating intermediate and final results of the program as well as means for feedback of assessments to the research and manufacturer communities.

This strategy will require unprecedented cooperation among all elements of the HPC community. The vendors of MPP systems will continue to require the guidance of the users and computer science research community, just as the users will continue to require the largest, fastest possible systems from the vendors. This synergism will contribute to rapidly advancing HPC technology.

Postscript

The Pasadena Workshop clearly was not an end but a start. Even as this report was being written, a number of plans and actions were under way to continue and expand upon the results of the workshop. For example, the Federal agencies conducted several intensive meetings to address the implications of the workshop for their coordinated HPCC programs. The agencies also initiated a working group to further explore and develop the concepts of a HPC framework (i.e., a HPC system software architecture). Finally, planning was started for an Applications Workshop to extend the Pasadena work of the Applications Working Group.

PART I

Chapter 1

Introduction

The Pasadena Workshop on System Software and Tools for High Performance Computing Environments was held at the Jet Propulsion Laboratory from April 14 through April 16, 1992. The workshop was sponsored by a number of Federal agencies committed to the advancement of high performance computing (HPC) a means to advance their respective missions and as a national resource to enhance American productivity and competitiveness. Over a hundred experts in related fields from industry, academia, and government were invited to participate in this intense three day forum to assess the current status of software technology in support of HPC systems. The overall objective of the workshop was to create a stimulating collegial environment within which could emerge new understanding concerning the requirements and current limitations of HPC software technology. The goal was to provide a basis from which new directions in research and development for software technology could be established to enable and accelerate the effective application of massively parallel processors (MPPs) to Grand Challenge application problems. Attention was given both to immediate practical considerations related to current tools and usage and to longer term requirements and approaches necessary to support TeraFLOPS computing by the end of the decade. This report has been formulated with the intent to capture and present the issues and findings covered by the workshop's seven working groups, and includes reports written by the working groups' participants. The report summarizes the findings and synthesizes them into a single cohesive framework with analysis of their implications for future directions in HPC software technology development.

1.1 Context and Motivation for Workshop

For the first time in the era of high performance computing, supercomputers based on vector processing principles are being successfully challenged for performance preeminence by massively parallel processor systems. Comprising hundreds or thousands of VLSI processors integrated through high bandwidth networks, these MIMD and SIMD architectures are exploiting parallelism at an unprecedented scale while leveraging rapid advances in semiconductor technology. Together, these complementary trends are yielding the potential for performance gains of two orders of magnitude in the near term, making TeraFLOPS capability potentially achievable before the end of the decade.

The successful application of MPP technology and power to Grand Challenge (GC) problems opens exciting new possibilities in science and engineering by providing new tools for discovery and design. Availability of such facilities to the industry and research communities will result in increased American productivity and enhanced competitiveness in the international high technology marketplace. The impact of these new capabilities will revolutionize how entire disciplines engage in pursuit of their goals. New abilities in simulation will permit accelerated design cycles of complex engineering systems unimagined only a few years ago. Scientists will be able to formulate hypotheses and test their implications at an unprecedented level of resolution and detail unveiling new opportunities for exploration and understanding.

Responding to these opportunities and after extensive study, the Federal Coordinating Council on Science, Engineering, and Technology (FCCSET) recommended a national program to harness the power of massively parallel processing for applications of strategic, commercial, and scientific importance. The High Performance Computing and Communications (HPCC) program was one of only three Presidential Initiatives in 1992. The program was officially inaugurated with the December 9, 1991 signing of the High Performance Computing Act of 1991 which received strong bipartisan support in both houses of Congress. The HPCC program involves nine Federal agencies, each providing leadership in complementary ways. Of particular interest to this workshop is DARPA's role in high performance computing (HPC) systems and the role of NASA coordinating activities in advanced software technology and algorithms (ASTA). By leadership and funding in important areas, the HPCC program seeks to achieve a thousand fold increase in sustained computational performance.

While the performance opportunities afforded by scalable MPPs may

match the demands of Grand Challenge problems, the means of achieving an effective fit between the two is another matter. Significant difficulties arise in attempting to bring parallel computing resources to bear on real world user applications. These problems involve parallel programming models and methodologies, parallel resource allocation and management, and portability and scalability across machines of disparate structure and scale. Yet only these systems offer the hope of orders of magnitude improvement in compute power. Such circumstances fix a defining moment in the evolution of an enabling technology. So it is now with HPC software technology. The role of software technology in the high performance computing arena is to provide the tools, programming environment, and runtime environment needed to mediate between the user problems and the machine functionality. Under these prevailing conditions, the need for this workshop was realized by key people throughout the community who joined together to establish and organize it.

Software technology encompasses a multifaceted community comprising many disciplines. These include application algorithms, languages and compilation techniques, operating systems, software support tools, total computing environments, the emerging field of scientific visualization, and mathematical software. Other delineations are possible, but the overriding feature is the diversity of needs and methods that make up the complex of elements contributing to software technology. While all of these are manifest to some degree within the framework of conventional computation, they must be rethought in the new context of MPP systems operation. The transition to MPP technology from conventional supercomputing truly constitutes a paradigm shift for algorithms, programming models, and resource considerations. For example, adding the dimension of parallelism to the physical and abstract problem space requires almost every aspect of software technology to be rederived. Compound this requirement with the notion of distributed resources exhibiting latency/locality tradeoffs, and the nature of support demanded of software technology to provide a tractable program definition and execution environment transcends anything experienced before in the arena of computing software systems. This workshop was conceived to define the set of problems confronting system software developers in the context of the HPC challenge.

1.2 Goals and Objectives

The objective of the workshop was to bring together experts from industry, universities, and government in the field of software technology for high performance computing environments with special emphasis on massively parallel processing systems. The goal was to achieve an assessment of current systems software and tools for HPC environments and to expose areas requiring development. While it is clear that current environments are inadequate in their support of HPC application programming, less certain are the specific advances required or the priority in which they should be pursued. Therefore, it was considered a primary goal of this workshop to delineate the areas of software technology support needed for HPC systems and evaluate the degree of success current offerings provide with respect to these requirements.

The very nature of system software implies collaboration among diverse groups and activities in development and research. Even before the workshop was conducted, it was assumed that success in this arena would demand cooperative development of the needed infrastructure and tools. The size and diversity of total HPC system software environments dictate the integration of components from many sources. Success in this venture is assumed to rely on effective coordination of efforts by many teams of software technology researchers and developers. A goal of this workshop was to identify major software development challenges and approaches for improving coordinated and collaborative development.

An important aspect of the high performance community is the major government sponsorship of the Federal HPCC program. Participating agencies together sponsored this workshop. Rather than being rigid and constrained, the HPCC program is an evolving and adaptive set of activities, continuously adjusting to changing conditions and opportunities. The NASA-led coordination of the ASTA component within HPCC focuses on issues of software technology as they relate to making effective use of TeraFLOPS-scale MPPs for Grand Challenge problems. The research and development plans of the HPCC ASTA component are still in their formative stages. An ancillary goal of the workshop is to provide input to that planning process.

1.3 Tasks

The workshop was organized into seven working groups spanning a number of disciplines that make up the field of HPC software technology. These included: applications, mathematical software, languages and compilers, software tools, operating systems, computing environments, and visualization. The groups were tasked to identify in their respective areas the important systems software problems requiring solutions for the success of HPC.

The groups also were asked to categorize the important software problems along lines that distinguish them according to who should address them and how. For example, some of the components of an overall HPC environment are probably best left to the vendors to provide, while others may always remain in the domain of the users. However, because of the complexity of certain challenges or because they cross multiple conventional disciplines, there may be problems that will require government agency attention to spawn new initiatives and research. Yet, other goals may, by their very nature, require collaboration with vendors. Recognizing early on how the challenges relate to potential resources could result in near term projects to address these problems.

A final task for each working group was to provide a vision for their respective component of the overall HPC environment. While there was no anticipation that such a vision could be complete and all encompassing, an initial cut could set direction and reveal questions that must be resolved early on before final plans can be set in place.

1.4 This Report

The purpose of this document is to disseminate the experiences and lessons of the workshop beyond the immediate participants to the broader U.S. HPC community. A wide circulation is anticipated to distribute the findings to all those actively involved in software technology research and development directed to the challenge of harnessing MPP systems for Grand Challenge problems. While the workshop has made an important contribution by providing a forum for discussion of the many issues related to HPC software technology, only a dialogue across the entire community will achieve the needed perspective and incentive necessary to translate such deliberations into actions. This document is offered both as a compendium of ideas contributed at the workshop and as a catalyst to elicit broad community re-

sponse from which a consensus may emerge.

The intent of this document is to convey to its readership the issues and findings that emerged from the three day workshop. Because of the complexity of the field, the document is organized to provide the information in as clear and useful a form as possible. Therefore, the document structure and content go beyond those of a simple proceedings. It also provides a summary of the findings, a synthesis of the issues and ideas, and an assessment of the implications of the conclusions for HPC research. However, all of the original information is provided so that our colleagues can form their own opinions regarding the "raw" results.

This report is organized into four general parts. The first part describes the workshop, its objectives, organization, participants, sponsors, and agenda. The second part provides the detailed reports of the seven working groups. In addition, a summary chapter is included providing an overview of the working groups' major findings. The working groups were divided along lines mirroring the conventional disciplines comprising the field of software technology. But, the major issues dealt with at the workshop related to many or all of the groups. Part III of this report provides a synthesis of the issues and key findings into a single well organized presentation, and includes major implications for future work that are highlighted in the conclusions chapter. The last part is an appendix section that provides lists of all of the workshop participants, including the working groups to which they contributed. Also, the complete sets of viewgraphs presented by the working groups during the workshop are provided (typeset and compressed). These expose the process and evolution of ideas during the course of the meetings.

Release of the final draft of this report was timed to coincide with a panel session conducted at Supercomputing '92 by the chairs of the sessions. This panel presented the workshop findings to the community and invited comment. Reactions from the community will be accepted over the internet and assimilated by a group chartered to carry out follow-on activities. This group will release an additional report incorporating this community wide feedback. Thus, it is expected that any shortcomings of this workshop report will be reflected by the comments related in the follow-on report and together they should provide a firm basis for the community as a whole to go forward with the next generation of system software for HPC environments.

Chapter 2

Participants

By virtue of its software coordination role in the Advanced Software Technology and Algorithms (ASTA) component of the HPCC program, NASA took the lead in organizing the Workshop on Systems Software and Tools for High Performance Computing Environments. An Organizing Committee for the workshop was formed with one member from each of the nine federal agencies that are involved in the High Performance Computing and Communications program plus a representative from Jet Propulsion Laboratory, which hosted the workshop.

The sponsoring agencies were:

National Aeronautics and Space Administration
Defense Advanced Research Projects Agency
Department of Energy
National Science Foundation
National Institute of Standards and Technology
National Oceanic and Atmospheric Administration
National Institutes of Health
Environmental Protection Agency
National Security Agency

Participation in the Workshop was by invitation only. The conference organizers felt that to achieve the goals of the workshop it was important to have as small a number of people as possible yet ensure adequate representation of experts in each discipline and topic area that would be covered. Keeping the number of participants small was important if participants were to have a real workshop: one in which ideas would be exchanged, position papers discussed and revised, and controversial concepts presented and de-

bated. In addition, some of the participants needed to be familiar with the roles and needs of the individual agencies involved in the Federal High Performance Computing and Communications Program. Finally, representation was sought from users with large-scale computational applications, i.e., the eventual beneficiaries or "victims" of the research that might evolve from the workshop findings. Consequently, the process of identifying and selecting people who would receive invitations to participate in the workshop was a lengthy one.

The members of the Program Committee were selected by the committee Chair and the Organizing Committee. Once formed, the two committees formulated in detail the goals of the workshop and selected the topics to be covered. It was decided to constitute a working group to cover each topic area. Chairpersons for each topic area were chosen and each was asked to nominate ten people who are experts in the topic covered by the group.

Each sponsoring agency was allowed to nominate five participants in the workshop. These typically were researchers familiar with the agency's large-scale computational applications who could help represent the needs of that agency. Most of this set of participants affiliated themselves with the working group that most interested them.

Once all nominations were gathered, the committees pared the total number to about 100 and invitations were issued. Inevitably, some had prior commitments and could not attend the workshop. Appendix C has a complete list of attendees, grouped by committee and working group assignments.

Probably the area that was least well represented was applications. Many Grand Challenge applications have been identified that the HPCC program must support. To correct this shortcoming, a separate workshop for HPC applications will be held in 1993. One of the goals of that workshop will be to gather more information on the application program requirements for system software and tools.

Chapter 3

Workshop Organization

3.1 Purpose of the Workshop

The workshop was organized as an early step in gathering information on research and development needed in the areas of system software and software tools to improve the usability of high-performance computing systems and ensure that they will support effectively the investigation of grand challenge problems. The organizing committee formulated the following statement of purpose:

> This workshop will bring together experts from industry, universities, and government to review current systems software and tools for high performance computing environments and identify needed developments. The workshop will indicate software priorities and mechanisms for creating that software. The workshop will inform emerging HPCC research and development plans.

System software and tools include many subjects and interactions among disciplines. To provide a manageable structure, the organizing and program committees selected seven broad topic areas to be covered at the workshop:

Applications
Mathematical Software
Compilers and Languages
Software Tools
Operating Systems
Computing Environments
Visualization

The role of the applications group was to identify the requirements that HPCC applications have for the software to be discussed by all the other working groups.

3.2 Position Statements

To stimulate thinking and to establish a common frame of reference in advance of the workshop, each working group was asked to address key issues and provide assessments in a consistent format prior to the workshop. Specifically, each group was asked to write a position paper on its area and disseminate it to all the participants of the workshop several weeks before the event. It was hoped this process would enhance the output of the workshop by surfacing important issues and even differences of opinion that could be explored in detail from the start of the workshop.

The groups were asked to address the following questions:

1. WHAT ARE THE SYSTEM SOFTWARE PROBLEMS/NEEDS (from the application developer's point of view)?
   - What are the priorities of the users?
   - How is the future likely to differ from the past?
2. WHAT IS THE STATUS OF SYSTEMS SOFTWARE (from the application developer's point of view)?
   - What feedback can you provide on the strengths and weaknesses of the software that already exists?
   - What is your forecast of results expected from software currently under development?
   - What is your outlook on research expectations?
3. WHAT ARE OR SHOULD BE THE PRIORITIES FOR THE FUTURE?
   - What is next? And for each possibility, what are
     - The expected payoffs (to applications)?
     - The expected difficulties (computer science and technology)?
     - The expected time frames?

Note: Address portability as you answer all of the questions; this aspect of software is so important that it deserves special attention.

In addition, working groups could add any questions or issues that they considered relevant. All working groups did produce position papers and these were mailed to all invitees before the workshop.

3.3 Workshop Agenda

The workshop spanned three days; each was devoted to a specific purpose. In a plenary session on the first day, each working group presented the salient points of its position paper. Appendix A of this report contains the presentation materials from that session.

The second day's session was held at the Pasadena Convention Center so that each working group could meet in separate rooms that were close to each other. On the same floor we also provided rooms with computers and copiers so that position papers and presentation materials could be produced on the spot. Each group had a session in which it began modifying its position paper to take into account the previous day's discussions. In addition, working groups spent part of the day sending representatives to each other's session to ask questions, seek clarifications, or present a point of view. The applications group tried to have a representative at each of the other groups' sessions for at least part of the day. This approach was feasible because of the close proximity of all the meeting rooms and was judged to be an especially fruitful way to explore the interactions between the different topic areas.

The third day consisted of a plenary session in which a spokesperson from each group presented a revised position based on findings and discussions from the previous two days. The slides presented at this sessions are reproduced in Appendix B.

3.4 Findings Statements

After the workshop, each working group revised its original position paper to take into account the insights and information gained at the workshop. These revised papers appear in Chapters 5 through 11 of this report.

In summary, this workshop demonstrated that with care and advance preparation it is possible to have extensive and substantive interactions even with a group of over one hundred people.

PART II

Chapter 4

Summary of Working Group Findings

4.1 Introduction

The Pasadena workshop was organized into seven working groups representing the principal disciplines conventionally associated with software technology for high performance computing. These included applications and algorithms, mathematical software, languages and compilers, software tools, operating systems, computing environments, and visualization methods and technology. In no sense was it assumed that this breakdown represented an orthogonal characterization of the domain of software technology. Rather, it will be seen that many of the groups dealt with similar or overlapping topics, although from varied perspectives.

The detailed findings of the working groups are presented in the following seven chapters (chapters 5 through 11). These reports were written by the members of the respective working groups and are provided in unedited form to retain the original intent and tone of the contributors. As a natural consequence, there is a variation in style among the following chapters which, while evident, does not detract from their importance to the community.

Because the total length of the collection of the working group reports exceeds a hundred pages, this chapter is offered as a quick summary and overview of the topics considered by each group and their principal findings. A separate section of this chapter is dedicated to each group's results and every attempt has been made to retain the integrity of the original reports while achieving succinctness. Even though some information has been ab-

breviated out of necessity, it is hoped the important insights and conclusions of the full reports are conveyed in essential detail. The reader is encouraged to make extensive reference to the succeeding complete reports using this chapter as an introduction and guide.

4.2 Applications

The Applications Working Group addressed the issues of system software requirements, direction, and development from the perspective of the user sophisticated in computational methods but less familiar with massively parallel computing systems. The fundamental conclusion from this viewpoint is that near term development of tools to ease the immediate burden of programming MPP systems is a far more advantageous application of currently available R&D resources than attempting to create total solutions which will not be ready until the end of the HPCC program. Reinforcing this conviction were the dual considerations of opportunity and understanding. The markets for large parallel computers and HPCC Grand Challenge applications are relatively small compared to more conventional computing systems and the tasks to which they are applied. The opportunity to change the way high-performance computing is realized is limited in the main to R&D funds from such programs as HPCC: it is not driven by the commercial market place which invests vastly greater resources on conventional systems. Thus, the targets selected for advancement must be chosen for their potential for immediate gain. Break-even will occur only when application scientists routinely procure HPC systems, thereby providing market place incentives for their future development.

The second consideration, understanding of the form such environments should take, is simply not sufficient to move immediately into total solution system software development. The needs of scalable applications in terms of resources are at best poorly understood. Parallel programming paradigms have yet to be adequately explored, and automatic resource allocation tools are at their inchoate phase of development. An insufficient base of knowledge on all these issues exists to establish a single all encompassing vision of system software architecture. Complicating the vision creation process are the apparently conflicting trade-offs between optimal performance and portability/programmability. It was not even possible to ascertain the specific range of applications to be considered.

The overriding requirement is for some basic robust tools to enable appli-

cation scientists to port their problems and algorithms to these new HPC architectures in the immediate future, albeit with hard work. To achieve this, a philosophy of "gradualism" was embraced through which the basic tools would incrementally evolve, to ever more general and sophisticated program support facilities. Among those of immediate necessity are compile time tools for code migration that perform dependency analysis and runtime tools for timing code segment execution. Interactive compilers for user-directed code optimization are desirable but unlikely in the near future. However, there should at least be node-oriented debuggers and compilers as well as other Unix-like program development tools such as *make*. All these tools should be layered to permit user-prescribed trade-offs. All sophisticated system capabilities would be desirable, but only at incidental cost. This cost-oriented approach supporting user selection of capabilities was deemed important for successful machine evolution from initial research platforms to an ultimate dominant commercial force.

Broad exploitation of near term HPC software investment depends on: (1) the near term adoption of standards for critical components and (2) a medium of distributing algorithms and applications among distinct centers of HPC research and disparate system architectures. Standards for message passing and data parallel language constructs are being specified from the current base of experience, but less clear is the definition of the user interface to parallel I/O. At least a temporary model for this interface is needed in the short term. Still yet to be resolved is the relationship between the new operating system and HPC hardware systems.

The group strongly advised that dissemination and porting of codes and algorithms can best be supported in the near term through the adoption of a scheme of templates for representation of algorithms. Canned packages have proven too general for efficiency or too specialized in terms of machine architecture. A templates approach is an alternative to fully documented, elegant, scalable software implementation, and it offers a realistic goal for early success. Templates may convey generic parallel codes that can be modified by users to optimize for specific requirements and systems. Clear documentation and sample data sets are necessary, but the precise language is less critical.

4.3 Mathematical Software

The Mathematical Software working group considered the role and requirements of mathematical routines and libraries in the context of massively parallel processing systems. Consultation with other groups, in particular applications and languages/compilers, exposed important issues and concerns across the broader community. It has become clear that while reliance on mathematical software will increase in light of the new high performance computing technologies, realization of effective and useful software will be a major challenge. To accomplish this will require a stronger vertical integration of all system software from language design to operating system implementation so that mathematical software can be merged cleanly into the execution framework. But, there is an urgent need for usable mathematical software capability *now*, and an incremental approach is required so that a continuing stream of useful tools and functions is forthcoming. There is also a need to educate the general users of mathematical software to facilitate the application of these sophisticated routines to their end problems.

Evolution of mathematical software is driven by the needs of applications developers. The resulting requirements cover a broad range. Mathematical library routines must be portable and scalable to run on different configurations of the various new high performance architectures entering the market. Equally important is access to the internals of the mathematical software to ensure user confidence in treating such functions as a "black-box". This, in turn, requires stable, logical user interfaces and robust functionality. A dominant requirement is for some mathematical software capability for parallel processing in the immediate future. This need should override any tendency toward more ambitious but longer term goals.

There is some debate as to the real merit of mathematical software as applied to applications programs. Many difficulties in its use have emerged over years. While these need to be addressed, little question exists that mathematical software is valued by a large community of computational scientists and that these tools will be important to users of MPPs as well. In particular, those who have relied on mathematical software using conventional systems are unlikely to migrate to new HPC systems without such support.

Numeric algorithms for parallel computation are central to mathematical software for high performance computing. Development of these algorithms is complicated by the variations of underlying architectures they are intended to optimize. Additional complications arise from differences in error

generation and propagation with respect to serial algorithms. The accuracy, stability, and convergence behavior of parallel mathematical software must be resolved afresh. Greater information must be provided to the user concerning resource/performance trade-offs and likely error conditions and bounds. Data structures employed by mathematical software are not always convenient for end-user applications. With the added dimension of distributed data structures in MPPs, this mismatch may only be aggravated. To alleviate this problem, abstract data types might be standardized as an interface between user and parallel application software. This standardization could improve portability and ease of use but should not impose a significant performance penalty for structure transformation. Such translation may be provided at compile time if appropriate linking mechanisms between application code and parallel mathematical software libraries can be established.

How the user will relate to mathematical software for parallel application execution is unresolved. The major issues are usability and portability. Several approaches were examined. The traditional function library might be minimally adjusted in going from serial to parallel environments. Servers on a software bus communicating via byte streams could hide complexity of algorithm details. Generic interactive environments, even incorporating expert drivers, might be applied in much the same way as is currently done with Matlab or Mathematica. Domain-specific problem solving environments could be created to narrow the generality of interface required. Also, templates that describe the algorithms in pseudocode might be adapted by the end user to particular problems. None of these, however, fully satisfy the requirements.

The traditional approach may be necessary for immediate results but will not yield the performance gains needed in an MPP environment. Templates might appear most general but they may require more work to implement than is anticipated, and they expose the dangers of numeric uncertainties that could easily be improperly addressed by the user. High level representation of algorithms permits source code translation but precludes optimizations at the algorithm level in response to machine dependencies. These problems are most significant in the distributed memory setting. In shared address space parallel computers, the interface problem will be eased because the program name space is not fragmented.

Like applications in general, exploitation of emerging MPP technology requires that mathematical software exhibit properties of portability and scalability. But, because mathematical functions are usually embedded inside larger parallel applications, it may be easier to exploit trivial outer-loop

application parallelism and keep the mathematical software functionality essentially serial rather than attempting a new parallel algorithm. This approach, where appropriate, can provide early utility. It does not address the broader question.

Developers of mathematical software have requirements which must be satisfied by the HPC environment. An overriding concern is that the mathematical software writer retain sufficient control to guarantee numerical behavior. The compilers should respect parentheses in generating code for floating-point expressions. The runtime environment must provide information about the properties of floating-point arithmetic, precision of expression evaluation, exception handling, and mapping of distributed data types. Access to exception handling facilities needs to be permitted at low overhead costs. Compiler analysis results should be provided to aid in software tool development. Finally, parallel prefix for arbitrary user-defined associative operations should be supported.

4.4 Languages and Compilers

The combination of programming language and compiler establishes the logical interface between user application and the operating execution environment. Together these two components of the computing environment contribute significantly to the effective performance of the overall system as well as to its ease of use and the portability of its application code. The Languages and Compilers Working Group considered the problems confronting users and system software developers in matching the capabilities of MPPs to the requirements of user applications.

The programming languages for massively parallel processors must go beyond conventional languages in expressivity. In addition to the normal requirements, useful languages for MPPs must incorporate semantic constructs for delineating program parallelism, identifying locality and specifying data set partitioning. The broad range of HPCC applications may embody a rich diversity of program structures. It is anticipated that no one parallel programming model will satisfy all needs. Data parallel, task parallel, and object parallel forms have all found applicability to end user problems and are expected to continue to do so. A parallel programming language will have to permit user accessibility to a mix of parallelism models, either by incorporating them into a single schema or by providing means for a single application to use different languages or different parallelism models for sep-

arate parts of a user problem. In either case, languages and compilers will of necessity support interoperability among independently derived program modules, perhaps even running on separate computers in a distributed heterogeneous computing environment. Adequate return from the investment in MPP software development depends on the ability to combine independently developed software modules.

Languages must provide the means to express locality relationships among tasks and data objects to minimize performance degradation from access latency and data migration. What the most general and useful characterizations of locality are and how they should best be represented are open questions. Sufficiently rich languages provide the programmer with multiple ways of representing the applications. Alternatives must be clearly related to costs in terms of resources and performance so that optimal trade-offs can be made.

Compiler development will have to be extended to translate new language semantics for control of parallelism into optimized parallel system operation. Critical among these is exploitation of locality. Management of the system memory architecture hierarchy, data placement, and coherency maintenance as well as interprocessor communications are all strongly influenced by exposed locality. Compilers native to given processor architectures are required to achieve highest performance through hardware structure-driven code transformations. But, these same sophisticated code manipulation methods can obscure the relationship of the actual processing activities to the original user source code. User productivity is largely derived from this binding because both program development and performance optimization rely on it. Advanced compilers will have to provide detailed information about code optimizing transformations so that debugging and performance monitors can present their findings to the programmer in a meaningful (source code related) way.

The working group identified key avenues of pursuit considered critical to the HPCC program that should be conducted during the next two to five years. Language features and compiler extensions should be investigated in such areas as parallelism, data placement, parallel I/O, and exception handling. Compiler optimization techniques should be extended in resource allocation including program partitioning and mapping with an emphasis on locality control and task scheduling. Robust programming support environments for successful experimental languages and compilers should be implemented and made accessible for rapid assimilation by the GC application community. Methods for evaluating alternative parallelism models and

computing strategies must be devised for meaningful comparisons of research results. Achieving these goals will involve research and development efforts as well as an environment conducive to sharing of results, collegial review, and funding.

The working group identified three stages of evolution for such work: research prototype, advanced development prototype, and commercial products. Many research prototypes reflecting innovative but untested ideas would receive modest funding. A few of those would be selected through a process of objective evaluation for advanced prototype development, the product of which would be disseminated among diverse research groups to acquire extensive experience. Ultimately, one or more of the most useful and successful of these would be commercialized and made available to the HPC community.

It is realized that many enabling factors within the HPC community can greatly contribute to the likelihood of success, should they be aggressively encouraged. Strong collaboration between user and system software development will greatly ease acceptance of new technology and address most urgent needs first. A shared approach to evaluation of results will favor early recognition of quality research results. Wide availability of networks and high-performance computers can accelerate ultimate adoption of research results by production environments. A globally accessible repository for software, documentation, and test case data will strengthen and bind together the community and greatly ease the sharing of project results and tools. Finally, the early adoption of standards at all levels of the logical and physical systems will make possible interoperability of diverse program modules, greatly enhancing the productivity of the entire community.

4.5 Software Tools

Adequate means for program debugging is foremost among the tools needed for the development of MPP applications software. The software tools working group observed that manufacturers of MPP systems do not offer debugging facilities capable of systematically detecting and isolating program errors and relating them back to the original source code. Parallel processing complicates the task of analyzing erroneous system behavior. The lack of proper debugging facilities severely hampers effective parallel application program development.

Beyond the more common determinant errors encountered with conven-

tional sequential programs, parallel processing introduces the possibility of transient errors which are not easily repeated or diagnosed. These are caused by timing conflicts, such as race conditions in shared memory systems and send-receive mismatch and buffer overflow in message-passing systems. Such errors are particularly difficult to isolate and correct. New debuggers must be devised to represent the abstract program state of the parallel programming paradigm so that transient errors can be related back to the source code.

Extensive compiler transformations for automatic optimization further complicate parallel debugging. The debugger must have access to information about these transformations in order to relate machine operations to their respective source code statements. Ideally, the debugger should be able to detect and isolate timing errors. But, without such sophisticated techniques, the debugger must at least be able to characterize sequences and timing events. A factor complicating debugging of parallel software is the intrusive nature of debugger operations which can easily perturb the real behavior of the native program, thus obscuring the original errors or injecting new ones. Virtual time strategies may have to be employed to ensure fidelity of program behavior characterization by the debugger.

Even if a parallel application code is executing correctly and producing valid result values, performance may exhibit significant variation depending on the relationship between the program computing demands and available parallel computing resources. Programming parallel computers presents many more degrees of freedom than the equivalent task with uniprocessors and therefore requires many more decisions of the end user or his system software surrogates. The challenge of correctly associating application program tasks with parallel computer resources requires user access to system performance information.

The Software Tools Working Group identified performance evaluation tools as among the most critical for effective exploitation of massively parallel computing. Such tools convey how well a program is performing on a given parallel machine, how performance would alter should parts of the program be modified, and how performance would scale with changes in problem and system size as well as changes in architecture. Not only is such insight essential to the application programmer attempting to realize portable optimal code, it is imperative to system software and hardware designers developing the next generation of high performance computing systems and strategies.

Performance is affected by a myriad of sometimes subtle influences during the interplay of hardware and software. These influences include pro-

gram parallelism, variable latency across the hierarchical memory structure, overhead of parallel resource management and task synchronization, and contention for shared physical and logical resources. Revealing the degree of contribution of each of these factors and relating them to user decisions in the application programs is the combined responsibility of compiler and runtime performance analysis capabilities. The compiler must establish the relationship between performance measurement data and program structure as represented by the application source code. The compiler also must perform some analysis, insert software measurement code for traces, and control hardware instrumentation. Because software for performance measurement is intrusive, extending the execution time and potentially altering the sequence of critical program events, hardware instrumentation may be necessary for certain fine grain measurements. Hardware support also must be provided for measured parameters that fall outside the programming model name space. Finally, the wealth of performance related information returned by the system challenges the user's capacity for data assimilation. Advanced methods of visualization, derived expressly for performance information, will be needed to minimize the program optimization cycle. Ultimately, the same techniques of measurement, evaluation, and code modification may be incorporated to support automated compile time and runtime optimizers. All of these problems are extended in difficulty when applied to larger heterogeneous parallel computing systems.

While debugging and performance tools were cited as those most urgently needed to facilitate effective use of MPPs, many other tools could be useful as well. Among those, support for global shared address spaces and source-to-source translation were considered explicitly. Shared address space greatly simplifies application and system programming. All variables, whether local or remote may be accessed by the same constructs. As the program or host system upon which it is running changes, the resulting changes in relationship between variables and their supporting processors need not require modifications to programs. Runtime resource allocation is greatly simplified because any processor has direct access to all memory resources. This does not eliminate the need for locality exploitation to minimize latency, but it does ease the complexity of program control.

Many experimental tools to manage parallelism involve the need to automatically convert an original source code to a new source code reflecting modifications and augmentations resulting from analysis. Meta tools to facilitate building source-to-source translation tools in the context of parallel codes would accelerate the rate of experimentation with alternate strate-

gies. An important aid in this domain would be the inclusion of hooks in compilers supplied by vendors to provide direct access to intermediate form representations of source code. Finally, it was the opinion of the working group that even if the many laudable but aggressive objectives above could not be met by the research community for some time, it was imperative that minimalist debuggers and performance measurement tools be provided by vendors on a per processor basis in the immediate future.

4.6 Operating Systems

The purpose of a computer operating system is to reflect an abstract user interface to the hardware and firmware resources, provide protection from other programs resident on the same system, support a file storage capability, and provide response to exception conditions. With the emergence of high performance computing systems that exploit parallelism at many levels, the role of the operating system and its relationship to program execution is changing in response to the demands and constraints imposed by these new systems. The operating system working group examined the implications of this new relationship as it affects the development of future system software scalable to TeraFLOPS performance.

Conventionally, the operating system is responsible for resource management and job scheduling. However, where parallelism is a source of performance gain as in MPPs, the finer granularity of tasks makes the overhead of kernel, calls prohibitively expensive for thread scheduling and message passing. Instead, a runtime system that exists within the application address space is used for this functionality. Temporal costs are greatly reduced in this manner because function calls are used instead of trapping to the kernel and application derived knowledge can be applied to permit lighter weight, more specialized functions to be employed.

For very large applications executing on MPPs, the loss of execution time due to premature termination can be unacceptable. An important capability for future operating systems is recoverability of application program state part way through its execution, permitting restart to completion. Long running applications need to be able to survive hardware and software faults, system crashes, unintentional network disconnect and logouts, and other forms of interrupted service without having to restart from the beginning. Key to achieving execution resumption is operating system support for automatic checkpointing and restart. User access to checkpointing facilities

from program code is necessary to optimize recoverability while minimizing the overhead of checkpointing. Both software and hardware system development must reflect the need for robustness through restart. Many unresolved issues still undermine realization of production level recoverability on MPPs. Among these are migration of checkpoint files to other machines, reducing checkpointing image sizes, compiler help to optimize checkpoint file organization, hardware and OS kernel support for checkpointing, and checkpointing of distributed programs across heterogeneous systems. Related to recovery from unintentional termination, is exception handling. Here, hardware support for managing flag mechanisms is essential, but operating system support is of less importance. User application code should be able to dictate the response actions to be taken under exception circumstances.

The central role of the operating system for managing the file system and general I/O is complicated by the exploitation of parallelism in MPPs. I/O performance must keep pace with increasing execution rates, although I/O mechanisms are intrinsically serial in nature. Parallel I/O functionality needs to be defined and standardized to eliminate ambiguity while benefiting from parallelism when practical. File storage capacity and transfer rates can be augmented through technology, but system software must be accelerated as well so as not to become the bottleneck. An added complication involves distributed file systems accessed by means of networks. Reliability, latency, throughput, and security are all compromised by remote storage of files. The operating system, through appropriate network file system protocols, negotiates with remote operating systems to affect the file transport, thus forming a heterogeneous system in the process. More than any other aspect of distributed computing systems, file and I/O management may prove the Achilles' heel of massively parallel processing.

The operating system is responsible for memory management. For MPPs the memory hierarchy is substantially more complex than for conventional uniprocessors. A wide range of memory access latency can be experienced depending on the relative location of the desired data with respect to the soliciting processor. Contention among multiple processors for shared memory units and communication channels further degrades access times. Program mapping at load time or runtime must consider this complex trade-off space. Memory management strategies supported by the operating system play an integral role in these decisions and control. Currently, distributed memory MPPs fragment the system memory, identifying blocks exclusively with individual processors. Remote access to a variable requires explicit message passing and local processor servicing. The operating system needs to man-

age all memory within the system. It is much easier to write system software (and application software) if there is one uniform address space throughout the system. At some level of abstraction system software designers should see a global name space. This does not preclude optimizing for locality, but it does simplify memory management. Also, these early generation MPPs employ physical address mapping. Program scalability is one of many advantages to employing virtual memory translation. Managing a distributed virtual memory is complicated if the task is to be done in parallel. This will be necessary if operating system functionality is to scale with parallel computer size.

Other issues related to operating system support for MPPs include general resource management, debugging and performance monitoring, and message passing. While all are reflected by conventional multitasking uniprocessors, the additional burden of parallelism complicates each of these support functions. Among the questions to be resolved is how multiple jobs are to be supported simultaneously on a given MPP. Currently, such systems are partitioned by groups of processors with some set of processors dedicated to a particular job. An alternative is to time slice processors such that a processor picks tasks from a global queue supporting all active jobs, or even allocating fixed time segments to a set of assigned tasks on each processor. The complexities of achieving this functionality go beyond current support and the cost-benefit trade-offs are not clear. The operating system role may be extended to incorporate all elements of heterogeneous systems. Whether this is done as a single operating system or as an ensemble of collaborating operating systems has yet to be determined. But at some level of granularity, protocols and strategies for coordinating mutually supporting operating systems over WANs working on a single problem will have to be devised.

4.7 Computing Environments

The Computing Environments Working Group addressed the wide array of issues associated with the principal classes of resources comprising the high performance computing systems of the future. The objective was to identify the needs for system software in managing the total system infrastructure as it is brought to bear on Grand Challenge applications at TeraFLOPS levels of performance. Of particular interest were the application driven requirements for the computing environments envisioned and their scaling behavior for I/O bandwidth and I/O caching. The three basic areas considered were

data support systems, communication support systems, and heterogeneous computing environments.

These questions were examined within the framework of a computing environment hierarchy which is expected to encompass all systems within the foreseeable future. Parallel computers are ensembles of equivalent computing elements and include vector supercomputers, SIMD processors, shared memory MIMD, and massively parallel distributed memory MIMD systems. The Meta-computer layer is composed of a heterogeneous collection of disparate computer systems and data storage systems integrated by high bandwidth LANs, all at one site. The Meta-center is the top layer and constitutes multiple computer centers connected by a very high bandwidth WAN. Associated with each level are the resource management and programming support software necessary to effectively apply the computing facilities to end-user Grand Challenge scale problems.

Porting and rewriting of applications programs require a support environment that encourages code reuse, portability among different platforms, and scalability across similar systems of different size. Achieving these requires standardization of key programming elements such as parallel I/O and message-passing primitives, and parallel programming support tools such as those discussed by the Software Tools Working Group. In addition, appropriate Unix tools and accounting mechanisms are necessary to the parallel computing context.

The data storage requirements for applications operating in the regime of TeraFLOPS performance are unclear at this time. An important near term result of the HPCC Program will be estimates of data storage scaling behavior for key Grand Challenge problems. What is known is that the magnitude of data managed by these systems will be far beyond the capacity of today's largest systems. Future file sizes may scale with memory size, but in any case can be expected to exceed 30 Gigabytes. Many questions remain concerning the best way to organize such large data sets to achieve acceptable file retrieval times. Archival storage systems will be more tightly integrated with primary disk storage systems; the latter caching the most recently used objects resident on the former. New storage technologies will undoubtedly play an important role by changing the balance of the component elements, but not for several years. Meta-computers and meta-centers will require transparent data exchange with common formats for which standardization in the near future will be needed. Ultimately, all main file systems across the meta-centers will be integrated into a single uniform file name space creating a National File System. But the resulting need to rapidly access a

wealth of shared data will demand that research questions concerning data compression, data privacy, and data integrity be resolved.

Application programs being performed at a TeraFLOPS performance level will generate data at rates exceeding a Gigabit per second. Development of software systems to manage such communications locally and across wide areas is considered the pacing technology along with development of the actual high bandwidth communications media. In question is the relative benefit of packet switched versus circuit switched protocols in achieving such throughputs. Where applications exploit parallelism across meta-centers, the degree of latency imposed by the communication subsystem as well as the degree of uncertainty may have a profound influence on the structure of the programs. Extensibility of network capacity will require models of local environments to include parallel channels to meet increased demand. While the the apparent speed of a network would need not exceed the memory bandwidth of a given computer, the integration of multiple computers in a meta-computer can impose peak demands exceeding those levels where communication channels are shared. The need for checkpointing as well as the use of performance instrumentation will aggravate demands on communication systems. Error handling standards will have to be developed and applied with the increased opportunity for data corruption as a consequence of heightened data transport. Systems that rely on assumed correct data may experience insufficient reliability in the environment of TeraFLOPS computing systems.

The motivation for migrating to heterogeneous environments is the potential to apply the most suitable computing systems to the different parts of a given program, better matching the resources to computing needs. Some consider this an opportunity to achieve super-linear speedup. Realizing this possibility will require the development of a uniform programming support environment and a software resource management infrastructure. Both are made more difficult than their homogeneous system counterparts because of the complex trade-offs between data locality and granularity across component systems. Analysis tools will have to be provided to assist in determining the mapping of activities and data to computing elements. The optimal balance point is a function of many factors, including communication bandwidth and latency, size and power of the constituent processors, and software overhead for coordination within the heterogeneous environment.

4.8 Visualization

Massively Parallel Processing systems offer the opportunity and challenge of generating, manipulating, and managing vastly greater amounts of data compared to conventional computing environments. User understanding and response to this wealth of information is impracticable by normal methods. Scientific Visualization is emerging as an ensemble of techniques for presenting data quickly and compactly in a form amenable to rapid user assimilation. While exciting in its recent successes, current tools and systems for scientific visualization will be severely stressed by the demands imposed by even near term MPP systems. Anticipated TeraFLOPS-generation MPPs would overwhelm contemporary visualization systems. The Visualization Working Group considered the functionality and demands of visualization, the state of the art in this field, the challenges facing system software supporting visualization, and an approach to evolving such tools to meet the needs of TeraFLOPS computing environments.

Computational scientists working on Grand Challenge problems employing MPP architectures will rely on visualization tools for managing the abundance of associated data. Foremost among requirements is rapid high resolution presentation of massive data sets generated either from simulation or as raw output from other systems such as remote sensing. Visualization of multivariate data structures challenges current methods both in form of presentation and in speed of enabling technology for storage, transfer, and rendering. However, the demands imposed by direct presentation of a data set are dwarfed by the requirements for data navigation and animation. The former permits the user to conveniently move through a target data set by examining graphical abstractions of different portions of the data being searched. The latter presents time dependent results as a time varying sequence of frames. Both require a multitude of visualization images of the target data set to be generated and displayed in near real time.

Visualization also can be extremely valuable in the development of application software; both in its debugging/optimization and during execution. Debugging is helped by showing program state among potentially thousands of variables. Optimization of resource allocation is greatly aided by depicting the relative utilization and other metrics of processor behavior in large multiprocessor configurations. Many simulation applications are interactive requiring user adjustment of critical parameters during run time. Again, visualization of key datasets on an ongoing basis is important in providing key user feedback. In all these cases, scientific visualization is extremely

valuable, even essential, in effectively managing the vast amounts of data associated with emerging massively parallel high performance computing.

The opportunities for exploitation of scientific visualization in the realm of high performance computing are constrained by current inadequacies in performance of the graphics infrastructure and the usability of such systems. Visualization systems custom designed for a particular application and host computing system/environment are too costly and restrictive for broad community benefit. The software must be easy to use by general scientists for a wide array of applications. Thus, reuse of visualization software is essential. Modularity of software with clean, well-defined interfaces will support both ease of programming and software reuse by allowing sophisticated application-specific visualization systems to be simply derived from a collection of predefined general function modules. Retention of investment in learning curve and software development requires portability of software modules among extant platforms and scalability to massively parallel processing systems of increasing size and complexity.

Performance limitations of current visualization systems must be addressed at all points within the visualization system support infrastructure. Dominant among these are disk and I/O bandwidth. But uniprocessor performance, even with special purpose acceleration hardware, will not keep up with the demands created by evolving high performance computing systems. While these areas are conventionally dealt with by operating system and computing environment design activities, the requirements of visualization exceed the capabilities provided by those communities. As a result, the majority of work engaged in by the visualization community is dedicated to system software infrastructure development.

The Visualization Working Group concluded that the key approach to addressing this challenge is development of distributed visualization environments (DUEs) that exploit parallelism in a heterogeneous framework of massively parallel processors, graphics workstations, RAID file storage servers, very high bandwidth fiber optic LAN, and high data rate WANs. The distributed nature of visualization systems is enhanced by the expectation that data sources, application computation hosts, and graphic presentation terminals may be widely separated, even spanning the country. A plug and play programming style permitting graphical dataflow structures of functional modules is anticipated where the modules may be distributed among appropriate elements of the heterogeneous system or different processors of a massively parallel processing system. Successful exploitation of this technology will require better development tools including an effective C based

programming capability for MPP systems and robust interfaces to Fortran and its emerging data parallel derivatives.

Chapter 5

Impact of Grand Challenge Applications on HPCC Software and Tools

Geoffrey Fox, Chair
Sanjay Ranka, Deputy Chair

5.1 Introduction

This paper presents the issues for development of tools and software for HPCC from the perspective of Grand Challenge application scientists. The views in this paper reflect the input of only some dozen individuals, although in many cases the individuals reflect experience of many others. However, this small number cannot cover the breadth of requirements of diverse applications and review of our conclusions by a broader group of users is essential. We note that our working group's conclusions are quite conservative, stressing the importance of practical, relatively near term, often seemingly mundane, goals. Probably the members of our group were more experienced than the "average" user of a parallel machine and we would expect our conservative approach to be endorsed and perhaps even emphasized by the broader community.

Parallel computers will only succeed if application scientists find them useful and purchase them. This leads to a set of sociological and technical requirements for HPC software. These range from qualitative goals, such as offering evolution paths from mainframes, workstations and personal

computers, to precise specifications of high performance Fortran directives needed to support a particular application area.

In the following section, we describe the context or ground rules under which our working group decided to base its conclusions. In section 5.3, we present our findings, while in the final section some application-specific remarks are made.

5.2 Context for Application Development and HPCC

5.2.1 User Profile

We considered that most HPC applications would be developed by talented and computationally experienced users who may not be very experienced in the use of HPC architectures. This model certainly describes the Grand Challenge teams being set up by DOE, NASA and NSF as part of their component of the HPCC Program. The two industry members of our group agreed on this for the initial introduction of HPC into their organizations. We assume that a broader range of users will become involved with HPC programming in an evolutionary fashion, as the hardware and software matures. Educational programs will be needed to train the initial cadre of users whom we expect to be knowledgeable in conventional computation. Later, education and training programs will be needed for the broad range of users.

Our model for the user suggests that software tools can assume that the programmer would be willing to "work hard" and that "complete solutions" would not be necessary, but rather, the systems software should view parallel programming as a collaboration between user and the software tools.

5.2.2 The Performance Trade-Offs and Types of Codes

Any software and hardware system makes trade-offs between performance (cost-performance), portability and ease of programming. The research community, perhaps typified at one extreme by a graduate student, would demand peak performance with codes of modest size (1,000–10,000 lines) with only a life span of a year or two. At the other extreme, we could see a Fortune 500 company developing a million line code which could be used with continued upgrades for twenty years. This HPC application would sacrifice performance for scalability (portable to future machines) and error free programs designed for easy evolution. Here the needs of HPC overlap with the

goals of software engineering. It is worth noting that DoD estimates that the costs of a large software project can be divided into three stages with only 10% of the cost for initial development, 20% for testing, and 70% in maintenance over its life cycle.

Thus, HPC software systems must support a wide range of trade-offs and, as we start developing major systems, consider not only the initial development but also the maintenance and ongoing evolution of large codes.

5.2.3 The Application Domain

Our group had spirited debate as to the range of applications we should consider. The HPCC Program is built around a set of Grand Challenges which can be expected to need TeraFLOP and higher performance machines. These Grand Challenges are critical to the missions of DOE, EPA, NASA, NIH, and NSF—the latter reflecting fundamental research in the academic community. We understand that the initial goals of HPCC software should be to support applications which these Grand Challenges represent.

However, we note there are important government and industry applications that could make good use of HPC, but have very different software requirements than the designated Grand Challenges. Two examples are a battle management system, and the information management, design and simulation support of concurrent engineering.

The particular application focus of HPCC is important and should be borne in mind when reading both this report and those of the other working groups.

5.2.4 The Real World

High performance computing is predicted to gain great importance; a Department of Commerce report estimates that its economic impact will be $100 billion worldwide by the year 2000. However, at the moment, parallel computing is a minor component of the total computing arena. Personal computers, workstations, mainframes running large information systems, are each more than an order of magnitude larger than the parallel computing arena. Further, scientific computing—the current application focus of the HPCC Program—is not in total a large market. Teradata, a parallel database company, is currently the largest parallel computer vendor.

The small size of the HPCC field today has implications for software systems that may not be very technical, but are nevertheless important. Parallel

computing needs to offer great cost performance advantages over conventional approaches to entice users to make the switch. HPC software systems need to consider carefully the implications of migrating codes and users from the dominant software paradigms of conventional computing.

5.3 Requirements for Systems Software and Tools

5.3.1 Gradualism and HPCC Software

The working group took a conservative attitude summarized as "gradualism" by the mathematical software group. In general, we suggested that HPCC needed far more emphasis on modest, robust software than on complete all-encompassing tools, compilers, mathematical libraries, operating systems and environments. The visualization group termed these needs as systems infrastructure. The operating environments on the different parallel machines are at different levels of maturity. The sample requirements listed below are satisfied on some current systems, but not all.

We did not doubt that a fully functional High Performance Fortran parallel compiler would be useful. We agree with the compiler group that long-term priorities include high performance, portability, usability, consensus on a few key programming paradigms, and their integration into the programming environment. However, it was suggested that these long-term goals be accompanied by important short-term objectives.

Software tools are two types: compile-time and run-time. For the compile-time type we need tools which will allow us to analyze the data accesses and modifications to arrays everywhere in a code. In particular, a simple migration tool would be a very useful product. This could be built around the High Performance Fortran compiler but would assume the user responsible for the parallel code. The migration tool would supply dependency and other code analysis but not make any parallelism decisions. The following simple features would be desirable: the number of clocks a loop will execute, detection of cache associativity conflicts, MegaFLOP counters, and a run-time ability to instrument code through the debugger, so that timing results can be obtained on portions of code. Both elapsed time and processor time are required in order to infer memory system delays. Further, the compiler should be interactive. It should be able to ask you questions, if necessary, whose answers will help it produce better code. In other words, it could

support dialects of FORTRAN having compiler cues to help optimization. Other important sample tools needed include good, robust node debuggers, node compilers, and a UNIX make command. Stated more generally, we want the basic UNIX (operating system) tools to be available on the parallel machine. These are tools with no deep parallelism issues except that they need to recognize the individual nodes and that a single user application consists of an ensemble of programs running on different nodes.

We endorsed the philosophy of the operating systems working group that an HPC O/S should be able to deliver high performance. It should get out of the way when it is not needed and not over-engineer and hence lower performance of its services when they are required. This probably implies a layered structure where the user can obtain from the same O/S a given service with different trade-offs between functionality and performance. For instance, message passing internally to an MPP requires low software overheads; however, on a network with lower bandwidth and higher hardware latency, message passing can afford more software overhead and could use this to implement a more robust protocol.

5.3.2 What Do We Want?

Many working groups asked us what the applications needed in the areas of software tools, machine capabilities, mathematical software, etc. We suggest a different approach—as we will surely take anything offered. Rather, we suggest that you ask us to choose between a set of trade-offs. For instance, we stated in section 5.3.1 that we would rather have some modest tools in the near future, even if they delay a system offering automatic seamless parallelism. We were asked if we wanted virtual shared memory, automatic checkpointing, and accurate treatment of race conditions in debugging. In each of these three cases, we certainly wanted these features if they were zero cost. However, we were doubtful if we were willing to pay a significant price in either performance or dollar cost for them.

5.3.3 Template Codes

We proposed an experiment for improving the communication of parallel algorithms (such as a parallel FFT or adaptive multigrid) and applications between users. Fully documented, elegant, scalable software implementations would be a wonderful goal for either mathematical libraries or a new parallel application to be distributed among several users. However, this

may be too ambitious and unnecessary. We suggest templates as an alternative to fully functional library routines. These would be generic parallel codes that could be modified by any user. These modifications could involve optimizations for a particular parallel platform or tuning the functionality of the algorithm. It was noted that "canned packages" tended to be either too general or too specialized. At this stage, we are in an exploratory mode with parallel computing where both the needs and implementation are hard to specify. Obviously, a template requires more effort from the user than a full, functional, flexible library. However, remember that our ground rules implied sophisticated users. Further, fully functional libraries are hard to produce, while templates are easier to produce and would be of great value. Thus we felt it was better to set a realistic goal (of templates) and succeed quickly, rather than wait for the design and implementation of libraries. Currently, much duplication exists in algorithm development for parallel computers. One possible reason is we lack a realistic way to communicate results.

Experimentation will be needed to determine the best template formats. We suggest clear documentation and the provision of sample data sets so the user can test implementation on at least one computer system. We felt we should not specify a precise template format. In particular, the language used is not important—Fortran, C, C++, and Pascal are all reasonable choices. Templates could have significant educational value in both courses and training sessions. They would enable the communication of results between different application disciplines and between users and the mathematical software community.

The arguments in this section also show the importance of the availability of source code for any parallel algorithm or application to be used outside the group that originally developed it. It is likely the original implementation will need changes due to either a new machine or environment changes such as a new message-passing subsystem.

5.3.4 Standards

We strongly support the development of standards appropriate to HPC. In particular, we believe the community has gained sufficient experience with message passing and data parallel languages to realistically codify the experience in the form of standards. We are concerned that other trends towards standardization may be premature, focusing as they do on standards primarily designed for sequential architectures (POSIX) or shared memory systems

(MACH, OSF/1). We are particularly concerned that some standards, e.g., POSIX on every node of an MPP, that were treated as unassailable and self-evident facts framed the OS working group's discussions. Unfortunately, time constraints prevented us from bringing our concerns directly to the attention of the OS working group.

We are also concerned about a perceived difference in "world view" between the OS designers and the application writers. Generally speaking, application writers perceive they are writing a single program which happens to run, with occasional communication and synchronization, on many processors. (Obviously, there are important exceptions to this generalization.) Resources like I/O and CPU, which are allocated by the OS, are thought of as pertaining to the entire application, i.e., the potentially large collection of individual processes. Conversely, OS designers perceive as a process the fundamental unit of computation. Individual processes are assigned resources (CPU, I/O descriptors) as they request them. Some attention is paid to the fact that multiple processes may be acting in concert, but not nearly as much as the application writers may expect. We believe this difference in viewpoint stems from background. OS designers are steeped in the traditions of distributed computing over networks. Client/server models, issues of name binding and reliability in the face of uncertainty are critical. Conversely, application writers are used to writing a single program to do a single task. They would prefer to hide as much as possible the fact their single program is actually running on a number of individual processors. The issue becomes painfully clear when one discusses parallel I/O. A participant asked the floor what one should expect from calling the C library routine `printf` (equivalently PRINT in FORTRAN) from a parallel program. The answers were varied and mutually incompatible. They clearly reflected the divergent world views associated with the OS and applications camps. Apparently, the answer is not likely to be addressed by the emerging specification for High Performance FORTRAN. At the very least, we conclude that a temporary working model for parallel I/O acceptable to both OS and application designers is urgently needed.

5.3.5 The Environment

The environment working group asked us searching questions about the resource needs for future parallel computer service centers built around a TeraFLOP parallel machine. We could not give authoritative answers to requests for needs of CPU memory, disk and archive as a function of available

machine performance. We were sure that applications would use all they were given and so the philosophy of section 5.3.2 is important. We need to examine trade-offs between different resources with realistic cost assigned to each component. We anticipate that simple linear scaling of the current use at Cray Supercomputer centers will be misleading. However, this is an important area and we suggest it be examined by a qualified group.

Our group was not enthusiastic about discussing the needed support for a seamless distributed heterogeneous collection of MPPs. We thought it would be better to understand a single seamless homogeneous machine first! Our conservative comments did not apply to functions such as visualization and common file systems which are already distributed. Clearly this practice will and should continue.

5.3.6 Computational Science

One can argue that parallel computing will require closer ties between applications and computer science as an ongoing practice. Thus, successful use of a parallel machine requires mapping (matching) the application to the machine requiring the user understanding the machine (as in the Fortran plus message-passing programming paradigm), or systems software understanding the application. In the latter case, languages such as high performance Fortran need directives which are based on application requirements. Thus we can argue that development of computational science educational programs will help to produce and properly evolve effective systems software.

5.4 Particular Application Requirements

5.4.1 Visualization

The visualization group identified a selected set of Grand Challenges, including computational chemistry, geophysics, fluid dynamics and plasma physics, that are particularly important. They also gave a complete analysis of I/O requirements that could have general applicability.

5.4.2 Industrial Users of Electromagnetics, Fluids and Structural Simulations

An industry meeting sponsored by United Technologies produced a concept similar to the templates of section 5.3.3. The participants suggested collab-

oration in forming "non-proprietary codes". This base set could be shared among different companies that could extend the "templates" by optimizing them to proprietary code for their particular applications. For fluids, this could vary from internal flow in an air-conditioner or a jet engine, to airframe simulations.

This industry group also has stressed the need for standards for multidisciplinary design environments which integrate fluid flow, structures, thermal, propulsion, controls, and perhaps radar signature.

5.4.3 Financial Modeling

High performance computing could have a major impact in financial modeling. Further, the dollar volume in many problems could justify very large MPPs. The financial community is interested in portability, but especially rapid prototyping. The cycle of a typical financial application is very short—a few weeks to a few months.

5.4.4 Battle Management, Command, Control, Communication, Intelligence and Surveillance

The software that enables the integration and operations of large complex systems is the focus of this application area, often referred to as the "systems of systems" problem. In the language of section 5.4.2, it is a multidisciplinary application with a very broad suite of disciplines to be integrated. We characterize this as a concurrent ensemble of multiple, interacting functions, and include functional parallelism, which may be geographically distributed, data parallelism, and control parallelism. The DoD Software Technology Strategy, December 1991 (draft), explicitly used two "systems of systems" concepts envisaged for 2007 to motivate their technology requirements: Integrated Combat Systems (ICS) and Corporate Information Management (CIM). We have adopted the term BMC3IS, for battle management command, control, communications, intelligence, and surveillance to more explicitly express the challenge of this difficult software application area.

A BMC3IS system is required to collect, integrate, and present information to a human decision maker, the Commander, and to distribute derived tasking for execution by others. Critical issues for the BMC3IS system are controllability and observability if it can be adequately tested and its performance predicted with confidence. Leading approaches to address these issues are the use of simulation and "test-beds," a software version of a

wind tunnel for the BMC3IS test article. Such tests must allow inclusion of highly nonlinear effects such as Commander error or natural or induced environments. The complexity of the test-bed software can daunt that of its intended test article. Tools and environments are needed for developing these systems

5.4.5 Environmental Modeling

Currently, second generation regional scale models run at about 100–200 MegaFLOPS on a CRAY Y-MP. The EPA is developing a "super model" system (Models3) to address issues critical to environmental analysts and regulators over the next generation. For Models3 to be of significant use, the EPA estimates it will need at least 1,000 times the current computing power within the next four years. This seems attainable only through algorithm and software development that takes maximum advantage of the emerging massively parallel and distributed computing environments. To get this performance the task requires everything, from compilers and profilers to high-speed communications protocols and distributed operating systems.

After 1996 the performance/power needs for modeling will continue to soar as scientific improvements are added to the system and reasonable turn-around times are provided for regulators and scientists to conduct credible and defensible studies into the causes, fate and effects of environmental pollutants.

Chapter 6

Mathematical Software

Mike Heath, Chair

6.1 Summary

This white paper summarizes discussions of the Mathematical Software Working Group at the Workshop on System Software and Tools for High Performance Computing Environments, held in Pasadena, California, April 14–16, 1992. Our goal was to identify the major issues in mathematical software that may influence the success of the Federal Government's High Performance Computing and Communications (HPCC) Program, and to provide a vision for the future development of the mathematical software component of HPCC environments. We did not expect to resolve any of these difficult issues immediately, and in this expectation we were not disappointed.

In the interactions among the various working groups, it became apparent that the new difficulties posed by solving Grand Challenge problems on massively parallel architectures require a major reassessment of each group's model of the others' needs. Mathematical software must be rethought in light of new applications, languages and architectures, and compilers and languages must be rethought in light of new applications, architectures and mathematical software needs. Supplying slightly improved but essentially similar versions of old tools will not solve our problems in the near term or the longer term.

Also apparent was the insufficient communication among the various groups to convey their real needs rapidly, and there was ample evidence

of serious misunderstandings of what others want and need. For example, the Mathematical Software group assumed that users want user-friendly software with complexity hidden as much as possible. This is embodied in the encapsulated "black box" modules supplied by conventional libraries and the higher-level abstractions and hierarchies that can be built on top of them. However, our sample users from the Applications working group emphatically expressed a preference for "nuts and bolts" templates they could modify manually to suit their needs.

Another example is provided by our interaction with the compiler group, which was surprised by several of the requests we made of them. They apparently expected much higher-level requests dealing with help in laying out data and expressing parallelism. Thus, the compiler community had been unaware of some basic needs of their user community (mathematical software developers), just as the mathematical software community was not fully aware of the real needs of the applications user community,

Thus, one of the main conclusions to be drawn from the discussions at the workshop is that there should be much greater vertical integration of HPCC software research and development. Collaborative teams are needed in which researchers in systems, languages, compilers, software tools, mathematical software, and applications end-users work directly together on Grand Challenge applications so that their results will be highly relevant and useful. Because the normal professional incentives of peer-reviewed publication and departmental appointments will continue to militate against such vertical integration, it will not be easy to build and hold together such teams. Appropriate incentives must be found to encourage computer scientists and mathematicians to get their hands dirty with real world problems, while at the same time recognizing that they must be given scope to exercise their professional skills and abilities. The results of such collaborative efforts must be communicated to as broad an audience as possible for the lessons learned to be propagated throughout the wider scientific community and the multiple disciplines involved.

Another conclusion is that there is an urgent need for near term results. Grand new schemes will undoubtedly be needed to solve some of the outstanding problems in high performance computing, but applications developers cannot wait for long term projects to pay off in useful products. They need immediate help in making the transition to massively parallel architectures. Thus, there should be a blend of projects having relatively modest objectives and promising near term results, along with grander designs that promise to reinvent the future, but may take several years to accomplish.

Finally, many of the suggestions we received from the applications users were really a plea for help in learning effective numerical and parallel programming at an advanced level. Education has not been a traditional goal of mathematical software, but we seem to have been singled out on this occasion as the most likely candidate to be able to provide such services. Although the mathematical software community certainly has a great deal to offer in this regard, clearly such educational activity must be pursued in a much broader context. Thus, it appears that educational initiatives in parallel computing should receive very high priority, and would likely have a greater immediate impact on the effectiveness and success of HPCC than any other software component of HPCC.

In the remainder of this white paper we give a more detailed report on the Mathematical Software working group's wide-ranging discussions, which for convenience have been organized into six categories:

1. Applications
2. Algorithms and Data Structures
3. User Interfaces
4. Portability and Scalability
5. Software Engineering
6. Enabling Technologies

These topics are listed above roughly in top-down logical order; no implications are intended as to their relative importance or urgency. We now summarize the discussions of each topic.

6.2 Applications

Since applications were the main topic of another working group at the workshop, the Mathematical Software working group made no attempt to discuss individual applications in detail. However, mathematical software development is ultimately driven by the needs of applications software development, and therefore we needed a critical reassessment of applications requirements. We were aided in this effort by a visit from the Applications working group, which led to a lively, and sometimes surprising, discussion of what users really want. It soon became apparent that "users" are not a

monolithic entity, but in fact represent a wide diversity of individual needs, from the sophisticated computational scientist who eagerly adapts to each new architecture in search of ever higher performance, to the relative computational novice who may approach change with reluctance and mistrust. Also, there is a wide range between users who want to see implementation details and those who prefer that such details remain hidden. Mathematical software developers must keep in mind this range of potential users, and realize it may not be possible to meet all of their needs simultaneously.

The strongest desire expressed by these users was simply to satisfy the urgent need to get applications codes running on parallel machines as quickly as possible. This pressing need leads such users to be wary of any grandiose plans by developers of systems software and tools. In contrast to the usual expectation that users will "ask for the moon," this group of users instead insisted that the goals of software developers should be sufficiently modest that useful results can be made available on a very short timetable, say within six months. They have simply seen too many grand designs that never bear fruit, at least not in time to be helpful in real applications of immediate interest.

These users also took issue with the entire conception of mathematical software as it is traditionally understood. Their claim was simply that conventional mathematical software has often not been a critical component of many real applications codes. For a variety of reasons, many such applications developers find they must write custom software for their mathematical subproblems. Reasons given included inadequate functionality in existing software libraries, inappropriate data structures that are not the most natural or convenient for a particular problem, and overly general software that sacrifices too much performance when applied to a special case. These considerations led to the suggestion that easily customizable software templates might be more useful for many purposes than the encapsulated "black box" routines usually supplied in conventional software libraries. It was also felt such templates could play a valuable educational role in teaching numerical techniques and parallel programming, and they would tend to enhance the understanding and confidence of users, compared to impenetrable "black box" software. We will discuss this concept further under the heading "User Interfaces" below.

This indictment of conventional mathematical software is somewhat unfair in that it is based on hindsight, and it ignores the state of hardware, and particularly software (languages, compilers, operating systems) technology available in the late 1960s and early 1970s when the mathematical software

industry initially took shape. Moreover, the mathematical software industry can point to many valuable successes: widely used public domain packages such as EISPACK and LINPACK; at least two viable commercial suppliers of general purpose mathematical software libraries plus many other vendors of more specialized software; and the fact that repositories such as Netlib ship out hundreds of megabytes of mathematical software per week to national and international users via the Internet. Nevertheless, even without a nudge from users, mathematical software specialists had already taken the transition to scalable parallel architectures as an occasion for completely rethinking the design and structure of mathematical software libraries; if indeed the term "library" is even appropriate in this new computing environment. For example, the Mathematical Software working group had already discussed the "reusable template" concept, along with several other potential new paradigms, before it was mentioned by the visitors from the Applications group.

A detailed discussion of the mathematical software requirements of specific HPCC applications is far beyond the scope of the working group discussions or of this report. Suffice it to say that user needs are both diverse and urgent, that we should consider anew the best framework for meeting those needs, and that genuine usefulness should be our paramount criterion in developing and evaluating mathematical software. Clearly, mathematical software should be broadly interpreted to include symbolic and statistical, as well as numerical, computation and also should include computational paradigms, such as n-body solvers, that arise frequently in many applications, but have not usually been within the scope of mathematical software libraries. Moreover, these diverse components and capabilities should be integrated to the greatest degree possible.

6.3 Algorithms and Data Structures

Parallel algorithms are obviously a prerequisite for parallel mathematical software. The development of parallel algorithms is currently an extremely active area of research by numerical analysts and others, and it warrants continued substantial effort and support. This is such an enormous topic, however, that we could not begin to scratch the surface in a day's discussion or in this report. The study of practical parallel algorithms is still in its infancy, with many difficult problems outstanding that are too numerous to list here. One of the few things one can say with confidence is that

parallel algorithms universally applicable to a broad range of problems and architectures are likely to be rare, and thus polyalgorithms will likely be even more important in the parallel world than they are for serial computation. An implication of this is that functions will have to provide performance estimates (of time, space, and accuracy) in order to guide the polyalgorithm.

Error analysis and computer arithmetic are important facets of numerical algorithm design. They have a direct and fundamental bearing on mathematical software development but have received too little attention to date in a parallel computing context. Parallel computation introduces a number of potentially thorny issues in these areas that are of obvious concern in the development of parallel algorithms and mathematical software. The effects of finite precision floating-point arithmetic on serial numerical algorithms have long been studied and are reasonably well understood. However, parallel counterparts of such algorithms may or may not enjoy the same properties with regard to accuracy, stability, and convergence. The most readily parallelizable algorithms are sometimes less stable than the best serial algorithms. Their speed makes such algorithms very attractive despite their possible inaccuracy or failure. This feature makes inexpensive error bounds or warnings of inaccuracy essential. For example, the fastest practical parallel algorithms currently known for the nonsymmetric eigenproblem fail a few percent of the time, much more frequently than their serial counterpart. Also, parallel prefix computations can be fast but over/underflow quite often in intermediate results even though the final result does not. This means that we need good floating-point exception handling capabilities. Exception handling needs to be carefully thought out in data parallel programming models as well. Even if we use algorithms that appear to be equivalent to serial ones, the lack of associativity of floating-point arithmetic may yield different rounding properties and different results, such as when computing a sum in parallel by a standard tree-like algorithm compared to conventional serial summation of consecutive terms. The order of operations may also be nondeterministic, yielding results that are not exactly repeatable or of easily assessed accuracy.

These considerations suggest that the presumption of "clean" standardized floating-point arithmetic, with powerful exception handling capabilities, on the individual processors (e.g., IEEE floating-point standard) is preponderantly more important for parallel mathematical software than it is in the serial world. Going beyond this, it may also be essential to establish new concepts, techniques, and standards for error analysis and computer arithmetic addressed specifically to parallel computation. In any case, results

produced by risky algorithms should always be accompanied by computed error bounds, and reliable alternative algorithms provided when necessary. Such error bounds must be obtainable cheaply or they will not likely be used.

Data structures is another vital topic in the design of scalable parallel libraries. We have already noted the difficulty of matching generic library data structures with user needs in specific applications, even for serial programs. This difficulty is substantially worse in a massively parallel environment, where an enormous variety of distributed data structures are potentially viable options. Our discussion of this topic focused on the desirability of formulating basic categories of abstract data types that would enable the implementation of parallel algorithms in terms of higher level objects that are independent of the details of the data structure implementation. Examples of such objects include

- matrices: dense, sparse, block-structured, etc.
- grids: regular, irregular, composite, multilevel, adaptive
- graphs: undirected, directed
- splines
- discretizations: interior, boundary
- time-step structures for the above
- sets (e.g., of particles)
- functions
- constraints

A great deal of intellectual economy, as well as software portability across different architectures and applications contexts, could be gained by standardizing and encapsulating such objects and the operations defined upon them.

Intimately connected with parallel algorithms and distributed data structures are the problems of how to partition the work of a computation into concurrent tasks, how to map those tasks and corresponding problem data onto multiple processors and memories, and how to schedule the resulting tasks for efficient concurrent execution. Principal responsibility for these problems does not properly lie in the software library development effort;

but they obviously affect such central library issues as problem representation, portability, and the user interface, so that progress in all of these areas will likely proceed in tandem. The close interdependence among these issues underlines the need for vertically integrated, interdisciplinary research efforts in which specialists in each area work together to obtain comprehensive solutions to the outstanding problems.

6.4 User Interfaces

As computer architectures and programming paradigms become increasingly complex, it becomes desirable to hide this complexity as much as possible from end users. The traditional user interface for large, general-purpose mathematical software libraries has been for users to write their own programs that call on library routines to solve specific subproblems that arise during the course of the computation. The pervasiveness of a single basic architectural paradigm (von Neumann), tightly coupled with a single standard programming language for scientific computing (Fortran 77), enabled a complete and reasonably concise description of the problem in terms of the parameters in a subroutine call. Adapted to a shared-memory parallel environment, this conventional interface still allows some ability to hide underlying complexity. For example, the LAPACK project incorporates parallelism in the level-3 BLAS (Basic Linear Algebra Subprograms), where it is not directly visible to the user. But, when going from a shared-memory paradigm to the more readily scalable distributed-memory paradigm, the complexity of the distributed data structures required is more difficult to hide from the user because the problem decomposition and data layout must be specified. Moreover, different phases of the user's problem may require transformations between different distributed data structures.

These deficiencies in the conventional user interface prompted extensive discussion of alternative paradigms for scalable parallel software libraries of the future. Possibilities suggested included:

1. traditional function library (i.e., minimum possible change to the status quo in going from serial to parallel environment)

2. reactive servers on a software bus communicating via byte streams

3. generic interactive environments like Matlab or Mathematica, perhaps with "expert" drivers (i.e., knowledge-based systems)

4. domain-specific problem solving environments, such as those for structural analysis

5. reusable templates (i.e., users adapt "source code" to their particular applications)

Options 1–4 are listed above in increasing order of their potential for hiding implementation complexity from the user. These four options are not incompatible, however, in that option 4 could be implemented on top of option 3, which in turn could sit on top of option 2, whose servers could in turn be based on modules from option 1. A user could interact with such a system at whatever level of detail and complexity was desired and appropriate for a particular application.

Little enthusiasm was expressed for option 1 as an end in itself, but it could nevertheless remain useful as a lower-level construct. Option 2 has a number of attractive features, including finessing the problem of interoperability among multiple languages, facilitating the integration of numeric, symbolic, statistical, and graphical modules, and fitting well into a networked, heterogeneous computing environment with various specialized hardware resources (or even the heterogeneous partitioning of a single homogeneous parallel machine). Option 3 and option 4 reflect the growing popularity of the many integrated packages based on these paradigms, and would provide an interactive, graphical interface for specifying and solving scientific problems, with both algorithms and data structures hidden from the user because the package itself is responsible for storing and retrieving the problem data in an efficient distributed manner. In a heterogeneous networked environment, such interfaces could provide seamless access to computational engines that would be invoked selectively for different parts of the user's computation according to which machine is most appropriate for a particular subproblem. Moreover, environments like Matlab and Mathematica have proven to be especially attractive for rapid prototyping of new algorithms and systems that may subsequently be implemented in a more customized manner for higher performance.

Option 5, reusable software templates, does not fit conveniently into the above hierarchy of paradigms 1–4. In fact, option 5 seems somewhat retrogressive in terms of hiding implementation complexity from users, since users would be directly involved in the details of adapting templates for their particular applications. On the other hand, templates offer the opportunity for whatever degree of customization the user may desire, and could

also potentially serve a valuable pedagogical role in teaching parallel programming and instilling a better understanding of the algorithms employed and results obtained. These factors accounted for some of the enthusiasm for templates evidenced by our sample users from the Applications working group, who asked for modules at the scale of about 1,000 lines of code. An even more significant factor was their belief that such templates could be made available on a relatively short time scale.

Although the Mathematical Software working group was sympathetic to the concerns of such users, we nevertheless noted some fundamental problems with the template approach in supplying mathematical software to users. First, there are delicate numerical details in many algorithms that are best left to experts. One simply cannot modify cavalierly the numerical details of some algorithms without seriously degrading, or even invalidating, the integrity of the numerical results. Could novice users be trusted to recognize which parts of the program are essential details and which parts are replaceable generic code? Experience with "cookbooks" like Numerical Recipes suggests perhaps not. Second, the template idea sounds seductively attractive until one asks just what level of detail is appropriate for such "prototype" programs. To be at all readable and understandable by other users, a program must be extraordinarily well structured and thoroughly documented, which would require a substantial polishing effort. Moreover, it is often difficult to find either the people or the funding to do such work. Indeed, one could argue that the messy details of Fortran code are simply not the right vehicle for pedagogical purposes. But any higher level of abstraction risks the same mistrust (and potential disuse) by users mentioned earlier. It is also worth noting that users sometimes express requirements that seem mutually contradictory, such as desiring powerful data structures and well structured codes, but insisting that it all be done in Fortran, which greatly inhibits such features. Thus, we expect a continuing need to explore a variety of approaches, with an emphasis on hierarchical systems with which users can interact at whatever level they choose.

Novel user interfaces that hide the complexity of scalable parallelism will require new concepts and mechanisms for specifying and representing scientific computational problems, and how those problems relate to each other when more than one is involved. New very-high-level languages and systems, perhaps graphically based, would not only facilitate the use of mathematical software from the user's point of view, but would also facilitate automating the determination of effective partitioning, mapping, granularity, data structures, etc. However, new concepts in problem specification and representa-

tion may also require new mathematical research on the analytic, algebraic, and topological properties of problems, such as the existence, uniqueness and sensitivity of solutions. The wide applicability of software depends not only on the traits of the software itself, but also on the mathematical generality of the underlying methods and algorithms. Moreover, new computational paradigms that emerge—such as neural networks, genetic algorithms, and cellular automata—require mathematical analysis for a full understanding of their fundamental properties and applicability. Whatever preferred interface, or interfaces, are eventually adopted, it should be emphasized that substantial efforts are already underway to promote software reuse, including Netlib, GAMS, and the HPCC Software Sharing Experiment, and these should continue to be encouraged and supported.

6.5 Portability and Scalability

It is unrealistic to expect users to adapt their codes to an ever-widening diversity of parallel architectures. Portability of programs has always been an important consideration, but it was easier to achieve when there was a single basic architectural paradigm (the serial von Neumann machine) and a single programming language for scientific programming (Fortran) that embodied that common model of computation. Such a uniform architectural and programming model facilitated both major facets of portability, namely the portability of program correctness and the portability of program performance. Architectural and linguistic diversity have made portability much more difficult, but no less important, to attain. Users simply will not want to make the investment required to create large-scale applications codes for each particular machine that may come along. By hiding machine-specific details, parallel software libraries can play a vital role in providing a large degree of portability for the applications codes that invoke them. Of course, this issue is intimately bound up with several other issues, such as parallel algorithms, data structures, and user interfaces. The concept of a problem solving environment seems to offer some hope for portability from the user's point of view, since new machines would simply be added to the network, where they would become new compute engines supplying cycles for that share of the load for which they were most appropriate. Of course, to supply such a seamless view to the user may require heroic efforts on the part of the developers of the underlying libraries and environment, especially if numerical consistency is required between processing resources in a hetero-

geneous system.

In addition to portability from the user's point of view, portability also is important from the mathematical software developer's point of view. Economy in development and maintenance of mathematical software demands that the effort that goes into a package be leveraged over as many different computer systems as possible. Given the great diversity of parallel architectures, this type of portability is attainable to only a limited degree, but machine dependencies can at least be isolated as much as possible. LAPACK is again an example of a mathematical software package whose highest-level components are portable, while machine dependencies are hidden in lower-level modules. Such a hierarchical approach is probably the closest one can come to software portability across diverse parallel architectures. It should also be noted that the high premium usually placed on portability is predicated in part on the high cost of software development and maintenance; new software development methodology that substantially reduced this cost might alter the relative value placed on portability.

General agreement existed in the Mathematical Software working group that the concept of scalability, however defined, is perhaps best viewed as simply a new facet of performance portability. In a sense, portability refers to running a single program on different machines, while scalability refers to running a single program on different instances of the same machine (possibly with vastly differing numbers of processors). Note, however, that as both the problem size and the machine size scale up, the physical model may change in ways that require different algorithms, such as the transition from two-dimensions to three-dimensions in a PDE model. Scalability in this sense is an extremely stringent requirement that clearly demands a deep understanding of the specific application in order to devise appropriate polyalgorithms to handle the full range of machine and problem sizes.

Portability always should be considered in terms of the effectiveness of a program, not simply whether the program will run at all on a different machine. Likewise, scalability demands that a program be reasonably effective over a wide range in number of processors. We did not try to go beyond this intuitive level and supply a precise definition of scalability, as this is the subject of numerous papers and some controversy. Moreover, scalability is clearly problem dependent. People buy ever larger computers to solve ever larger problems, but the manner in which a particular problem scales with the number of processors varies with the application and the purpose in solving it. Thus, while scalability is a very desirable trait, it is not easily defined or attained.

The scalability of parallel algorithms, and software libraries based on them, over a wide range of architectural designs and numbers of processors will likely require that the fundamental granularity of computation be adjustable to suit the particular circumstances in which the software may happen to execute. One plausible approach to this problem is block algorithms with adjustable block size, but in many cases it is clear that polyalgorithms may be required to deal with the full range of architectures and processor multiplicity likely to be available in the future. The situation is further complicated by sensitivity to context, even for the same type of problem on the same machine. For example, some very effective parallel algorithms have been developed for solving individual systems of linear equations on multiple processors. But, if an application calls for solving one hundred modest sized systems of equations on an architecture with one hundred processors, then the best approach is likely to assign a separate system to each processor, rather than spreading each system across all of the processors. Thus, it is important not only to have parallel algorithms available, but also to know when to use them.

Before significant effort is expended in developing mathematical software libraries for massively parallel machines, consideration should be given to what software would actually be useful for very large scale applications on massively parallel architectures. It is not clear that parallel counterparts of the traditional mathematical software library contents are necessarily the most appropriate. For example, many existing serial programs tend to be adapted to parallel machines at a very coarse level of granularity, so that many of their mathematical software needs are simply for serial routines to run on individual processors. Indeed, one significant use of parallel machines is simply to replicate many copies of a serial code across many processors in order to compute many individual cases simultaneously, thereby reducing the time to solve the overall problem. Attention also must be paid to how given applications scale as they become larger: do the relevant subproblems become larger, or do they simply become more numerous? For example, is there a real need to solve arbitrarily large, general dense systems of linear equations, or does enormous size usually imply some sort of additional structure that should be exploited?

6.6 Software Engineering

The internal design and structure of mathematical software libraries is another important research issue. Again, dealing with increasing complexity is the driving force. Even for conventional architectures and programming environments, large software development projects are notoriously prone to chaotic organization, missed deadlines, and buggy results. Given the added complexity implied by the requirements of scalable parallel systems, it is likely that this situation will be worse still for the exotic new computing environments of the future. Code reuse has long been one of the unique strengths of mathematical software; as larger and more sophisticated procedures are built, the amount of internal reuse can be expected to increase. Higher levels of rigor in development methodology, testing, and validation must be pursued.

Scalable parallel architectures of the future are likely to be based on a distributed memory architectural paradigm. In the longer term, progress in hardware development, operating systems, languages, compilers, and communications may make it possible for users to view such distributed architectures, without significant loss of efficiency, as having a shared memory with a global address space. For the near term, however, the distributed nature of the underlying hardware will continue to be visible at the programming level, and therefore efficient procedures for explicit communication will continue to be necessary. Given this fact, standards for basic message passing (send/receive), as well as higher-level communication constructs (global summation, broadcast, etc.) become essential to the development of scalable libraries that have any degree of portability. In addition to standardizing generic communication primitives, it also may be advantageous to establish standards for problem-specific constructs in commonly occurring areas such as linear algebra (e.g., the BLACS, Basic Linear Algebra Communication Subroutines). As it becomes more feasible to support a programming model with a global address space across distributed memories, new optimizations will come into play, including prefetching, cache management, and other aspects of dealing with memory hierarchies.

Rather than appearing as a monolithic entity, software libraries of the future will need to be very flexible in their potential uses. Some users will want to do little or no programming, turning over all details to the generic setting of the library, while other users will want to access individual modules from the library for use in a detailed code of their own devising that is tailored and tuned to a specific problem. An intelligent hierarchical structure

would seem to be the most appropriate approach to this issue. Another sense in which libraries must be flexible is in the interrelationships of the individual library modules. There must be a mechanism to allow various modules to be used interchangeably or "mixed and matched" within a given application when users need such capability. This is a stringent constraint on the algorithms and data structures involved, as their problem representations and data layout must either be compatible or easily transformable.

One trend in scalable parallel architectures that is already apparent is the emergence of hierarchical and heterogeneous systems. This trend is motivated by a number of technological factors, such as the sharing of memory or other resources by local clusters of processors and the introduction of special-purpose processors (e.g., systolic arrays) into a general-purpose computing environment. The problem solving environments we have envisioned might be most naturally supported by a heterogeneous collection of computational resources interconnected by a local- or wide-area network. Here again, however, architectural diversity and lack of standards greatly complicate matters. For example, the coordinated use of heterogeneous computational resources in solving a single problem requires that we deal with incompatible byte orders, word lengths, floating-point formats, etc., not to mention the logistical problems of compiling for multiple target architectures and coherently maintaining and updating the resulting executable modules. Such details will have to be handled transparently by the library/environment if users are to benefit from heterogeneous systems without excessive effort.

Another sense in which heterogeneity may arise is that a homogeneous parallel architecture may be best used in a heterogeneous manner. For example, a given application may be best implemented by allocating clusters of processors within a homogeneous architecture to various special purposes, with data flowing through the system from cluster to cluster as such services are needed. This point of view is quite consistent with the networked problem solving environments discussed earlier, except that now the various computational resources may be further subdivided and allocated in new ways. Such use would require new techniques for partitioning and mapping the computation, including the optimal allocation of computational resources and balancing the computational load among the processors.

Another area greatly complicated by the complexity of scalable parallelism is the monitoring and analysis of program behavior and performance. Yet, it is precisely in such a complex environment that an understanding of program behavior is most vital for both debugging and performance tuning. The automatic collection of performance data, as well as the provision of graph-

ical and statistical tools for comprehending it, will be an essential factor in ensuring that the library itself is efficient and that it is used effectively in solving applications problems. In general, instrumentation and performance monitoring should be implemented at a lower level than mathematical software libraries, specifically in the operating system, compilers, communication systems, and hardware. However, additional insight into performance can often be gained from higher-level, problem-dependent knowledge, and hence it may be desirable to instrument mathematical software libraries as well. Library modules also could be accompanied by operation counts and performance predictions for comparison with results actually obtained; and perhaps deviations from "expected" performance should be flagged to alert users to possible mismatches in algorithms or data structures.

Mathematical software that is intelligently designed and highly useful must take into account not only the machine environment but also the overall computational context in which it will function. For example, choices made in polyalgorithms may depend not only on issues like granularity and portability, but also on the location of the input and other such factors relating to the surrounding computation of which the library routine is only a part. A typical question is whether it is better to redistribute the input data to meet the assumptions of the mathematical software or to provide mathematical software that can deal effectively with a broad class of input distributions. The answers to these questions depend on the usual trade-offs such as performance vs. ease of use (i.e., machine efficiency vs. human efficiency).

Another issue that arose repeatedly in our discussions was the need to have library source code available, whether the user interface is in the form of reusable templates or of a more conventional variety. Compilers need source code for all modules in order to do interprocedural analysis on the whole program. Intelligent resource management, whether automated or manual, may require knowledge and control of internal algorithmic details that would be unavailable with an encapsulated, "black box" approach to mathematical software. Understanding and improving performance, and incorporating improvements from one domain to another, may also require access to source code. Users may want source code in order to customize it for a particular application or for educational purposes. Access to source code is no problem for public domain software, but it will require some rethinking and a new attitude by the commercial vendors of proprietary mathematical software. In general, a number of issues regarding intellectual property rights are raised by this new demand for providing source code, reusable templates, and other novel user interfaces.

6.7 Enabling Technologies

The developme it and use of mathematical software for high performance computing cannot take place in a vacuum. A substantial array of enabling technologies must be in place, or at least under development, in order to provide a meaningful context for mathematical software libraries. Such technological areas include architectures, algorithms, data structures, programming languages, compilers, operating systems, and software tools. Successful establishment of a rich environment for high performance computing will require concurrent progress in all of these areas. Because many of these areas were themselves topics for other working groups at the workshop, in our discussions we addressed only how these issues interact with mathematical software libraries and their development. Our discussions resulted in the following list of needs that mathematical software requires from compilers and/or operating systems:

- Provide correctly rounded conversions between decimal and binary.
- Allow the programmer to insist that the compiler respect parentheses in generating code for floating-point expressions. It should be possible for the user to turn off compiler optimization for portions of the code.
- Supply run-time environmental enquiries to:
  - discover properties of floating point arithmetic
  - determine how (with what precision) expressions are evaluated
  - determine how exceptions are handled
  - determine how distributed data types are laid out in memory

  Attempts by language and compiler people to handle the first three items have failed because the needs of the mathematical software community were not understood. It is the responsibility of the mathematical software community to write the specifications for these enquiries and then to collaborate with the compiler and language people to have them correctly implemented.
- Permit access to exception handling facilities, such as IEEE floating point sticky flags, provided by the architecture. Low-overhead exception handling should be provided, such as an OR flag of recent exceptions that can be queried cheaply at the end of a series of computations. Precise interrupts are too expensive to expect on heavily

pipelined architectures, so we need to determine appropriate alternative strategies.

- In order to build software tools, results of internal compiler analysis should be provided, including: symbol table, parse tree, type analysis, and mapping from source to optimized code.

- Parallel prefix for arbitrary user-defined associative operations should be supported. Conflicts between system and library (e.g., in message types) should be automatically avoided. There should be better support for mixed language programming.

Chapter 7

Languages and Compilers

Ken Kennedy, Chair
Marina Chen, Deputy Chair

7.1 Introduction

This report is a summary of the discussion by members of the working group on Languages and Compilers, for the Workshop on System Software and Tools for High Performance Computing Environments. The members include: Ken Kennedy (Chair) of Rice University, Marina Chen (Deputy Chair) of Yale University, Jeff Brown of LANL, Nick Carriero of Yale University, Mani Chandy of Caltech, John Dorband of NASA Goddard, Mark Furtney of Cray Research, Maya Gokhale of SRC, Jim McGraw of LLNL, David Mizell of Boeing, David Padua of CSRD, Bernardo Rodriguez of NOAA, Burton Smith of Tera Computer, Lauren Smith of DoD/SRC, Larry Snyder of University of Washington, and Thomas Sterling of USRA.

In this report, we identify key challenges in the realm of languages and compilers for high performance computing, and suggest strategies which will enable the HPCC effort to meet those challenges.

We start with an assessment of user needs in section 7.2. A rich base of ideas is being proposed in new languages and in new compilation techniques to exploit parallelism. We recognize that understanding various models and approaches and their interoperability is crucial in providing solutions or partial solutions to the user needs. These issues are elaborated in Section 7.3. We find obstacles, however, to the widespread dissemination of these ideas. The path from research prototype to *usable* prototype to commercial prod-

uct is hampered by a lack of generally available tools, standard tests, and evaluation standards. In Section 7.4 we suggest a prioritized plan to remedy this situation. We then discuss areas of research which should be emphasized in the five year time frame in Section 7.5. Our intent is that the research emphasis and technology development strategy proposed here will help to ensure long term success of the HPCC program as well as visible short term results in the two- to five-year time frame.

7.2 User Needs

Although parallel computing has been widely available for over half a decade, scientists and engineers are still reluctant to use it. A major reason is parallel machines lack software systems that would make them easy to use. The application developer wants to program in a *standard language* which is *portable* across a broad range of platforms. Compilers for these standard languages should deliver consistently *high performance*, so that the programmer can avoid consideration of low-level details of managing parallelism and the memory hierarchy unless it is absolutely necessary to achieve the desired level of performance. *Tools* such as intelligent editors, debuggers, and performance monitors and tuners should be available, along with access to *standard math libraries* through well-defined interfaces.

The user should not have the expectation that a sequential Fortran 77 program can be automatically transformed into a massively parallel one. However, the user does expect a program written to exploit massive parallelism to be *retargetable* to different and future generations of parallel machines with only minor adjustment.

Thus high performance delivered to the application requires a combination of (1) the user's expertise in recognizing the parallelism in the problem and in understanding how to map it onto a parallel computing model, (2) language expressivity, (3) compiler technology, and (4) debugging and performance tuning tools. Feedback and well-defined interfaces are needed among these components of the process.

7.3 Priorities

In this section we elaborate on the user needs identified in section 7.2. We begin by addressing issues related to the understanding and interoperability

of programming models. We then explore those issues regarding compiler technology and tools.

7.3.1 Models: Understanding and Interoperability

Broadly speaking, parallel computing can be classified into three programming models: (1) data parallel, (2) task parallel, and (3) object parallel.

In data parallelism concurrency is achieved with a single thread of control operating over elements of large distributed data structures such as arrays. Implementation of a data parallel program can be either SIMD or SPMD. In addition to entirely new data parallel languages, data parallel extensions exist for various dialects of Fortran, C, and Lisp. For example, Fortran 90, with operations over arrays and array sections, can be considered a data parallel language. An important issue in efficient implementation of data parallel algorithms is proper management of locality. For a data parallel program to run efficiently on large distributed memory SIMD and MIMD machines, data should reside in memories as close as possible to the processors that will need them. Data placement can be directed by the user through language extensions or directives, and enhanced by compiler optimization of interprocessor communication costs and the interaction of compiler and memory architecture in maintaining memory coherency across the processor array.

For task parallelism, a program is partitioned into cooperating tasks. These tasks can be quite different from one another, execute asynchronously, and use a variety of techniques for synchronizing with each other. Examples of task parallel languages are process-based languages: Linda, Schedule, Concurrent Logic Programming, Strand, PCN, and applicative languages such as SISAL. Many languages support some degree of both data and task parallelism. Locality is once again an issue in task parallelism on most architectures. Language extensions and directives can be used to distinguish local, private, or global data, and the compiler can play a role in managing uniprocessor memory hierarchies. As in data parallelism, the compiler and memory architecture can interact to maintain global memory coherency. Another issue in task parallelism is concurrency specification. Language constructs are necessary to allow the user to exploit fine-grain and coarse-grain parallelism, as well as levels in between.

The problems of locality and concurrency also affect parallel object-oriented programs. Examples of parallel object-oriented languages include concurrent C++, concurrent Smalltalk, and languages based on monitors and actors. Parallelism in object-oriented languages is achieved in different

ways: (a) a member function of one object can call a public member function of another (remote) object, and (b) an abstract data type, such as an array, can be implemented in a distributed manner.

This partitioning of programming models into three categories is incomplete and fuzzy; it is intended to serve only as a rough guideline. Furthermore, a finer categorization—for instance partitioning task parallelism into functional programming, communicating processes, etc.—has not been carried out in the interests of brevity.

Understanding the Choices. Our treatment of the various computing models in some ways mirrors the situation with parallel languages today: a number of good ideas have been proposed, and some implemented, but little or no effort has been made to provide an organizational framework which would help the user decide on the appropriate model for a particular application.[1] As a result, the typical user pays a high "mental" start up cost attempting to provide this structure for himself. Often one does a poor job, getting so lost in the details that one never arrives at a coherent picture of the whole. And not just the end users—new or even relatively seasoned researchers in the field could benefit from a better organized context to help place their current work and guide future explorations. Such a framework would form the foundation for a methodology for parallel programming and would begin to elucidate ways in which paradigms might profitably be integrated.

Interoperability of Models. For many applications, a combinations of paradigms will be the most effective to solve the problem. This could either be achieved in a single language (e.g., by including both data layout directives and task-oriented extensions in the same language) or by making it easier for the user to use different languages/models for different parts of an application. In fact, it is very desirable to standardize parameter passing data descriptors, and calling conventions, so that different languages can communicate. The various computing models also must deal with issues such as concurrency specification, parallel I/O and exception handling in a consistent fashion.

[1] In fact, it is likely that a combination of methods is required. We address this problem below.

Language Expressivity and Trade-offs. Programming language features must allow for portability across machines, must be expressive and yet must also expose concurrency and locality information to enable exploitation of the underlying machine's potential. These demands present a number of challenges. For example, how can users of these systems represent the parallelism inherent in nature in easy-to-grasp ways? What language constructs are necessary and sufficient to exploit fine-grain and coarse-grain parallelism, and those in between? Given the distributed and hierarchical nature of many hardware systems, what easy-to-understand and use mechanisms for specifying data layout for managing locality should be standardized and promoted? Some models also deal with non-determinism. One technical challenge is to integrate deterministic and nondeterministic constructs in such a way that those who want determinism can get it easily and cheaply. The additional expressivity of non-determinism comes at a price but can still payoff due to the nature of applications such as command and control systems. Such cost should be made apparent for those models that support such expressivity so that users can make intelligent trade-offs.

After stressing the importance of understanding parallel computing models and expressivity of languages, we now turn to the issues of compiler technology and parallel programming tools.

7.3.2 Compiler Technology and Tools

High performance and usability of massively parallel machines and other types of high performance platforms require compilers and tools that can help map the user's conceptualization of the problem to the target configuration easily and efficiently.

Given the concurrency and locality information specified in the program, the target code performance depends critically on how this information can be used to expose the structure of the problem (to match it with the most suitable target execution model based on static or dynamic profiling information) and to apply optimization strategies wherever possible.

One major technical issue in optimization involves the exploitation of *locality*. To obtain high performance and portability at the same time, *native compilation capability* is required. High performance obtained at the expense of usability cannot be a lasting solution and the balance between the two is a challenging issue. Tools that promote end-user productivity are needed to support true usability.

Exploit Locality. Because parallel machines are typically synonymous with complicated memory hierarchies, efficient programs must exploit data locality as much as possible. In this context, data locality means having the data reside in memories as close as possible to the processors that will need them. Over the past several years a great deal of progress has been made in compiler exploitation of data locality. We believe this will be a key area of research and development emphasis because the data locality problem will be so central to parallel processing.

Among the techniques that should be considered in this category are language extensions and the corresponding compiler techniques to enhance data locality, compiler management of uniprocessor memory hierarchies, compiler optimization of interprocessor communication costs, and the interaction of compiler and memory architecture in maintaining memory coherency across a processor array.

Native Compilation Capability. While translators can be used to achieve trivial portability (e.g., NAG's Fortran 90 to C translator), their lack in performance prevents them from being serious contenders to native compilers. Native compilation technology aiming at top performance, however, must deal with issues such as the complexity rising from the proliferation of hardware components as well as system configurations (e.g., memory system, operating system software, interconnection networks, etc.).

Exploitation of instruction-level parallelism is essential in attaining overall performance since the node processor performance is the basis for the performance of a massively parallel machine.

Balance between High Performance and Usability. Future compilers for high performance computing should be components of integrated programming environments designed to maximize end-user productivity and reduce time to solution.

Performance of the generated code is a primary goal of a high performance compiler. This is accomplished by applying increasingly complex program transformations and optimizations to the user source code. The resulting machine code can be difficult to map back to the original source, limiting the effectiveness of source-level software tools operating on the compiler-generated code, such as debuggers and performance analyzers. A net reduction in end-user productivity can result from a programming environment that emphasizes performance at the expense of debugging, performance

analysis and verifying program correctness.

Thus, a high performance compiler should provide detailed information about optimizations applied for possible use by source-level software tools through a standard interface mechanism. Ideally, this could be accomplished in a way that would not constrain the compiler and provide optimal performance along with a robust programming environment.

The debugger and performance analyzer would use this information to map the optimized program back to the source code while providing correct information about program state. Performance data should be fed back into the compiler to provide more effective optimizations in the next development cycle. The compiler can assist gathering performance data through instrumentation of the generated code, although this might be done in a less intrusive way by the debugger via dynamic instrumentation at run time.

In short, the effectiveness of future compilers for high performance computing should be measured by how well they balance performance with net end-user productivity.

Usability. In a more general context, actual *use* is a critical component in the evaluation of languages and compilers for parallel systems during the advanced development phase. *Usability*, almost tautologically, encourages (indeed, enables) use. So, by usability we mean not expressivity (important though that may be) or commercial quality software (desirable though that may be), but software mature enough that others can be reasonably expected to use it. Thus, it is a set of practical considerations that need to be addressed. These include:

- Clearly characterized domain of applicability, for instance, a list of unimplemented operations that, if used, cause the system to fail. Simple tools that scan for code inconsistent with the current domain would be desirable.
- *Responsive* processing time for those codes in the current domain of applicability.
- Trivial applicability. It should be possible to push a trivial code through the system and get a working output (even if the result is uninteresting). This lays the foundation for a comfortable introduction to the system and could be the starting point for a step-wise refinement approach to using the system.

- Stability. Fight the attitude that it is just research code, so it's okay if the system or the output routinely crashes.

Clearly, different levels of usability are required at different points in a project. Accordingly, software development needs to be designed from the start to be *staged*, i.e., to yield a series of usable (within designed constraints), well-defined intermediate systems. For example, early on in the development of a compiler, some subset of analysis or optimizations might be developed, while a language system might implement a subset of operations or use restricted semantics. As the development continues, and the complete compiler or language emerges, usability concerns shift to debuggers, performance tools, manuals, tutorials and making the system more widely available. However, to achieve usable tools, some initial thought and design to later incorporate such objects into a programming system needs to be done.

One valid strategy for making the transition from research prototype to production tools is to start with a basic product familiar to users and incrementally introduce advanced features on top of a familiar base. This strategy leads users into using advanced capabilities without the "culture shock," and provides a way to get early feedback of the usability of research ideas.

Finally, we believe that a strategy to address these priorities in a timely manner is critical to the success of HPCC over the next 2–5 years. We must choose which paths to follow based on the best information available, and start working on the solutions. Some of these topics are clearly research issues, but we must recognize the necessity of moving from research to development faster than we like.

7.4 Investment Strategies

In this section we recommend a number of HPCC investment strategies that we believe will have a substantial impact on the success of the program. We postpone the discussion of specific areas of research investment to the next section.

7.4.1 Technology Development Investment

Our primary strategic recommendation is that HPCC should make a major investment in technology development. Currently, we see three stages in the

development of a new language or compiler system from idea to product.

1. Research Prototype

   A research prototype is the typical result of a single-investigator basic research project. Such prototypes are typically used for concept validation and are not ready for use by real users because they are not robust or general enough.

2. Advanced Development Prototype

   To be useful, a typical research prototype needs to go through a development effort that will complete the implementation and make it ready for evaluation by real users. We call the result of such an effort, which is typically five to 10 times as expensive as development of the research prototype, an advanced development prototype. Examples of such prototypes are Berkeley Unix, Mach, X windows and the gnu C compiler.

3. Commercial Product

   The final stage in the technology development cycle is production of a commercial product, which will be robust, tuned and fully supported. This is typically done by a vendor.

In the United States we have good methods for supporting the development of research prototypes and the transition from advanced development prototype to product, but we have not developed a standard method for moving research prototypes to advanced development prototypes. We recommend that the HPCC program make a substantial investment in this stage of technology development.

We see several aspects of such a program. First is a need for a well-developed method for evaluating research prototypes and selecting the most promising ones for further development. We expect fewer than one in ten research projects would be so selected. The next subsection will develop this notion more fully.

Second, there needs to be an investment in infrastructure to be used in both research and technology development projects. Important parts of this infrastructure would be a shared software repository and a repository of test cases. These will be discussed in later subsections.

Finally, we believe the evaluation process should involve end users—scientists and engineers who will contribute ideas, criticism and meaningful applications.

7.4.2 Evaluation Standards

We believe the success of the HPCC Program will depend to a large extent on making the right choices for investment. Currently, the computer science community has not developed a strong and consistent way of evaluating the results of research in system building. Without evaluation standards, we will not be able to make intelligent investment choices. Therefore, we recommend that the HPCC Program foster development of such standards.

Several methods could be used to develop evaluation standards. First, we must construct a repository of test problems for investigation of parallel programming languages, compilers and paradigms. This repository will be discussed in a later section. Second, the agencies should make the use of test cases in project evaluation a requirement for funding. This would imply that sufficient resources be provided to carry such evaluations out. Finally, the agencies should require each project to contribute the code and experimental data (this might be thought of as the laboratory notebook) to the repository.

7.4.3 Collaborations with Users

A useful strategy for insuring that new programming systems are responsive to the needs of users is to fund projects which are collaborations between computer scientists and applications researchers. These collaborations could be in the form of efforts to exercise the results of language and compiler research in Grand Challenge implementation efforts. We believe that inclusion of users in an active way is essential if the HPCC evaluation process and technology development efforts are to result in useful products.

7.4.4 Infrastructure Support

Investments in infrastructure will result in an accelerated rate of progress in all areas of the HPCC research and development. Infrastructure can mean software resources or shared hardware systems. An example of hardware resources is a special parallel machine with extensive instrumentation hardware that could be used for non-invasive studies of performance. Of course, all projects will need access to some massively parallel computing resource. Two critical pieces of software infrastructure that will be essential to the success of the technology development efforts: a shared software repository and a repository of test cases.

7.4.5 Software Repository

The software research community needs a better method of software sharing. All too often valuable project time is lost in redeveloping standard components that have already been developed in other projects. Other times, effort is wasted on building standard software components that have no inherent research value but are required to build a useful prototype. Examples of the latter are scanners and parsers. We recommend that the HPCC program invest in the construction of a repository of useful software components.

We see two mechanisms that could be used to develop such a repository. First, standard components like scanners and parsers could be directly contracted for with software vendors. These vendors would be required, as part of a continuing contract, to maintain these components and integrate improvements and extensions that are developed by the research community.

The second method is through the technology development projects themselves. Enough funding should be included in such projects to permit the resulting software systems to be incorporated into the software repository and maintained. The project would thus control the integration of extensions to the system, whether developed inside or outside the project. We note that inclusion in the repository could nevertheless require a license for use. This is much like the mechanism used in Unix and the gnu tools.

The software repository would build on the technology development effort to insure that researchers can build on the efforts of others within the program. However, by concentrating on the technology development efforts, we can insure a degree of quality for the included software systems.

7.4.6 Test Case Repository

A successful software evaluation program must be built on a comprehensive library of test problems. Test problems need to range in size (in both execution time and code size) from small applications to full application systems, with different data set sizes also included. This would enable researchers to test their programming languages systems test at different ends of the spectrum, depending on where their project is in development, and what type of computing resources they have.

The same two strategies could be used to build this library as well. Initial versions of the repository could be built via direct contract. Subsequently, requirements for project funding should insure that the byproducts of each evaluation—rewritten code for the problems, parameters of the experiment

such as machine configuration and performance statistics—would be contributed to the repository. Thus researchers could compare their own results to the results from other languages and compilers.

7.4.7 Role of Standards

We believe standards efforts are an important tool for ensuring early capitalization of research efforts. For example, a new set of features to address a specific problem area in a particular language could be standardized to make sure the same features will be implemented by all vendors.

Other areas appropriate for standardization include standard procedure calling conventions and data descriptors, and standard interfaces for runtime environments and operating systems.

Massively parallel machines today differ wildly in their runtime environments and their homegrown methods in dealing with I/O and other operating system issues. Standard interfaces for such should be established, and viewed as an integral part of the end-user/language interfaces.

It is quite reasonable to invest HPCC resources to foster such standardization efforts for the results of technology development projects. However, care should be exercised to ensure that standards do not become strait-jackets that stifle creativity rather than foster it.

7.5 Areas for Research Emphasis

As part of our group discussion, we addressed the issue of identifying specific areas of funding within the category of Compilers and Languages that we perceived as having the most leverage for HPCC funding. Our recommendations for areas of emphasis can be summarized as follows:

- Build robust environments for successful, high performance languages/compilers
  - emphasize programmer productivity
  - continue performance enhancements in the compiler
  - extend performance analysis and debugging tools for user
  - expand information disclosure by compiler to tools and user
- Propel "qualified" emerging research languages into advanced development stage

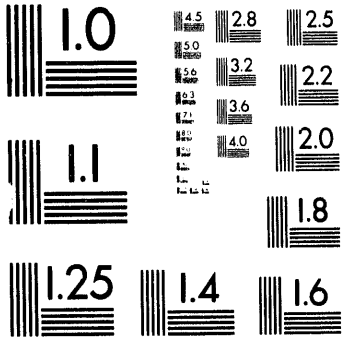
- support "non-research" activities: documentation, basic tools, robust implementations, conventional optimizations, user interfaces
- emphasize teams of compiler and applications personnel (also in strategy section of this document)

- Expand optimizations for critical compilers—focus on peak performance
  - Automatic techniques and Advice schemes
  - Key areas: partitioning, mapping, scheduling, locality management
- Investigate language features/compiler extensions that address areas of critical weaknesses
  - concurrency specification
  - data layout
  - parallel I/O
  - exception handling
- Continue funding of promising basic research efforts
- Explore schemes for paradigm evaluations and integration
  - head-to-head shoot-outs using a spectrum of applications, with an emphasis on algorithms having complex parallelism
  - evaluate all parallelism forms: data parallel, message passing, functional, shared address space, object-oriented, distributed data structures
  - encourage schemes for integrating different forms in one model

Note that we do not give a relative ranking among this list of candidates. Our inclusion of topics in this list is designed to maximize the likelihood of substantial impact on the outcome of the HPCC effort and also to best position the field of Higher Performance Computing for continued success beyond the specific five year timeframe. In the following, we describe these topics in two groups according to their primary nature as research or advanced development.

7.5.1 Basic Research

In the area of basic research, we begin with the comment that we do not yet know the *best* or even *generally-accepted* languages and compilers for


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0

2 of 3

parallel computing. As such, it is imperative that new ideas be fostered and developed to the point that they can effectively tested.

Approaches that create new paradigms need to be given priority for basic research funding. Likewise, efforts that explore strategies for integrating different paradigms into a coherent parallel computing model deserve careful attention. Given the current models, it is fairly clear that no one model is best for the broad spectrum of applications the HPCC Program needs to address. In funding studies of parallel paradigms, high priority should be given to those effort geared to detailed and thorough comparisons of different models (including data parallel, message passing, shared address space, object-oriented, functional, and distributed data structures). These efforts should provide objective results on the relative merits of each style on a significant cross section of applications requiring varying complexities of concurrency. Also, basic research that addresses the issue of *whole* solutions to Grand Challenge problems is important (e.g., methods that will allow integration of various models and paradigms using homogeneous or heterogeneous ensembles of machines).

Our group also recognized the need for funding to address several specific needs in terms of known critical weaknesses in the area of language design. Two well-known and yet still unresolved areas of work are the specification of concurrency and the specification of data layout. These two topics comprise the two most critical areas for high performance on almost all parallel machines. In addition, two lesser identified but critical topics for the future are parallel I/O and exception handling. For large complex applications, both of these areas must have far better solutions than we have at present or the process of developing new applications will be slowed substantially.

The one topic area that seems to span both basic research and advanced development is that of expanded optimizations for critical compilers. The goal here would be peak performance. We believe that in the next five years the state-of-the-art in both automatic techniques and advice techniques for partitioning, mapping, scheduling, and locality management is ready to show substantial gains. Such additions to heavily used compilers will enhance experimentation with massively parallel systems and dramatically improve both the quality and quantity of results on these machines. Technically solid proposals in this area of study need and deserve priority for funding.

7.5.2 Advanced Development

The key issue in advanced development is supporting the transition from good technical research ideas to production quality tools in heavy use by applications developers. Our point here is to identify the best candidates for moving to the next phase of development and providing adequate support to make that next step possible.

Successful and qualified research efforts in languages and compilers need time to develop a solid testbed and a willing applications effort to use and evaluate their potential. It has been our experience that people developing real applications need a level of tool development and sophistication that cannot be attained under research funding. Likewise, tools developed too far without substantial input from potential users will likely suffer major weaknesses. For those projects identified as the best candidates for advanced development funding (i.e., solid positive research results, potential for substantial use by applications, and sufficient technical expertise to make the next level of advances), we encourage substantial funding for teams of compiler/language personnel working closely with a challenging application effort. We would expect that funding would be needed to cover the application team as an integral part of the effort. In this stage of development, funding would cover many aspects of development that would be necessary to project success, but that would not generally constitute research. For example, developing solid documentation on how to use a system, adding conventional optimizations that are well known but need to be adapted for this system, building helpful user interfaces to better understand what the software is doing, consulting with users on how to use the system to best advantage, and implementing robust implementations of the software that will permit users to reliably evaluate the software. This kind of HPCC funding is expensive, but imperative to the long-term success of parallel computing.

In the case of successful, high performance software that has passed the above level of testing, the greatest need is for robust software that delivers peak performance of both the computer and the user. Where the previous described software needed basic tools, this software needs the most sophisticated tools available. At this point it would be most valuable to integrate the languages and compilers thoroughly into the best user-interface tools available. Detailed cooperation between the compiler and other tools (e.g., debuggers, performance analyzers, mathematical libraries, and operating systems) must be made to enhance maximum performance. The intent of funding for this work would be to emphasize programmer productivity,

continue performance enhancements in the compiler, extend the performance analysis and debugging tools available to users, and expand the amount of information to everyone about what kind of optimizations the compiler can and cannot perform. Funding for projects fitting this profile will probably have the greatest near-term impact on software for HPCC efforts.

In our group discussions, we recognized that although HPCC is a large new funding program, it includes a large spectrum of goals we must achieve. In trying to give our best recommendations on how to prioritize projects in languages and compilers for potential funding, a few principles became apparent. First, begin with a broad range of paradigms for parallel computing because we cannot authoritatively identify for you the best approach for the work supported by the HPCC Program. Second, use a combination of review by technical peers and foresighted users to select projects for advancement to higher expectations and higher funding levels. Third, recognize that system software development is a costly but critical component of high performance computing. Acceptance of new styles takes high quality software and experience in using it by applications people. Give it the necessary time. And finally, encourage tight collaborations among applications and systems teams by offering appropriate incentives. Recognize that the highest risk is incurred by those applications groups using the latest ideas in system software.

7.6 Summary and Conclusions

In this chapter, we identify important user needs in the area of languages and compilers, and priorities critical to the success of the HPCC Program. Motivated by the identified user needs, we elaborate on a set of priorities to be addressed. The key recommendation of this report is a set of investment strategies we believe will have a substantial impact on the success of the program. We pinpoint the weak link in the process of taking a bright idea to successful commercial product, i.e., moving research prototypes to advanced development prototypes, and recommend substantial investment in this technology development step. We believe a well developed evaluation method selecting promising research prototype for further development is critical. We also view investment in infrastructure support, software and test case repository, and collaboration with users to be necessary part of this strategy. Finally, we discuss areas for research emphasis, focusing on the next 2–5 years, both in basic research and advanced development.

Chapter 8

Software Tools

Joel Saltz, Chair
Frederica Darema, Deputy Chair

8.1 Debugging Tools

8.1.1 Tools to Trace the Origin of Known Errors

The feedback a user needs from a debugger depends heavily on the types of errors the user will encounter, and these are a function of the user's choice of programming paradigm and language constructs. Errors fall into two classes: determinate errors and indeterminate (or transient) errors. Determinate errors always arise on each execution of a program given a particular input data set. Such errors are fairly straightforward to isolate with traditional source-level debugging facilities. Transient errors that arise during executions of explicitly parallel programs are considerably more difficult to isolate. Transient errors typically arise through race conditions. When using a shared-memory paradigm, races include conflicting, unordered accesses by multiple processes to values in a shared data array; when using a message-passing paradigm, races arise through indeterminate matching of sends and receives. Buffer overflow in message-passing programs also is another common source of transient errors.

Single-threaded languages in which users specify data distributions (e.g., Fortran D, Vienna Fortran, and the emerging High Performance Fortran standard), prevent (or discourage) programmers from explicitly describing the details of how programs on each processor coordinate and share data. Because interacting processes are generated automatically by a compiler,

users do not have the freedom to introduce synchronization or message-passing errors. Although we expect a large fraction of scientific programmers will use single-threaded data distribution languages, systems programmers and many applications developers will continue to use other programming paradigms.

Debuggers need to be able to trace errors back to source code. This is likely to be a particularly challenging problem when a compiler carries out radical transformations during compilation. With the growing interest in High-Performance-Fortran-like languages, compilers performing aggressive transformations will become commonplace. To provide source-level debugging for such language models, it is clear that debuggers must have access to compilers' symbol tables and to extensive information about program transformations the compiler has performed.

A crucial unresolved issue is determining good ways to abstract program state. Program state abstractions should reflect the programming paradigm employed by the user. We might expect a debugger associated with a High Performance Fortran program to describe program state using the same address space seen by the programmer. Through the use of a debugger associated with a High Performance Fortran compiler, a user should be able to obtain values of globally indexed array elements. When programmers employ paradigms that support explicitly specified parallelism, a debugger will need to be able to represent the program state of individual processors. Because we can expect TeraFLOP multiprocessors to have thousands of processors, debugging tools will need methods making it possible for users to obtain information summarizing the state of a large number of processors. Programmers should be able to easily pose queries concerning the values of variables stored on specific groups of processors. In addition, a debugger for such machines may have to represent the state of a large interprocessor communications network.

Debuggers either need to automatically support isolation of transient errors (e.g., data races in shared-memory programs or buffer overflow in message-passing programs), or they need to supply users with information about sequences and timing of events, leaving the fault isolation to the users. Debugging support for automatically pinpointing causes of transient errors is complex. While significant progress has been made in this area, the asymptotic space and time requirements for using such techniques may be unacceptable for users pushing the limits of a parallel machine. In the absence of automatic support for detection of transient errors, perturbation of the program by the debugger can make isolation of transient errors difficult.

It is not clear at this point what degree of performance perturbation will be acceptable in practice. However, to minimize the impact of performance perturbation on debugging, virtual time strategies should be used for analyzing event orders whenever possible.

Among members of the software tools group, there was a consensus that funding agencies insist that vendors support rudimentary source-level debugging tools for parallel machines they offer. The lack of such basic tools for some commercially available parallel machines substantially impedes the progress of applications development on those machines.

8.1.2 Tools to Verify Correctness of Complex Codes

Current software tools used for establishing confidence in the correctness of applications in all possible situations are extremely inadequate—especially since the functionality provided has not scaled up with the added complexity of multidisciplinary applications and the added complexity of scalable parallel computers. Classical debuggers are very low-level and are helpful in fixing known specific problems; but generally they are not helpful in establishing confidence in the application as a whole. Software engineering methods need to be adapted to maintain test suites that can be used to verify program correctness.

8.2 Performance Tools

To determine what performance methodologies and what tools need to be developed, we need to identify the prospective users of performance methods and tools and to define the users' expectations and needs. We can identify the following categories of potential performance tool users: (1) end-users or application developers, (2) applications software package developers, (3) the system software (programming model, language or environment, and operating system) developers, and (4) hardware designers. Each user category has a set of specific requirements.

We make the following observations:

> Several major parallel architectural approaches and programming paradigms are represented in prototypes and products. Most current architectures are homogeneous systems consisting of many identical computational processors. Some of these architectures include hardware support for a (logically) shared memory paradigm, and others

support a (logically) distributed memory paradigm. The situation is complicated by the fact that virtually all vendors promise in the near future to support shared address spaces by some combination of hardware, operating system and compiler support. There are also examples of heterogeneous parallel architectures in which not all processors are identical and examples of compound parallel architectures in which multiple parallel architectures are interconnected. We need to know the extent to which we can support the same performance methodologies and tools for different architectures and programming paradigms.

Many factors influence performance of programs on parallel systems. The architecture of the base processor of the parallel system and the complex memory hierarchy (cache, local memory, disk) of the traditional uniprocessors is further augmented by the off-processor memory hierarchy in the parallel systems. This more complex hierarchy contributes to many differing time delays associated with accessing data spread among the various levels of the hierarchy. The system software (operating system, programming model, compilers, etc.) also affects performance. For instance, loop transformations and data partitionings carried out by compilers are designed to have substantial performance impacts. Latency hiding methods carried out by programmers, compilers and operating systems also are designed to impact performance.

8.2.1 Performance Measures

In discussing performance tool requirements, it is useful to first define the kinds of performance measures that various tool users might want. A number of performance measures are possible and users from the different categories we listed above might be interested in different (but probably overlapping) subsets. The following are a subset of potentially useful performance measures:

Total wall-clock (or elapsed) time to execute a given problem

Overall computation and communication time, profile of communication and computation time spent in various procedures

Breakdown of the time spent in computation and the time spent accessing each level of the memory hierarchy

Speedup or efficiency as the number of processors increases

The total CPU time, the MegaFLOPS and MIPS attained

A communication time breakdown consisting of hardware delays, and the overhead imposed by the system software (which varies depending on the methods used by the system software to manage the communication)

Contention at each level of the memory hierarchy

8.2.2 User's Requirements

Many users will want tools to help make decisions needed in performance tuning, or in studying the effects of incremental changes made to optimize the performance of a pre-existing code. Users want fast feedback—performance tuning should not require a lengthy and time consuming series of trial and error experiments.

Many performance measurements are likely to be made using a small prototype of a given highly parallel architecture. Much code development will probably be carried out on small local multiprocessors, and codes may be designed with next generation hardware in mind. The capability of methodologies to support predictions and extrapolations to larger and/or faster systems is consequently very important.

To a limited degree, hardware designers already use application-driven performance analysis to design machines. Vendors could use performance projections to predict consequences of architectural changes and as guidance to architectural choices. Therefore, from the hardware designers' perspective, there is a need to predict performance of code on new or scaled-up hardware.

8.2.3 Role of Compilers

Performance tools need to be closely coupled to compilers. To relate performance to user programs, performance tools may need access to extensive information about program transformations by a compiler. Information on code structure can potentially be used to make predictions on how performance of a program will scale with problem size or number of processors. Performance tools can also be fed back and used to allow users to determine interactively which strategies will be used to partition data or work.

8.2.4 Display of Information

Programmers are interested in performance measurements for individual sections of executed code, especially in cases where the total performance is below what the user might expect. Tools should provide the capability for the user to zero in on portions of code that exhibit poor low performance. Performance feedback then needs to be related to specific lines of code. There currently exist a small number of tools able to elucidate the relationship between performance and code text.

Methods for performance display have to be appropriate for machines with thousands of processors. The content and presentation of performance feedback should take into account the programming paradigm employed. For instance, in High Performance Fortran directives are used to control the distribution of data and work. Users will need performance predictions and feedback to ensure they are making good partitioning choices. Performance feedback and predictions associated with a High Performance Fortran programming environment need to be analyzed and presented in a way that provides insight on the relationship between performance and distribution decisions.

At this point, except for the kinds of general principles stated in the last two paragraphs, it is not clear how performance information should best be depicted. There was a consensus that funding agencies insist that vendors support obvious extensions of standard profiling tools. On many commercially available machines, performance profiling support is much more rudimentary than that offered on workstations.

8.2.5 Collecting Information

It is clear that the process of collecting performance information will itself have an effect on performance. Hardware instrumentation and monitoring subsystems can and have been designed to minimize the impact on the underlying executing applications and to improve the quality and quantity of performance information. Also, software instrumentation can be designed in a way that attempts to minimize the performance impact on the underlying application. Clearly, one would like to minimize the performance impact associated with gathering information. To the extent this can be done cheaply, it is clearly desirable. It is not clear that users are willing to pay a significant performance or monetary price for a non-intrusive performance monitoring capability.

8.3 Support for Shared Address Spaces

At the present time, shared address spaces in multiprocessors are currently supported by compilers, by the operating systems, by procedures explicitly invoked by users, and by hardware.

Distributed shared memory mechanisms support shared address spaces in different manners. On some architectures, such as the Kendall Square and Alliant, extensive architectural support is provided for address translation and data migration. Distributed shared memory on other architectures, such as the Intel or nCUBE, can only take advantage of much more rudimentary hardware support.

These mechanisms move fixed sized chunks (pages or subpages) of data between processors in response to the patterns of data referenced in each processor. Distributed shared memory mechanisms determine data layout dynamically, pages or subpages migrate to processors where data is read or written. Different distributed shared memory mechanisms are designed to take advantage of varying types of hardware support.

Compilers for languages like High Performance Fortran can generate patterns of explicit communication calls to transport data. The compiler allocates memory on each processor to store portions of distributed arrays. In High Performance Fortran type languages, data layout is specified by user directive. The compiler transforms loops so that the code on each processor will reference the appropriate locally stored distributed array elements.

In irregular problems, shared address spaces can be supported by tools that are invoked explicitly by users. Procedure calls are used to specify irregular (or regular) distributed array data layout. Runtime preprocessing procedures are employed to coordinate interprocessor data movement and to manage the storage of, and access to, copies of off-processor data. High Performance Fortran compilers are being extended to support irregular problems through the use of these kinds of procedures.

The compilers, distributed shared memory mechanisms, and tools outlined above show strengths in their ability to handle different types of problems. In the coming years, we anticipate the development of increasingly integrated strategies for supporting shared address spaces. This integration should reduce some of the overheads that result when these software systems are designed separately and integrated in an *ad-hoc* manner.

8.4 Additional Issues

The need is growing for robust, well-documented tools that can be used to transform source codes. The widespread availability of such transformation tools would greatly increase the productivity of researchers and vendors in their efforts to produce compilers, debuggers, performance estimation and monitoring tools. Such transformation tools can also play a role in the development of a variety of specialized tools. One example of such a tool is ADIFOR. This tool generates as an output a code that produces derivatives of variables calculated by input codes.

Chapter 9

Operating Systems

Bob Knighten, Chair

9.1 Introduction

This chapter presents issues and recommendations for the evolution of operating systems in support of high performance computing.

After a substantial discussion the Operating Systems working group reached a consensus that the traditional organization of operating systems does not lend itself to achieving the highest performance. This is the most controversial aspect of this chapter. As it also affects all other parts, the recommendation and the reasoning supporting it are presented in the next section.

The most requested feature from others at the conference was for checkpoint/restart and recovery features. We discuss these issues in Section 9.3 below.

A number of other topics were considered as well, and they are discussed below, each in its own section.

The general model we followed in our discussion and presentation was to try and address the following series of questions offered to the work groups by the program committee:

1. **What are the system software problems and needs** (from the application developer's point of view)?

   (a) What are the priorities of the users?

(b) How is the future likely to differ from the past?

2. **What is the status of systems software** (from the application developer's point of view)?

   (a) What feedback can you provide on the strengths and weaknesses of the software that already exists?

   (b) What is your forecast of results expected from software currently under development?

   (c) What is your outlook on research expectations?

3. **What are or should be the priorities for the future?**

4. **What is next? And for each possibility, what are**

   (a) The expected payoffs (to applications)?

   (b) The expected difficulties (computer science and technology?

   (c) The expected time frame?

9.2 Appropriate Division of Labor Among the Hardware, the Kernel, the Runtime, and the Application

The fundamental goal of a high performance computing system is to deliver high performance. The operating system must facilitate rather than impede this goal.

In a traditional operating system design, the kernel (the guts of the operating system) provides a great deal of functionality, while the runtime (which sits between the kernel and the application and consists of library code written by the operating system and/or compiler implementors) is a relatively "thin" layer. In particular, functions such as thread scheduling and inter-node communication within a single application are performed within the kernel, in addition to functions such as address space maintenance and processor allocation that transcend a particular application.

It is becoming increasingly clear that this organization does not lend itself to achieving the highest performance. Functions such as thread scheduling and inter-node communication can be handled by system-provided runtime

(library) code executing in the application address space, where these functions can be implemented with greater efficiency and where the knowledge resides to make appropriate decisions.

The efficiency gains in this organization arise from a variety of sources. "Common case" operations are supported without kernel intervention. This provides several performance benefits. The kernel is accessed by means of a trap and context switch, whereas the runtime library is accessed by means of a (much more efficient) procedure call. Further, functions implemented within the kernel must be fully general (supporting the union of the services required by all applications), whereas the services provided with the runtime can be tailored to the requirements of a particular language or a particular class of applications.

The role of the kernel in an organization such as this is to mediate between separate applications (for example, to partition the machine and/or perform processor allocation to applications) and to vector information to the runtime of an application, allowing it to make its own decisions (e.g., concerning thread scheduling). The kernel attempts to keep out of the way of a single application, yielding control to the runtime.

It is important to emphasize that this organization does not result in increased complexity in the high-level application code, or place any additional burden on the application programmer. Rather, it simply gives the compiler and runtime system more opportunity to optimize the use of system resources so they can provide a more efficient implementation of the user's program. The application and programmer interfaces don't change: 'ls' still works, as do the traditional library calls.

Nor is it the case that this organization necessarily compromises protection, provided that adequate hardware support is available. We consider inter-node communication as an example. The communication network hardware must support *message containment*—that is, messages from a given application must be restricted to destination node addresses assigned to that application. (Messages can be sent to other network addresses via the operating system kernel—the same way all messages are sent in the traditional operating system organization.) Also, communication registers that control the transmission and receipt of user messages must be directly accessible to the runtime layer. Given this support, inter-node communication within an application can be supported without kernel intervention with greatly increased efficiency, but protection is not compromised.

The operating system kernel should be written in such a way that relevant state information is vectored to the runtime system layer. This information

might concern page faults, exceptions, disk I/O events, and information necessary for the efficient implementation of checkpoint/restart by the runtime.

Moving support from the kernel to the runtime is an example of *delayed binding*: an application can employ a runtime that provides precisely the right functionality, and various programming paradigms, checkpoint/restart strategies, etc., can be supported efficiently.

9.3 Recoverability, Checkpoint/Restart, and Job Swapping

For many reasons recoverability mechanisms are important for both batch and interactive systems. Specifically, we see that long-running applications need to deal with:

- error conditions (both hardware and software generated)
- user requested checkpoint
- network disconnect situations
- intentional logouts
- dedicated system time interruptions and
- unexpected system crashes.

In these cases, the user code requires means by which the program can control a graceful abort that can be later restarted, or the means by which to continue execution independent of the disruption (such as network disconnects).

Recoverability mechanisms include the ability to reconnect an interactive session after disconnect, or conversely to hang up a disconnected session; and of course, a reliable means to create checkpoint files that can be restarted. It is important to note that such restartable checkpoint files should be migratable so they can be stored on archival storage systems and later retrieved and restarted. It is desirable for these files to be system independent, that is, restartable on similar hardware and software systems, independent of whichever system was the original environment.

We further see that some of these same mechanisms are required for debugging. Users need the means to "save" and "restore" an executable image

from within the debugger, and also to edit an executable image and then continue its execution.

In addition to the above, the kernel itself can use these mechanisms for efficient job swapping and resource management. It should be noted that we consider the image saved and restored by the kernel during job swapping to be a different object from the traditional restartable checkpoint file. However, there is no inherent reason why the kernel has to make this differentiation; some systems use only a single object for both functions.

It is important to the users that recoverability mechanisms be accessible from the user-level. It should be relatively easy for users (both from the keyboard and from within the executing program) to create a checkpoint file and/or to specify actions to be taken upon logout or network disconnect, etc.

Future systems should take these features into account when designing the hardware. In particular, checkpoint may be more efficiently implemented with hardware support. And certainly, distributed memory architectures require different checkpoint mechanisms than do shared-memory multiprocessors and vector supercomputers.

Today, there are no production-quality recoverability mechanisms for MPP systems. Most MPPs today do not have an efficient job swapping mechanism. Research prototypes are being developed, however. Some vector systems have comprehensive checkpoint/restart and reconnect capabilities, but they are based on dying technologies. Other vector systems based on new technologies have immature recoverability. They have restrictions on applications' checkpoint and do not provide migratable checkpoint images. Reconnect functionality is rarely implemented today.

The research community has a large body of literature on checkpoint for database systems, and distributed systems, but most research results do not apply to checkpoint for MPPs. The research checkpoint prototypes for MPPs are just emerging and they are implemented without adequate kernel support. They have restrictions on file accesses, interprocess communications, and resource sharing. In the next few years, we hope that production quality checkpoint mechanisms will be available.

Many research issues are relevant to recoverability:

- Reconnect capabilities
- Concurrent checkpoint
- Low latency mechanisms

- Migration of checkpoint files to other machines
- Saving checkpoints on archive systems
- Reducing checkpoint image sizes
- Compiler help to optimize checkpoint file organization
- Interfaces with high-level tools
- OS kernel support for checkpoint
- Hardware support for checkpoint
- Removing restrictions on checkpoint
- Checkpointing distributed programs on heterogeneous systems
- Active checkpoint vs. system swap images
- Standardization of checkpoint image contents

The items outlined above require algorithm studies and collaboration with OS kernel designers, architects, and compiler researchers.

9.4 Exception Handling

Support for efficient exception handling is important for high performance computation because some numeric routines generate floating point exceptions as part of their normal operation. These routines fall into two classes. The first consists of routines that do not need to check for or respond to individual exceptions, but simply need to know an exception has occurred (and if an exception has occurred, some of these routines are likely to have generated a large number of exceptions). This class of routines is best supported by a "sticky" bit that records exception occurrence (set by an exception, cleared by runtime software). The most reasonable place to implement this is directly in the hardware (in a user-accessible register), as this will be at least an order of magnitude faster than entering the operating system for each exception. Maintaining a separate "sticky" bit for each type of exception is a valuable aid to the after-the-fact analysis performed by these routines because the implications of different classes of exceptions (e.g., underflow, overflow, imprecision) vary based on the routine that generated them. For

example, many floating point routines can make use of imprecise results, but this is not the case if the floating point unit is being used to perform integer arithmetic.

Routines in the second, less frequent, class need to respond to individual exceptions, but usually not in a precise fashion; an exception usually causes a line of computation to be abandoned (instead of correcting the problem that caused the exception and continuing). Hardware support for imprecise exceptions is sufficient, provided compilers can generate code that produces precise exceptions (e.g., for debugging purposes). The Unix signal mechanism provides sufficient (even if inefficient) support for this class of routines. The existence of both classes of routines requires that hardware support the ability to mark (and change) types of exception events as signalling and non-signalling (IEEE terminology). In general the primary responsibility for servicing exceptions rests with applications and runtime systems, and operating systems should implement only those mechanisms necessary to notify these components of the occurrence of exceptions.

9.5 File Systems and I/O

We need to define some basic parallel I/O modes and standardize them, including their normal behaviors. It may be necessary to say that some properties are *undefined.* Suggested modes include node-synchronized and node-independent modes. Another candidate for standardization and common support on distributed memory systems is distributed file pointers, i.e., allowing the multiple parts of an SPMD program to have either shared or separate file pointers into a single file.

Standardization is also needed for user interfaces to I/O in a parallel environment at various levels. The POSIX P1003.4 working group is currently specifying, for example, the behavior of "printf" in a multithreaded program, but questions raised during the discussion session show that standards are needed at higher levels as well.

Buffered I/O modes tend to fall apart on parallel machines, unless the application programmer is conscious of buffering issues and addresses them directly. This is something which is typically not an issue on serial machines, so may come as a surprise to the programmer.

Different technologies may be applied to improve I/O system performance. Examples include RAID subsystems, Solid State Disks, and simpler techniques such as disk striping. However, the actual performance afforded by

these technologies will be system-dependent, because the system architect will be responsible for attaining that performance. Regardless, parallel I/O can be defined at a logical level and left to the vendor to implement. Applications can select the model that suits their structural and performance requirements.

Distributed file systems are clearly desired. However, I/O to networks and local disks is already a potential source of performance limits, so I/O to remote disks will surely be more so. This will require serious attention to get right. The Andrew File System offers some suggestions, as does the IEEE Mass Storage Reference Model.

Intermittent connections: how to deal with less than reliable services in a distributed processing environment. Long run times in a less than perfect world imply that loss of service is an eventuality. It may be desirable to start by defining fatal vs. non fatal interruptions. This is a current area of research in the context of distributed computing environments and network file systems.

Network I/O is seen by some users to be a problem, especially in the meta-computer environment, and for users of work stations acting as front-ends or data analysis platforms. Again, the OS can support logical parallel I/O modes, but it may be up to the vendor (or a resourceful application programmer) to make it work well for a given machine. Both bandwidth and latency are important.

File system security: local file system security is fairly well understood. Network file systems security is less so. These issues apply, but have been addressed elsewhere. DCE, ONC, and Kerberos are examples of this work. Unlike some operating systems issues, where the user/programmer wants control over the internal operation of things, file system security, once enabled by the system manager, must be not be something that can be disabled or circumvented by a clever programmer.

Hardware requirements: specific to a given hardware platform and typically handled by the vendor. For performance, the clever programmer may wish to explore I/O modes which map well to the underlying hardware, although one would hope these modes would be equally well supported.

Information required from higher levels: information from applications, runtime libraries, and/or the compiler regarding I/O access patterns and modes. It may be possible for the OS to schedule I/O operations more efficiently given good clues about the expected access modes and patterns.

9.5.1 Problems/Needs

User priorities:

- Faster I/O (file system and otherwise), is necessary to support current and future high performance CPU speeds. The dramatic speedups in computation afforded by high performance systems can turn a CPU-bound application into one that is bound by I/O performance or other communication performance limits.
- Parallel I/O is not well defined. For example, the question of the behavior of the "Hello World" on a parallel system was raised during the general discussion of operating systems. Even though this is a programming model issue, rather than an operating system issue, standards do need to be defined, even if the definition is *undefined.*

How is the future likely to be different from the past?

- The I/O supported by the system must scale to match the computational power. Some applications will apply the greater power to faster results, others may desire to process more data. It is not likely I/O capacities will keep pace with increasing processor power, leading to a growing gap. This will put greater pressure on file systems to be efficient, and not slow things down any more than necessary, given that the limitations of the underlying hardware will likely remain a bottleneck.
- Increased exchange of data between large machines and work stations will impose a burden on the smaller systems. Unfortunately, not much can be done from the large machine's perspective.
- Files will be larger than 2^{32} bytes, meaning that many (if not most) smaller machines will not be able to handle them. Support for files greater than this limit should be planned for, standardized, and extended to smaller systems as it becomes practical to do so.

Status:

- strengths/weaknesses
- forecast results of current work

- outlook on research expectations

Priorities:

- Faster I/O is crucial to the successful application of supercomputing technology to data-intensive processing tasks. It is also crucial to distributed computing issues.
- Disk throughput is fundamentally limited by the physics of disk drives. Solid State Disks are still very expensive with respect to the price of CPUs. Parallel disks and RAID systems may help.

9.6 Heterogeneity

Most of the issues related to supporting distributed applications in a heterogeneous environment fall within the domain of the runtime layer in keeping with the philosophy presented in Section 9.2. However, the operating system implementors do need to concern themselves with a few basic underlying issues. This section discusses these basic issues.

An application programmer who attempts to implement a distributed application spanning a heterogeneous network of machines faces several serious problems today. In a few important ones the operating system facilities or lack thereof play a significant part. These are:

- Lack of network and architecture transparent communication among programs executing on different machines with different environments

  In order to provide network and architecture transparent communication the issues that need to be addressed by the operating system community are:

  - Data representation and conversion
  - Management of remote execution
  - Uniform naming
  - Seamlessness, i.e., application does not have to know what sort of system it is talking to

  Each of these areas is addressed, to some extent, similarly but differently in DCE and ONC. DCE and ONC provide technology that

addresses the concerns of the HPCC community in the short term. There is a need for a single agreed upon set of programming interfaces that provides this functionality.

Considerable variation exists in how floating point numbers are represented in the various existing high performance systems, which makes it hard to provide a seamless interface among components of an application executing on machines that use different floating point representations. There is a need for a single agreed upon standard for representation of floating point. Currently, the only existing standard is the IEEE standard, and it is not used by all the important vendors in high performance computing.

- Data storage format (archival storage interaction)

  It is important to have the ability to use data independent of storage format. Data stored in natural format is easier to use and wastes less time in repeated conversion to and from formats that cannot be directly used in computation. However, this is an issue that concerns the Computing Environments group more than the Operating Systems group.

- Efficient communication among programs on single machine

  Distributable interfaces based on RPC schemes are not efficient when the two modules communicating through the interface happen to be colocated on the same machine. There needs to be a way to plug in different runtime systems, depending on the relative locations of the modules. Ideally, one would like to be able to use the location transparent interface but allow change in underlying implementation for speed for the local case.

Existing software and operating systems present certain areas of strengths and weaknesses in dealing with heterogeneous environments. Broadly speaking these are as follows:

- Ubiquitous availability of networking is an area of strength in current operating systems. Networking is now expected of operating systems. Base technology (e.g., TCP/IP, etc.) is available widely.

- Current networking technology that is widely deployed may not be adequate for efficient communication over future high-speed networks.

This is an area of weakness that needs to be addressed both in the academic and vendor communities.

- Heterogeneity support currently available is relatively weak, and there is much room and need for improvement. The weaknesses in this area have been discussed at some length in this section. While some technology is available in the form of DCE and ONC for hiding details of communication mechanisms from applications, they do not perform well when the correspondents are colocated. Floating point related issues are generally ignored. While seamlessness as an absolute requirement may be a myth, there is room for improvement in the area of reducing the visibility and roughness of the seams.

In the future, many more applications will span multiple machines of different architectures with different computation models (e.g., MPP and vector). Components of applications will be mapped to different machines based on matching the computational needs of the component and the capabilities of the machine. Application programmers will be able to do this using a relatively uniform programming interface for inter-module communication. This phenomenon will be made possible by the availability of industry-standard technology to support distributed heterogeneous systems.

In the short term, DCE, ONC and similar technologies will provide acceptable solutions to much of this problem. They will provide a reference model for the type of capabilities needed to form part of the infrastructure for heterogeneous systems. Eventually, a single programming interface will evolve through the application of persistent user demand upon the vendors.

For the longer term, it has been mentioned that the data representation conversion, etc., as done in both DCE and ONC are not fast enough for HPCC. More efficient RPC for high speed networks requires further research, but an evolution towards a single standard representation is unlikely to happen on its own. The chances of it happening are considerably enhanced if the user community persists in requiring it of their system vendors.

The following activities are recommended for ensuring that required support for heterogeneous environments is incorporated into operating systems in the near future:

- In the short term, DCE, ONC or similar technologies need to be broadly deployed to achieve heterogeneous interoperability. Broad availability of such technologies and experience in using them in real life applications also will help in flushing out a single interface. Some

work in that area is already taking place as a part of the OMG Common Object Request Broker (CORBA) effort. Such activities should be encouraged, and vendors should be encouraged to participate in such activities, in order to deliver a single interface paradigm in a timely manner. Developments in this area need to be closely monitored by the HPCC user community to ensure the performance needs of the HPCC community are met by the technologies and the vendors provide efficient and quality implementations of these technologies in products.

Considerable overlap exists between the operating system and the computing environment in this area. Issues that do not need direct operating system intervention should be handled by the computing environment.

Payoff: Heterogeneous interoperability.

Problem: Agreement on single interface and software integration

Time frame: Next 12 to 24 months.

Action: Users encourage vendors to deliver these technologies in the form of implementations of adequate performance and quality, and persuade them to converge to a single set of interfaces.

- Floating point format support needs to be added to DCE, ONC, etc.

  Payoff: Automatic conversion among several floating point formats in the process of doing an RPC.

  Problem: This is a standards issue, and it impacts more than just the high performance computing community.

  Time frame: 12 to 24 months.

  Action: Influence the vendors to add functionality to DCE, ONC, and to future interface technologies like OMG's.

- There is a need to identify and convert to a standard floating point representation adequate for HPCC for all future systems. If IEEE 754 is found to be adequate for the purposes for HPCC, then it could serve as this future standard to move to.

  Currently, IEEE 754 is a standard which is not adhered to by many vendors. Does it meet the requirements of the HPCC community? If it does, then the user community should encourage the vendor community to conform to IEEE 754. If not, then there is a need to understand

where it falls short and fund research and standardization activity to address those problem areas.

Payoff: Gets rid of floating-point conversion problem on new systems.

Problem: Some hardware vendors may not want to do this for perfectly good business reasons. In general this is a hardware issue.

Time frame: Could be forever. (Depends on vendors).

Action: Determine if IEEE 754 is adequate as standard floating-point representation. If so, encourage vendors to change to IEEE 754.

9.7 Memory Management

The memory management issue for high performance computing is undergoing a fundamental change. In the past, hardware designers have debated the wisdom and utility of providing virtual memory management support (i.e., a memory management unit) in high performance computing systems. There are examples of successful high performance systems with and without hardware support for virtual memory management. Hardware technology has changed to the point that the next generation of high performance systems will be massively parallel systems based on industry standard microprocessor chips. These chips include hardware support for virtual memory because the work station market (which consumes the bulk of these chips) requires that support. Thus, the area of memory management is changing from a hardware issue, to one of whether to provide support to a software issue and how to use the available hardware support. At the same time, the underlying memory system architecture is changing from a single pool of memory shared by all processors to distributed pools based on the distribution in an MPP architecture; this provides new challenges to the software that manages the memory resource. This section concerns itself with the memory management component of system software for MPP systems.

The high performance computing community harbors at least two unfortunate misconceptions about memory management hardware:

- It slows down the processor by increasing the cycle time.
- Using memory management hardware automatically means that the operating system will implement paging, which also costs performance.

The first assertion is no longer true for the industry standard microprocessors of interest; the fundamental cycle time does not depend on whether memory

management (i.e., translation) hardware is enabled. A second order effect is caused by misses on translation buffers (hardware cache of translation information) and the resulting refills, but the impact of this is very small for reasonable system designs. The second assertion represents a fundamental misunderstanding of the potential uses of memory management hardware.

Memory management hardware has found uses far beyond its original use to support applications whose address space exceeds the size of physical memory (i.e., paging). Among the uses made by operating systems in the commercial and research communities are:

- Support for shared memory programming models on machines and configurations without hardware shared memory
- Use of memory mapped files to improve I/O efficiency
- Operating system optimizations, such as copy-on-write
- Efficient support for the memory related features of programming languages, especially garbage collection
- Efficient support for memory-based update checkpoints (only differences from the previous checkpoint need be recorded)

For the purposes of this discussion, we distinguish these new uses of memory management hardware from paging by referring to them as *memory management* and *paging*. It should also be noted that the distributed memory architecture of MPPs provides new opportunities for use of paging techniques:

- Paging among nodes, rather than to and from disk.
- Extend address space beyond size of physical memory in a single node.

One of the consequences of these uses is that industry standards for operating systems are now requiring features that can only be implemented on hardware with memory management capabilities (e.g., mapped files, shared dynamically loadable libraries).

The use of standard microprocessors in MPP systems creates expectations in the area of software portability. Not only will users expect to be able to use the industry standard programming interfaces present on the corresponding work stations, but they also will expect the ability to freely move binaries that do not depend on MPP features between the MPP and corresponding

work stations. This is a major advantage for MPP systems in that it avoids the need to rewrite large pieces of software that are not performance critical (e.g., program development tools that are not specific to high performance applications), but it also creates corresponding requirements for the MPP to fully support the binary interfaces that such software expects. Needless to say, this includes memory management. For example, at least one work station vendor's program development tools rely on memory mapping for file access; MPPs based on this chip will have to provide mapped file support to run these tools effectively.

From the user's standpoint, the following priorities are identifiable:

- Efficient and effective use of MPP hardware, including both processor and memory resources
- The ability to run standard software and use standard programming interfaces, including those that depend on memory management
- The ability to run work station software binaries on an MPP when the work station and MPP use the same microprocessor
- Support for innovative uses of memory management in application domains and languages that can take advantage of it

System software for MPPs is still in a relatively early stage of development. Most MPPs are back end machines that require support from a host system. The memory management systems are primitive, and often place limits on application size and functionality. In some cases these limits are direct consequences of hardware design decisions (e.g., an architecture that requires that physical addresses be exposed to applications). Work is in progress at levels ranging from research to product development to improve the level of memory management functionality in MPP systems. This mixture of work is productive, as the problems being addressed range from research (e.g., effective support for shared memory programming on non-shared memory MPPs) to product development (e.g., mapped file support). In addition, having active projects at these levels makes it that much easier to transition technology from research through advanced development to products that are of use to the high performance computing community.

We believe that memory management functionality can be a valuable and important feature of MPP systems. Vendors should be encouraged to provide it. Shared memory across MPP nodes should be investigated as a programming paradigm even for machines with no or incomplete hardware shared

memory functionality. MPP operating systems should evolve to incorporate better management of the memory resources of the MPP; the multiple pools of distributed memory are more difficult to manage than a single pool of centralized memory. Hooks should be provided to applications so that runtime systems can provide information about memory usage and access patterns to help the operating system better optimize memory usage. This should include the ability to exercise suitable control over paging behavior, and prevent runtime page faults (as opposed to TLB misses) in high performance applications.

The benefits of making this functionality available include:

- portability and reusability of software among MPP and work station systems
- support for shared and extended (virtual to physical) memory programming models makes it easier to program some classes of applications
- improved utilization of MPP systems based on better resource management.

These benefits and others can be achieved in the next few years with a continued investment of resources in systems software development for MPP systems. The appropriate source of this funding varies with the maturity of the effort; product development efforts should be funded primarily by vendors, whereas government agencies clearly have a role to play in funding both research and advanced development.

9.8 Job Scheduling and Resource Management

9.8.1 Job Scheduling

Job scheduling is concerned with the scheduling of single jobs in a heterogeneous computing environment. It differs from CPU scheduling which schedules processes and threads for execution in a timesharing environment. Job scheduling deals with the relatively infrequent tasks of initiation, preemption and termination of jobs while CPU scheduling deals with a much finer time scale. Typically, CPU scheduling is implemented inside the OS kernel, while job scheduling is implemented outside the kernel, possibly as a daemon.

There are two levels of job scheduling that need to be implemented, a local scheduler and a global scheduler. It is assumed the global scheduler has hooks into the local scheduler to inquire about the availability of resources at the local host and to reserve resources on a local host.

It is assumed that the resource of the MPP (such as nodes) can be partitioned into local interactive jobs, local batch jobs and global batch jobs. The partitioning can vary from prime time to non-prime time. It is assumed that global interactive jobs are very difficult to schedule and so are not dealt with here.

The local scheduler needs to be able to properly schedule local batch and interactive jobs. It also needs to respond to queries and scheduling requests by the global scheduler. Some resources that can be scheduled are the following:

- Computing nodes
- Disk space (how much and how long)
- Priority of job
- Time limit
- Accounting group
- Minimum resource requirement (computing nodes only)
- Maximum resource request (computing nodes only)

The global scheduler must be distributed. It is assumed that the local scheduler is local to a single machine. The global scheduler may be hierarchically distributed. Each computing center is responsible for its own machines but may cooperate with other centers to run large applications.

If a job is scheduled to run, it first needs to check that all of the resources are actually available even though they have been reserved. Also, scheduled jobs need to be able to preempt running jobs.

Currently, the standard scheduling mechanism is NQS. NQS is a queuing system originally intended for scheduling batch work on single processors. It is not a true scheduler. At a minimum, it needs to be extended to do scheduling for an MPP machine and for global scheduling. This extension may not support the dynamic issue of interactive execution. It should at least recognize the resources that are reserved for interactive uses and refrain from allocating these resources to batch jobs.

9.8.2 Resource Management

Resource management in concerned with the management of compute nodes, disk space, network bandwidth, and time in a heterogeneous computing environment. Resource management on a distributed system is difficult. For accounting purposes, an accounting group ID is needed. The user group ID is for file access privileges and not very useful for accounting. A single user may be part of several accounting groups, one for each project. It is also anticipated that accounting will need to be collected locally and queried from other systems. A user should be able to specify his default account or an account during job submission.

Usage statistics are collected by system daemons.

Other Issues

Other issues of importance that are open research topics are graceful degradation of a distributed job, load balancing on a single MPP and in a distributed environment and dynamic resource allocation during job execution on a single MPP and in a distributed environment.

Who Does the Work?

The development of job scheduling and resource management facilities should be done by collaboration of vendors with universities.

This area is another example where a collaboration with compiler specialists could be useful since some information known to the compiler would be beneficial for job scheduling and resource management.

9.8.3 Hardware Support

Very minimal additional hardware support is needed. One possibility is an additional built-in instrumentation that may provide more timely data about current resource status for resource management and job scheduling, especially in heterogeneous systems. (Part of this issue is one of standardization on the representation of information needed for management of heterogeneous systems.)

9.8.4 User Support

The user needs to specify resource requirements and accounting group when submitting a job.

9.9 Message Passing

Massively parallel systems fundamentally depend on inter-node communication ("message passing"). Managing this message passing is a basic operating system task on such a system, though, as explained in Section 9.2, one that should be done outside of the kernel.

On many massively parallel systems, message passing is an explicit interface. In particular the send/receive model is extremely common. One of the tasks which has just begun and which should be completed within the next year or so is standardization of such send/receive interfaces for interprocessor communication.

Although the understanding of inter-node communication is sufficiently advanced that it is wise to standardize the send/receive model, the best model is not yet clear and support for development and experimentation with alternative models jointly by academia and vendors is very important.

Chapter 10

Computing Environments

Reagan Moore, Chair

10.1 Introduction

The computing environment topics include data storage, communications, and programming support environments. Applications are inherently limited by the capabilities and access the computing environment provides. One way to analyze the application requirements for high performance computing is to examine the associated requirements for the computing environment software infrastructure.

The hardware computing environment provided by vendors consists of heterogeneous computing platforms with associated data storage devices, linked by communication channels. The computing environment needed by application developers is support for rapid porting or prototyping of software codes that can make efficient use of the vendor supplied hardware. The software infrastructure that normally provides the interface between the application developers and the computer hardware is seriously lagging hardware advances. Difficulties also exist in providing system software for heterogeneous platforms since historically system software is developed by vendors who optimize the software for only their own platform. This has become further exacerbated by bottlenecks in the development of application software because of associated paradigm shifts that are now occurring. They comprise:

1. Development of massively parallel computers.

2. Integration of either homogeneous or heterogeneous computing platforms into metacomputers. (The metacomputer is a uniform interface to disparate computing platforms, providing the application programmer the opportunity to decompose an application across multiple machines.)

3. Integration of multiple computer centers into a metacenter. (The metacenter provides the same capabilities as the metacomputer, but across geographically remote computer centers.)

The needs of the application developer in this rapidly changing environment are dominated by the difficulty in programming parallel computers. For the next five years, the computing environment systems software requirements can be categorized into immediate goals, and three-year, four-year, and five-year goals:

Immediate goals

- Parallel Computer Support Environment (distributed memory)
- Standards for Parallel Computing Support Environment, including support tools
- Standard for parallel I/O
- Standard for message passing on distributed memory machines

One to three-year goal—Metacomputer

- Integration of multiple homogeneous computing platforms (cluster of workstations) to dynamically support multiple jobs distributed across the cluster
- Uniform environment for file formats and job checkpoints to allow a job to be restarted on another platform

Four-year goal—Heterogeneous metacomputer

- Integration of multiple heterogeneous computing platforms to dynamically support multiple jobs

Five-year goal—Metacenter

- Integration of multiple geographically distributed metacomputers into a national machine room

The computing environment software must provide the support needed to create a balanced system that masks or moderates throughput bottlenecks. This support includes the resource management to control diverse computer architectures (e.g., traditional vector supercomputers, massively parallel processors, clusters of homogeneous compute platforms, and combinations of these platforms). Future throughput bottlenecks may be driven by memory constraints, disk storage constraints, I/O channel rate constraints, or even data locality constraints. It is expected that jobs will be executed that use a sizable fraction of any combination of these resources, without necessarily using all of the available CPU power. Job scheduling will be necessary to maximize utilization of the resources.

10.2 Objectives

One approach to understanding the High Performance Computing technology requirements is to use the Grand Challenges as case studies rather than as end goals. This ensures the required systems software will be available to support particular Grand Challenge applications, while allowing development of a uniform systems software computing environment.

The system software problems currently confronting application developers can be summarized by six areas:

1. Provision of a uniform programming environment allowing portability of codes across heterogeneous environments
2. Program decomposition support for optimizing hardware resource selection and utilization in homogeneous and heterogeneous computing environments
3. I/O support for moving large files or storing massive amounts of data in real time
4. Development of scientific database interfaces for supporting access to massive data sets
5. Development of a national file system with uniform file names and uniform access to allow users to move program execution between computer centers and promote collaboration among scientific teams

6. Resource management infrastructure allowing execution of programs in an heterogeneous environment

Solution requirements to these problems are driven by application developers and by operating system development.

10.3 Applications Requirements

The requirements for the computing environment software are most strongly driven by the needs of the application developers, but also are driven by system managers, service providers, and service developers who are attempting to provide a stable environment. The most urgent need is to be able to use parallel computers efficiently. This implies the ability to write new applications and the ability to port applications.

10.3.1 To Write New Code: A Familiar Environment

The computing environment for parallel computers should be familiar, supporting the same UNIX tools as currently provided on supercomputers (make, dbx, ls, ps, etc.). The associated systems software needed for accounting, resource management, and scheduling should support multiple users of the parallel computers, as is currently done on vector supercomputers. A system that works and is in place is much better than prototype research software systems. A strong concern was expressed that it is better to support the simple computer environment utilities, make them work, and only then move on to the more challenging end goals such as the metacomputer and the metacenter.

Providing efficient UNIX tools, however, is dependent on research and development of new technology, such as parallel I/O. What is a familiar I/O interface? When the user runs a simple command on an MPP, what actually happens? Does a 'printf' statement print n times when the job runs on n nodes? A standard for parallel I/O is needed. Vendors are independently creating this technology for parallel computers. Coordination of the effort to make the MPP programming environment look like that of a workstation may need to be done under government grants. This is a hard problem, and may become more difficult unless a standards effort is started now. One hope is that supercomputing centers can have a big impact on the development of these standards by providing common, uniform environments. An informal collection and distribution of a set of current and/or recommended practices

could be useful in the long run as a way to promote the creation of a standard environment.

10.3.2 To Port Code: Standards

Users want tools for porting strategies. A need is evident for a standard, or set of standard parallel programming support environments (at the level of Express, PVM, ISIS, etc.). For message-passing architectures, the current programming support environments need a standard message-passing library to facilitate porting to similar hardware architectures. The standards should include some cost/benefit analyses and trade-offs so that the user can make reasonable choices between the programming support environments.

10.4 Application Data Requirements

Even with a standard environment, applications may generate more data on TeraFLOP computers than disks can hold. TeraFLOP computers will produce proportionally more data than GigaFLOP computers of today, requiring an equivalent jump in storage technology and availability. This may mean a 1000-fold increase in storage need, something not practical with today's disk technology. There is no foreseeable limit to the amount of data that may be manipulated. It is not at all clear how to scale up the I/O requirements from today's machines to the TeraFLOP environments. Efforts to predict future data support requirements are currently too simplistic. A major difficulty is that future architectures will probably not comprise the same environment as we are working in today.

What are the data storage constraints? Previously, the major constraint was memory size, so mechanisms were developed to page or swap jobs to disk. In the future, it appears the major constraint may be disk size. Will scientists be able to live within memory and storage constraints by changing their algorithms to be more efficient in terms of how much data needs to be saved?

One possibility for expanding the amount of total storage is the use of archive storage systems to integrate local supercomputer disks and archive storage devices. Such attempts must handle the maximum future working file set size. The working file set size, however, may be limited by the local disk size because it may not be an advantage to manipulate data larger than this. On the other hand, observational data sets will be generated in the future that will be arbitrarily large. Creating subsets of such data that can

be manipulated efficiently will require major advances in scientific database technology. The maximum acceptable working file set is effectively governed by the rate at which it can be archived or restored. Users need to be able to access their working file sets on time scales of at most minutes. This implies that the maximum amount of data a user can manipulate effectively should be on the order of (100 seconds) x (total I/O channel rates in bytes/second). Alternatively, users will need to access their data through caching file systems such as the Andrew File System. The effective use of such a caching system for manipulating large amounts of data will still be highly dependent on the communication channel rate.

The data storage environment also must be significantly enhanced to handle future memory sizes. Observations of current applications suggest that roughly a gigabyte of memory should be provided per GigaFLOP of CPU performance. For a TeraFLOP computer, current applications would then use a terabyte of memory. One model for predicting memory size is to assume that the algorithm complexity measured in operations per bit of memory determines the maximum amount of memory to associate with a CPU for a given CPU execution time. Assuming the algorithm complexity remains invariant, larger memory jobs may not be computed fast enough on TeraFLOP computers and therefore may not be run. This implies jobs may scale up by the same factor in CPU time and memory. Extensions to larger relative memory sizes may then not be needed.

Alternatively, if massively parallel processors are made up of current technology components, such as workstation processors, then it may be possible to apply current technology to data storage and I/O requirements. If both are scaled in a parallel fashion (a disk for each processor, etc.) and the problem complexity becomes greater as problem size grows, then one could predict that these very large problems running on MPPs will require less I/O performance than today's less complex computations running on workstations. These may be naive models for predicting future needs... but are other models less naive?

An associated aspect of data storage requirements is network performance. A predictable performance is more desirable than great performance one day and substantially less on another. The point is a balanced system. The need in terms of capacity or performance will be dictated by the performance and capacity of the other components in the system (disk, network, tape, memory, CPU). The appropriate scheduling of resources on TeraFLOP computers is a major design consideration for the computing environment software.

Future systems must be able to manipulate massive amounts of data. If

a certain error rate is introduced by networks, channels, storage, etc., what effect will this have on the application? Future archive storage devices will have non-zero error rates. Applications assume data are reliable and errors are unacceptable. The computing environment currently guarantees to the application that the data are reliable. These guarantees have mainly been implemented in hardware. An example is the use of SECDED (single bit error correcting, double bit error detecting) in CRAY memory. This is a hardware solution: 72 bits are stored for each memory word with eight bits used for error detection/correction. A similar technique may need to be used with archive storage devices. Detection may be good enough as long as the application is checkpointed and the results are partitioned so the error can be isolated. Otherwise fault-tolerant applications will be needed.

10.5 Application CPU Requirements

Heterogeneous distributed systems are interesting for some problems, but probably very few problems can be automatically distributed by a programming support environment. Initially, the user will need explicit control. Before automated tools are created to handle heterogeneous environments, homogeneous environments must first be understood.

Massively parallel processors constitute the simplest possible homogeneous environment. However, even for MPPs, it is not yet clear what level of kernel functionality needs to be on each MPP node. Current support ranges from a minimum of basic message passing to a complete implementation of the full UNIX environment. This is a cost trade-off question. The majority of data parallel codes would not work well with high operating system latency levels. But if there were no latency cost, the more functionality supported on the node by the operating system, the easier the application developer's job would be. Often the trade-off is in the amount of memory on each node dedicated to the operating system—some message-passing kernels cost on the order of 30% of the memory on each node, a high cost!

Part of the functionality supported by the operating system is the partitioning of resources for multi-user access. Amdahl's law predicts the maximum number of nodes an application should use to achieve a given CPU node utilization for a given parallelization level. High utilization may be easier to achieve by executing multiple applications on the parallel processor than by highly parallelizing a single program. This is another trade off question. Should the computing environment sacrifice CPU power and memory

to support multiple users rather than provide the optimal environment for a single application which may not run as efficiently? An alternate way to view the choice is whether utilization should be maximized for the entire multi-user workload or for a single application? Application development research efforts have been primarily evaluating the computing environment as a system on which to execute a single computation, and have been having great difficulty in achieving the required parallelization level. Heterogeneous computing systems impact this choice. They provide a distributed environment in which to work with the computation processed on some of the platforms, storage provided elsewhere, visualization and control provided yet elsewhere. The heterogeneous environment may be predominantly a multi-user environment because coordination of dedicated access for a single application may be very difficult to achieve.

Is there a viable metacomputer for optimization of single applications? Perhaps the metacomputer is too difficult to optimize to be useful to the applications scientist in the next two years. Is there an ideal application working environment to migrate and run applications on a metacomputer? One possibility is to start with a homogeneous computing environment of multiple similar compute platforms.

10.6 Operating System Requirements

The computing environment software also is dependent on requirements from the operating system. These are driven by the need to select the appropriate software level for implementing new functions. The operating system also imposes a significant data-handling load on the computing environment.

The operating system functionality can be divided into conceptual layers consisting of the kernel and run-time support environments. The kernel takes care of a single compute server and its local resources, while the run-time environment supports other network-connected resources such as storage devices, other hosts, thread management, and resource management. Applications access the run time environment for all required support. The computing environment software resides mainly at the run-time level.

Operating system requirements for data support include file system caching and network I/O caching. Both need support for I/O striping to improve bandwidth. For parallel computers, support for parallel I/O is becoming increasingly important, with disks possibly being dedicated to subsets of processors. Standardization of file formats and data formats is needed

to simplify archival storage interface and parallel I/O support.

In addition, the operating system may severely stress the data systems through checkpointing of job images and through collection of performance statistics. Both of these system support functions can generate as much data as executing applications. The issue of checkpointing is strongly governed by the cost trade-off between saving an entire execution state, saving only a portion of the execution state needed for an application to restart, or saving none of the execution state and reexecuting from the beginning of the run. The choice must be made between local use of disk space and use of additional CPU execution time. Only the application researcher has the knowledge required to make the decision.

10.7 Software Technology Development Area

The computing environment system software that addresses these issues can be sorted into data support systems, communications support systems, and heterogeneous computing environments. Each of these areas has critical elements, some of which are currently being explored through research efforts. Important technology development efforts in each of these areas are outlined in the following sections, with an attempt to identify the organizations providing the driving technology push and the time scale needed for the systems software development.

10.7.1 Data Support Systems

Key elements in the technology associated with data support systems are I/O scaling (including data distribution), archival storage (including network attached peripherals, third party data transfer, and caching file systems), a national file system, scientific database interfaces, data format standardization, data manipulation tools, and data privacy.

I/O Scaling. I/O scaling of computer center data storage requirements to TeraFLOP compute platforms is needed to understand how to construct balanced systems. If the systems are poorly designed, bottlenecks in either provision of disk space or sustainable communication rate to archival storage could limit the performance of the center.

Estimates for how data storage/rates scale to TeraFLOP computers are highly application dependent. For example, it is expected that chemistry, computational fluid dynamics, and lattice gauge theory will have quite different scalings as a function of CPU power for memory and disk space re-

quirements. It may be possible to define categories of general computers for which these applications can be analyzed for current resource usage, and then develop scaling requirements across the categories. Additional analyses defining the major types of applications are also needed to determine the number of different application classes for which scaling should be done.

Analysis of the performance of the overall data storage system should include effects associated with data distribution. Data movement in current parallel computers limits program efficiencies because of relatively poor communication bandwidth compared to CPU execution power. To minimize data movement between different sets of distributed memory, current parallelization efforts attempt to localize data to the distributed memory associated with the compute CPU. This restricts the variety of applications that can be successfully ported to parallel computers. Architectures which allow decoupling of the data space from the compute CPU space will be able to execute a more general mix of applications.

A similar approach is needed to understand future I/O scaling. Data movement may occur not only between segments of distributed memory and the executing CPU, but also between memory and disk and between disk and archival storage. A data flow analysis is needed to understand the degree to which data space can be decoupled from the CPU space. Data access can also be viewed from the perspective of whether it is better to move the process to the data or whether it is faster to move the data to a supercomputer. For a sufficiently complex algorithm, the total execution time for a single process can be minimized by moving the data to a supercomputer. This may not maximize the overall job throughput since some compute resources will end up being used to manage the data movement.

Archival Storage. A significant advance in data support systems has been the creation of the IEEE Mass Storage Systems Reference Model. It provides a conceptual architecture and a set of common terminology. This effort has been vigorously supported by commercial software vendors, but it is not yet certain whether these distributed systems will be able to support TeraFLOP computers.

At issue is the expected performance of storage systems based on the IEEE model. Archival storage systems must support not only multiple users with a sustainable aggregate I/O rate, but also the expectations of retrieval time for individual files. If very large files are being moved, user expectations of short retrieval times can force a requirement for much higher communication rates than needed to support the average communication load. Data access in archival storage systems in the future must support a wide range of methods,

such as sequential access or random access, for various types of applications.

Archival storage performance can be enhanced by the use of third party file transfer which avoids forcing the data flow through the archival storage server's CPU. Software development efforts are currently being done on the support of network attached peripherals and caching file systems. The need for third party transfer from network attached peripherals is currently driven by the difficulty of supporting 100 megabytes/sec communication rates using the TCP/IP protocol. Competing technologies include processing the TCP/IP protocol in hardware or managing the data flow with higher speed protocols. The development of powerful microprocessors, provided I/O bus and memory speeds increase proportionally, may make it no longer an issue whether data is moved through a controlling CPU, if the controlling CPU is substantially faster than the other computational resources.

Archival storage systems at present support the migration of data through a hierarchy of caches from the supercomputer local disk to archive tape storage devices. Software development efforts are being done to integrate caching file systems to allow the transparent movement of data from the archival storage system back to the supercomputer disk.

National file system. The creation of a national file system will have the same impact as that of the creation of the original Arpanet. Collaboration between researchers will become significantly easier when they can share the same files. A national file system will be a basic infrastructure REQUIREMENT for High Performance Computing and Communication work. An implicit assumption is that the underlying technologies will be in place to support the associated data movement, including gigabit networks, storage technology, and interfaces between caching file systems. Government leadership is needed to ensure that these technologies will be put in place to satisfy the expectations.

Prototype versions of a national file system are already occurring, led by efforts at Carnegie Mellon University in the development of the Andrew File System (AFS) and archival storage integration efforts at the National Science Foundation supercomputer centers. National AFS cell registration is being done by Transarc today. In the future, the Andrew File System will be subsumed within the OSF DCE/DFS environment. Specific technology issues include generalization of user/address registration, the guarantee of data privacy and file sharing between DFS and NFS. The creation of gateways may make it possible for anyone to hook into the National File System, perhaps by becoming DFS clients. Equally important is the issue of intellectual property protection. General usage of the National File System is not

likely to be successful until some means is provided to give data owners confidence that data and information will be handled consistent with ownership rights and usage authorization.

Scientific database interfaces. The object of storing large amounts of data is the creation of knowledge. Scientific database interfaces have the capability of making knowledge storage and retrieval much easier. The important data to store may then be the knowledge learned about a given data set (metadata) rather than the data set itself. The organization of large data sets is an important research effort. Should data be stored as one large file (100 GB) or as lots of smaller files? How should large files be indexed to maximize knowledge retrieval? How can random access retrieval of large archived files be supported? Observational data set storage, image sequence generation (two dimensional and volumetric) and visualization processing will be major users of scientific database technology.

Databases to date are based on relational or object oriented interfaces and require well-defined classes of object. Scientific users are interested in abstractions of the data points, e.g., surfaces, flow lines, etc. The creation of interactive mechanisms for generating the data abstractions is a research effort. The data flow paradigm used by graphics packages (AVS, Explorer, Ape, Chorus) for constructing visualization interfaces is a possible solution. Integration of research efforts in this technology with large object manipulation research and archival storage research is needed.

Data format standards and conversion tools, XDR/data compression tools. There are many "standard" data file formats, usually discipline specific. Conversion tools that translate between the existing standards are beginning to appear. Examples are the NASA effort for supporting the Astrophysical Data System and the IEEE standards effort for creating a Scalable Coherent Interface (SCI) standard. The SCI standard has been developed specifically for the support of gigabit interconnection of heterogeneous systems and addresses communication and data format issues. The development of standard file formats can be used to help define I/O standards for parallel systems. At issue is the parallel-to-serial data stream conversion required when an MPP transmits data to networks or to archival storage.

Unfortunately, data format conversion tools (e.g., XDR) are still needed to convert between different binary data formats. The IEEE floating point format is now the data standard. Conversion routines between IEEE, CRI, and VMS floating point formats exist. The conversion problems are "man-made" and vendors should standardize on IEEE Floating point and standards (such

as "big endian/little endian" byte ordering, character format) even at the chip level.

Standards for compression of scientific data are being initiated for lossless compression. Lossless compression techniques (e.g., in hardware) for scientific data would be helpful. One difficulty is the decompression of the data when the compression hardware is no longer available.

Privacy. Guaranteeing the privacy of data is a central issue for both networking and data storage. The fundamental problem is trusting the authentication of users and hosts—that they are who they say they are. Possible approaches include the implementation of time-based authentication systems such as Kerberos. A second approach that might be provable is to be able to guarantee you will know when your data has been compromised, even if privacy cannot be guaranteed.

10.7.2 Summary: Data Storage Requirements

Large file access (response time for gigabyte files)

Location transparency (National file system with uniform naming conventions)

Petabyte storage archives (distributed IEEE reference model)

Associated data reference (Scientific database systems—knowledge storage rather than just data storage)

10.7.3 Summary: Data Storage Questions

Is there a "standard" data hierarchy for the storage of data? (Data: Distributed memory, ram disk, local disk, remote disk, tape or optical robot; Scientific database: metadata file systems, archive storage system)

Should data be stored or regenerated? Are there viable technologies such as holographic storage or chemical/optical storage devices that can make data storage cheaper than CPU reexecution of programs?

Should data storage devices be error free, or should error correction be an attribute of the stored data? Bit error rates of one in 10^{12} are too high when terabytes of data are stored.

10.8 Communication Support Systems

Key elements in the technology associated with communication support systems are software support for gigabit per second communication links (in-

cluding high-speed protocols, levels of service for bandwidth reservation, and data integrity), communication media standardization, data privacy, and parallel network I/O.

10.8.1 Gigabit per second Communication Links: Requirements for NREN

A dominant element needed for communication systems is development of the software infrastructure for controlling gigabit per second communication links. With the advent of TeraFLOP computers, data will be generated by a single application at rates higher than a gigabit per second. This is a government-led, high priority research project to develop a basic underlying technology. Gigabit per second testbeds are expected to be up within a year, with the production gigabit per second NREN available in five years. Government funding includes high-speed protocol research and investigation of possible communication paradigms. At issue is whether packet switched networks will be sufficient for high performance application bandwidth requirements? Applications will need to be able to get low latency or uniform latency access for distributed processes. Other types of services (guaranteed integrity, real-time bandwidth reservation, etc.) are also being investigated.

High-speed communications support is needed for many small users (workstations) as well as large users (supercomputers). In such a heterogeneous environment, it may not be enough to reduce network latency to a minimum; it may be necessary to figure out how to deal with variations in latency, both unpredictable (due to operating system scheduling, network load, etc.) and predictable (due to finite speed of light, e.g., satellite based links). Latency effects may dominate algorithm design for efficient use of a wide area network spanning the continent. Algorithms may need to optimize message size versus frequency of messages, and develop mechanisms for communicating without doing lots of round trips. The trade-off for the algorithm developers will be between communication efficiency and computing efficiency for a given wallclock turnaround time. Analyses are needed to provide algorithm and applications developers with strategies for dealing with latency.

Heterogeneous networks comprised of IPI-3 channel oriented traffic and IP and ATM packet/cell oriented traffic will need to be controlled. Issues include control of frame buffer streams or IPI-3 data blocks over a network. Research efforts are being done to understand advantages of packet switched, circuit switched, and virtual circuit paradigms for high-speed networks.

A current issue is the rapid growth of the number of addresses needed to

support just the current network infrastructure. A future need will be to support addressing of entities besides hosts (e.g., individual or subsets of MPP processors, processes, etc.). The addressing requirements may require conversion to the OSI addressing model.

A related need is for networks that can scale to higher communication rates, and not just provide point-to-point access. As we go to MPP, the aggregation of many processors communicating in parallel will mean higher effective LAN rates than a single pipe will be able to support. The model of local environments with higher speed communications than wide area networks will probably continue with new technologies being needed beyond HIPPI and fiber channel. It is anticipated that LANs will take advantage of parallelism on a larger scale than HIPPI (which is 32 or 64 bit parallel). The upper limit for the required speed of the LAN is probably less than the memory bandwidth of the computers. Lower bounds may be derived from application workload data distribution requirements, archival storage requirements, and local disk speeds.

A more realistic bound of the required speed of LAN networks is going to be needed by the Grand Challenge applications. A study of this in today's gigabit per second testbeds may be possible along with a forecast for the TeraFLOP environments.

Application bandwidth requirements are not sufficient for designing appropriate LANs and WANs. Within a computer center, checkpointing will be a very high bandwidth operation. Other sources of operating system bandwidth include collecting instrumentation/performance monitoring data.

If the operating system is distributed, e.g., via multiple functional servers, additional bandwidth requirements are imposed on the network because it becomes the computer backplane. A distributed operating system in the true sense of the concept will not be here within five years. Instead, we are going to see a meta-operating system above individual local operating systems in the HPCC time frame. The concept of a network server will be important as the interface between the local backplane LAN and the wide area network.

10.8.2 Data Integrity and Privacy

Standards are needed for error handling. The transmission of large amounts of data will mean errors are even more an issue. Even very low error rates with transmission of very large amounts of data will translate into almost a guarantee of an error. Correction of errors without packet retransmission

will be needed for supporting networks with high latency.

One networking viewpoint is that security is the responsibility of the host because the host is the entity that does authentication. It is indeed not a physical transport network issue, but a network service issue in terms of joint responsibility between the network and host (operating system, application) communities. If "the network is the computer" then security issues span the entire environment. This should be a base-level function of the entire environment, not just an issue for one or another community.

The higher the level of 'security' the distributed environment has, the more danger there is in losing functionality that users need (e.g., if we do not allow "r" commands, we lose some functionality). The key is to figure out how to provide the functionality in a secure way rather than simply remove the functionality.

Kerberos seems to be the best common denominator for authentication for the vast majority of users. Distributed environments such as Express and PVM/hence must address security.

10.9 Heterogeneous Computing Environments

Key elements in the technology associated with heterogeneous computing environments are programming support environments, resource management, and resource control.

10.9.1 Programming Support Environment

Dominant elements needed for the creation of heterogeneous computing environments are the standardization on a uniform programming support environment (perhaps within four years) and the development of software resource management infrastructure. Current programming support systems span multiple software levels, including entire application environments such as provided by Express, ISIS, and PVM/Hence, languages such as Linda, and even performance analysis tools such as IPS-2.

Standardization of the programming environment is complicated by the competing factors of data granularity versus data locality. Data granularity is a decomposition metric with the issue of whether one should restrict fine-grained decompositions to local environments (i.e., within one machine) or to very tightly coupled machines in the same machine room (e.g., the communications path between an MPP and its front end is as fast as the inter-node rates within the MPP).

Data locality is a data distribution metric which addresses the issue of whether a particular application will work well on distributed memory architectures. Because the application algorithms may require a particular data locality/data granularity pattern for efficient execution, there may be multiple programming environments that will need to be supported.

Increasing the user friendliness of parallel computers requires the automation of parallel code creation. Similar efforts took about ten years for successful automation of vectorization directives for vector supercomputers. Providing users with systems that make generation of efficient parallel codes less painful is a harder problem and may not be completed within five years. Possible approaches include object oriented programming or identification of an easily parallelized subset of Fortran. Other examples arise in the graphics data flow systems such as AVS/Chorus. As mentioned in the discussion of application requirements for parallel computers, optimizing resource utilization for heterogeneous computing environments may require supporting multiple users working on problems that efficiently use a subset of the distributed resources. A closely related issue is the development of tools for supporting a distributed environment. Tools which are appropriate for a single computer may not be easily extended to the distributed environment. An example is the creation of distributed debuggers.

10.9.2 Resource Management

A central component of the metacomputer will be distributed resource management. This consists of remote system status, error handling, and accounting. Systems are being developed to provide support for homogeneous environments. Examples are DQS (Florida State), NQS Exec (trademark, Cummings Group), and Condor (University of Wisconsin). Condor checkpoints jobs when user interface activity is detected to allow "personal" workstations to be integrated into a homogeneous metacomputer. Enhancements are needed for these environments. For example, accounting should be integrated as part of a 'metamanager' resource scheduling system. The metamanager services would also include scheduling, resource management (including load balancing), security, remote status, and possibly even data collection for application workload characterization. Most software developments in this area are still basic research efforts, with some initial working systems that are primarily coming out of universities and government laboratories.

10.9.3 Resource Control

Resource control is needed to do load balancing at all levels of the computing environment hierarchy, from parallel computers to metacomputers to metacenters. The operating system architecture is moving towards the concept of distributed servers. This effectively is the basis of the metacomputer and the metacenter. However, the issues and problems are still being identified. For example, how will distributed scheduling systems co-exist with distributed programming environments? One approach will be to put PVM on top of DCE (i.e., implement the application environment on top of the resource management systems.)

In summary, the heterogeneous computing environment may be a straightforward extension of the parallel computing homogeneous environment. Many of the same issues will be appropriate for both systems. For instance, the question "Should parallel computers be treated as CPU limited resources or as memory limited resources?" could apply equally well to the heterogeneous computing environment. The answer may be that the scientist does not **care** about this if the turnaround is better. Perhaps the efficiency of individual distributed applications should not be so important if the resources can be more efficiently **shared**. The ultimate goal of doing heterogeneous computing is to achieve super-linear speedup of the execution of an application. The mechanisms for doing this will be distributed between the applications, the computing environment, and the operating system kernel. An integrated solution is needed, not only to avoid bottlenecks in performance, but also to minimize the effort needed to create this new environment.

Chapter 11

Visualization Methods

Lew Tucker, Chair
Paul Woodward, Deputy Chair

11.1 Introduction

Today the importance of visualization is clearly recognized in scientific computing. Exploration of large datasets, display of simulation results and interactive steering of computation all require some component of data visualization. Recent advancement in massively parallel systems and the demands of Grand Challenge problems, however, severely test the limits of today's graphics systems. Despite many advances in VLSI and graphics processing engines, many problems remain in scientific visualization that touch upon almost all aspects of high performance computing. These include networking, parallel programming languages, data storage, data formats, and the use of distributed resources.

11.2 Visualization Needs

Application developers in high performance computing have before them a wide range of options for graphics processing. Traditional graphics systems, however, have been developed in large part for realistic rendering and are not always suitable for the display of scientific data. Systems are required that provide flexible methods for displaying large arrays of multi-variable information with implicit or explicit geometries.

Moreover, the sizes of the datasets produced by today's Grand Challenge problems exceed the capabilities of traditional graphics processing platforms. Visualization systems must therefore be designed to handle large datasets, to scale with increasing processing resource, and function effectively in a heterogeneous environment.

Visualization needs of application scientists fall into several categories:

- **Data navigation.** Very large data sets defy manual inspection of numeric output. Visualization combined with navigation allows the scientist to explore rapidly extremely large datasets.
- **Presentation of results.** Pictorial form is increasingly being used to communicate results to other scientists, sponsors, or the general public.
- **Animation.** Animation is needed to show time-varying images produced by long running simulations. Videotape production facilities are becoming commonplace but impose significant delays between the "experiment" and viewing of results. Advances in mass storage systems based on RAID technology offer one solution for immediate storage and playback of image sequences.
- **Interactive steering of computation.** Display of intermediate results permits the scientist to modify or abort computations before completion. In some problems, feedback from the investigator may be required to steer computation around local minima.
- **Application development and debugging.** In the development of parallel applications, visualization is often the only way to examine thousands of variables or the behavior of asynchronously executing tasks (see Chapter 8).
- **Remote access.** Scientists working on high performance systems increasingly use remote network-based access methods. High-speed national networks are clearly required for remote visualization of results (see Chapter 12).

11.3 Visualization Software

Goals for the development of scientific visualization software in high performance computing include:

- **Ease of use.** Computational scientists are experts in their own disciplines and in general do not want to spend time learning about or programming computer graphics. Visualization software needs to be easy to use, intuitive and flexible.
- **Software reuse.** Supercomputer centers typically employ staff to assist in developing visualization software. This is costly and it is therefore desirable to obtain the highest benefit by developing software that may be reused in a variety of applications.
- **Software modularity.** Modular development of software has proven to be essential for increasing productivity in systems design. Separation of visualization modules from specific application solutions makes systems easier to develop and maintain.
- **Scalability.** High performance systems generate extremely large datasets. Visualization systems designed for parallel machines are required if scalability is to be achieved.

11.4 Distributed Visualization Environments

In today's computing environment, graphics workstations provide cost-effective solutions for visualization of scientific data. Client-server graphic protocols such as X Windows and PEX have also emerged as standards, making it possible to envision delivery of visualization to the scientist's desktop.

Software-based distributed visualization environments (AVS, SGI Explorer, Chorus, etc.) also are gaining popularity. Although originally designed for workstation operation, the framework provided by these systems offers a means by which distributed systems may be connected, be they workstations or high performance computing systems.

Distributed visualization environments (DVEs) are characterized by a user-built framework of interconnected modules representing the dataflow for a given visualization application. Such systems support rapid prototyping, interactive application steering, and device independence. From the application writer's perspective, this approach relieves the programmer from needing a detailed understanding of graphics and offers significant software savings through the reuse of visualization modules. Because the interfaces between modules are well defined, modules developed at different computer sites may be freely shared.

The dataflow network, constructed at run time, supports distributed computing in a heterogeneous LAN or WAN network environment of high performance systems and graphics workstations. Modules themselves perform the underlying work of data input, feature extraction, data-mapping, rendering, and output. DVEs hide the inherent complexities of different internal numerical representation, byte ordering, vector/scalar/parallel visualization algorithm dependencies, or serial/parallel I/O constructs in a modular HPC heterogeneous framework. Overall, DVEs give the user control over the optimization of heterogeneous resources and network capabilities.

Users may interactively steer simulations through adjustment of an application's critical parameters. Also, modification and adjustment of the visualization process that may be performed in an interactive manner. The same framework which facilitates the easy creation of complex visualization applications also provides for its quick modification to include new and different data-mapping and feature extraction functions.

The modular approach of DVEs lends itself well to software sharing, module reuse, extensibility and flexibility. Software sharing means that modules created for one high performance graphics architecture can readily be ported to another similar architecture. The reusability aspect of DVEs means that HPC visualization code and algorithms are modularly developed and compiled once, and then reused indefinitely in a plug-and-play mode in a variety of diverse scientific applications. The extensibility of DVE frameworks is such that users may write their own modules in their favorite HPC language binding.

11.5 Distributed Visualization in HPC

In summary, distributed visualization systems offer many advantages for high performance computing in terms of software reuse and modularity. For such systems to be effective, attention needs to be given to several key software issues:

- High Bandwidth Parallel I/O

  While the separation of visualization modules from the application provides an intuitive mapping of processes to a distributed system, it is not without cost. Performance of these systems on massively parallel machines will in large part depend upon the systems' ability to transfer quickly massive amounts of data between modules. System software is

needed to support high-bandwidth parallel I/O and/or shared memory segments between processes. Protocol standards are needed to support communication between different parallel machines.

- Common framework for workstations and high performance systems

  To leverage the growing software base of modules developed for workstations, the frameworks provided by various distributed visualization systems need to be extended to include the execution of remote modules on high performance machines. At run time, scientists should be free to mix local and remote modules into a common network.

- Development of massively parallel visualization modules

  Serial modules developed for workstations will not be sufficient for visualizing the large datasets of high performance machines. Modules designed to execute on massively parallel machines need to be developed. This implies support for research and development of parallel rendering techniques.

- Distribution and sharing of modules

  To take advantage of the reusability of visualization modules, a national repository and distribution center of visualization software written for high performance systems needs to be identified. Individual centers should be encouraged to share their development efforts.

11.6 Conclusion

DVEs allow the researcher to concentrate on analysis without the need for code editing, compiling and rerun cycles associated with more traditional monolithic application approaches. The resulting visual dataflow graph offered by the DVE graphical user interface (GUI) provides an intuitive visual map of networked remote information resource utilization. DVEs therefore allow the researcher to focus on data exploration, leaving the details of distributed computing and graphics to system software and library developers.

PART III

Chapter 12

Issues and Observations

12.1 Introduction

The working groups addressed the challenge and problems of exploiting massively parallel processing systems from the perspective of their respective disciplines. But the split into working groups was along the lines of conventional disciplines and did not constitute an orthogonal set of viewpoints across the domain of interest. Many underlying issues were dealt with by several, if not all, of the working groups. The purpose of this chapter is to provide a synthesis of the observations contributed by the groups concerning the issues that dominated the workshop. Also, we highlight important points of disagreement about which multiple contending views were expressed. Finally, we suggest topics that did not receive much attention that might be considered relevant by others in the community.

12.2 Shared Goals

The goals of system software in the context of high-performance computing were uniformly embraced. Where these goals were in conflict, different balances in trade-offs were supported by various parties. However, consensus quickly emerged on the key attributes of the computing environment that systems software should provide or enable and that can be developed through near-term research.

12.2.1 Performance

The foremost objective of all work in systems software is to exploit the raw peak performance potential of large ensembles of very high-speed microprocessors in solving Grand Challenge end user problems. It is expected that hardware vendors will supply the best technology available for processing, interprocessor communication, file storage, and wide area networking. System software will provide the glue, hooks, and handles to tie the pieces together into a logically consistent whole and to give user access to it. System software will expose the performance opportunity, not consume or inhibit useful cycles for ancillary purposes.

12.2.2 Portability

The single greatest hindrance to significant penetration of MPP technology in scientific computing is the absence of common programming interfaces across various parallel computing systems. The effective lifetime of a major piece of code is cut short if the programming model and underlying execution substrate change continually. Such variability precludes investment in large mathematical libraries or major applications. System software is required that bridges the gap between long term code investment and continuously evolving systems. Further, the same programming model must reside on diverse platforms so that the value of code development can be shared among a broad user community. Finally, MPP systems are characterized by scale. A given program should run on small or large configurations without rewriting, and preferably without recompilation. All of these requirements fall into the category of portability. A parallel program, once developed, should operate effectively on different system types, of variable scale, over many generations.

12.2.3 Usability

If lack of portability inhibits broad acceptance of MPP technology for operational scientific computing, the complexities of marshaling its capabilities in support of real world problems severely limits the productivity of those who try. The combination of parallel computing elements and disparate access latency times greatly compounds the problem of effectively matching the demands of the problem with the resources of the system. In comparison, conventional uniprocessor programming is relatively simple. It must be the goal of system software research to harness the capabilities of MPPs while protecting the user from its low level details. Where details must be

confronted, tools must be devised to assist the programmer in rationally selecting among a clear set of choices with all necessary information readily available and conveniently presented.

12.3 HPC System Structure

While many variants are possible, a general and clear consensus existed about the basic form that high-performance computing systems are likely to take over the next couple of generations, including the first TeraFLOPS systems. This shared view simplified discussion of the role and nature of system software evolution. The principal components and structural attributes are summarized as follows:

- MPP component processors off-the-shelf from commercial fabrication lines
- High density local memory
- Rapid message-passing interprocessor communications
- Multiple computers of different types integrated by LANs forming composite heterogeneous systems
- Very high bandwidth wide area networks interconnecting geographically separated computing centers for shared computation and uniform national file system
- User interface via graphics workstations

12.4 Role of System Software

Within the framework of these goals and class of systems, system software must play multiple roles. These can be grouped in two major categories: facilitating the task of programming these complex computing systems, and contributing to the runtime management of the resources themselves.

12.4.1 Facilitate Programming

System software, which includes programming languages and compilers as well as debuggers and performance monitoring tools, presents a logical view

of the parallel execution system to the applications programmer. All machine resources are acquired and manipulated through this sometimes convoluted layer of abstraction. To the extent that the programmer controls the system, all media of manipulation are granted by means of system software. While there were many opinions of what exactly is the proper balance between programmer and system responsibilities, the following were recognized as key requirements of system software imposed by the applications programmer:

- Language includes all basic operations
- Means of representing program parallelism
- If part of the programming paradigm, direct message processing
- If part of the strategy, direct resource mapping notation
- Debugging tools for program correctness
- Performance monitoring and presentation

12.4.2 Manage Resources

Many approaches to resource management are possible, as will be discussed in succeeding sections. The main objective is to be able to apply available resources to pending work with minimum loss to overhead, contention, and latency. In many instances this means that resource management is driven directly by application program specification. In other cases, system software is responsible for allocating processor and memory resources as a function of runtime status and compile time analysis. While distinctions do exist among approaches to controlling MPP resources to best support parallel applications, the following dominant capabilities are shared by most system architectures.

- Interprocessor communication and buffering
- Interprocess synchronization
- Program task processor assignment
- Memory hierarchy management including cache and page tables
- Task scheduling and context switching

- Collaboration, coordination, and management in heterogeneous computer ensembles
- Exception handling, and checkpoint/restart for robustness
- Observability of performance and correctness sensitive mechanisms

12.5 Uncertainties Concerning TeraFLOPS Resource Requirements/Balance

Although representatives from a number of different sectors of the computing community participated in the workshop, it became clear that predicting the nature of behavior and requirements of TeraFLOPS scale MPPs with any confidence was not possible with the knowledge base available. Experience with this class of computing system is sufficiently novel and the two to three orders of magnitude performance ratio between the largest of today's systems and the TeraFLOPS systems overwhelmed what little could be said concerning actual scaling attributes. Of particular concern are the relative ratios of resources. For example, how much memory will be required per processor? What will be the ratio of computing operations to I/O operations? What size backing store will be necessary for the resulting data sets?

The confusion regarding requirements at the TeraFLOPS scale is compounded because of uncertainties of the methodologies to be used. It can be anticipated that programming models, software environments, and hardware support will alter measurably, in certain cases dramatically, over the next few years as a consequence of the widely shared base of experience that will accrue throughout the intervening period. Therefore, the programming driven demands and balance between system software and hardware support may exhibit little resemblance to the relatively primitive systems and strategies only just emerging.

12.6 Immediate Needs

It is clear that the problem definitions and system hardware are more advanced than the corresponding system software. An urgent need exists for some basic tools to help applications and systems programmers grapple with the immediate programming activities. While favorite hit lists may vary among individual users, broad, even enthusiastic, agreement was evident about the following needs.

12.6.1 Debuggers

Writing correct programs requires the means to examine program state at any point within its execution, reconstruct the flow control sequencing (sometimes incrementing one step at a time), and relating the memory content transitions back to source code variables and flow control. While reasonable tools for debugging programs on uniprocessors have been available for some time, this same capability is not generally available on vendor MPP offerings. The problem of debugging in the context of parallel execution is aggravated by the potential for race conditions, message send/receive mismatches, and other concurrency-related timing errors. Contending with this form of problem is largely beyond the capabilities of almost all debugging tools. At this stage, even the most basic debugging facilities that make available the state of the distributed processors would be a major improvement. Without these tools, parallel programming is difficult.

12.6.2 Performance Profiling

While correct program execution is essential for useful problem solving, almost as critical is the effective use of parallel resources in delivering scalable performance. At this stage in the evolution of MPP techniques, users contribute significantly to the determination of resource allocation in order to satisfy application program demands. Dealing with the complexities involved requires feedback depicting the behavior of the parallel system while executing the parallel application. The nature of this feedback needs to be such that the programmer can recognize the sources of performance degradation and can institute corrective measures. In the long term, performance monitoring systems will have to be implemented combining hardware and software instrumentation in conjunction with appropriate visualization subsystems. But before such capabilities are realized, it is at least necessary to provide profiling tools on a per-processor basis similar to those found on conventional uniprocessors.

12.6.3 Checkpointing

Checkpointing facilities are essential in large-scale computing. Solving a single problem often requires more hours than one can run in a single shot, and even mature computer systems with conventional architectures break occasionally; saving intermediate results is essential. Current generation MPPs

are still experimental and suffer from problems of hardware and software reliability. Checkpointing is the generally applied method of offsetting the bad effects of crashes, etc., by preserving the partial results of program execution and enabling a restart without having to initiate computing at the beginning. MPP operating system and compiler support for checkpointing/restart are urgently required. While it may not be possible to achieve transparent recovery, at least the means for explicit user-specified checkpointing should be made available.

12.6.4 MPP C

While most scientific applications can be expected to be crafted using one of many variants of Fortran, development of system software is better done using C, which has become the de facto standard for such work, at least in the context of Unix like environments. With the added complexities associated with parallelism in MPP structures, hooks to the additional resources and controllers such as message-passing hardware should be easily accessible to the systems programmer from C in the most efficient way. Near term augmentation of C to incorporate such facilities is critical to support rapid and reliable implementation of system programming. An MPP C needs to be developed and at least informally standardized so that systems programmers can work effectively and systems source code can be easily portable among compilers.

12.6.5 Accomplished through Incrementalism

The urgency of demand for the above tools emphasizes utility and immediacy over sophistication and finality. Basic but trustworthy tools were universally preferred over sophisticated but unreliable or inefficient systems. Progress in each of these areas cannot be delayed in order to benefit from research results downstream. Thus, a philosophy of incrementalism was adopted which encouraged "picking the low hanging fruit" by making frequent gradual changes to tools to gain some benefits in the near term. This overt pragmatism from a research community reflects the intensity of the demands for even the most primitive tools to assist in grappling with the challenge of harnessing the potential power of MPPs.

12.7 Sharing

Key to early success in system software for HPC is sharing of results and tools throughout the community. Scientific computing is a relatively small part of the world's total computing budget, and parallel processing is a small portion of even that. The resources available to address the MPP system software problem are minuscule in comparison to the rest of the computing market. For this reason, any investment by the research community in advanced system software development must leverage preexisting components as much as possible and must be made available to as broad a community as possible. Sharing in this context is very difficult and requires some discipline in how software and information in general are represented and distributed. These issues were encountered repeatedly throughout the workshop and are summarized below.

12.7.1 Interoperable

System software is an assemblage of many logically interconnected components whose true value is only realized through their synergistic interaction as a software system. To share system software components among separate research groups, the components must be designed in a way that makes assembly reasonably straightforward without prior knowledge of the total system in which they are to reside and to which they are to add functionality. Interoperability is that characteristic of a component software module that ensures compatibility with other appropriately designed software modules. Interoperability is an evolving software engineering discipline incorporating ideas from functional programming, object oriented programming, and plug-and-play methodologies. True robust, mindless interoperability is probably beyond the capability of current techniques. However, if intent for seamless interface is part of adapted software design style, ease of integration will be greatly enhanced, facilitating software sharing. Of utmost importance is that portability, one of the three critical attributes of HPC software environments, relies on effective interoperability techniques.

12.7.2 Standards

Repeatedly throughout the workshop, areas were identified where standardization was deemed essential. Interoperability of independently developed experimental software modules is greatly facilitated where interface stan-

dards have been devised and adhered to. Standardization at all levels of user interaction with system software enhances the utility and extends the lifetime and value of software investment. It shortens the design cycle because target I/O and interface formats are then already thought out and guaranteed to correctly support logical data transport. A number of opportunities were cited for which the adoption of standards would greatly enhance sharing and development of software. These opportunities included message-passing constructs, compound data formats, data parallel language syntax, distributed file system protocols, and support libraries. Together, such standards provide an infrastructure that establishes the means by which software can be written to be shared and, of more immediate value, borrowed to reduce development time to operation.

12.7.3 Libraries

Conventionally, libraries have been used to archive large, important, and widely applicable software packages such as scientific mathematical routines. More recently, libraries have been well-defined substructures of a system hierarchy which defines much of the functionality of a computing environment. With the advent of MPP systems, many of the highly valued libraries no longer are germane because they either can not serve the needs of the multiple processor system structure or (more usually) they can not take advantage of the class of parallelism offered by MPPs to achieve maximum performance for their respective functions. If MPPs are to become truly as useful to the scientific community as conventional supercomputers have been, then the important libraries upon which much of the scientific computing community has come to rely must be replaced with identically functional libraries designed to exploit the new capabilities of MPPs. But, in assuming a parallel execution model, many subtle algorithmic considerations arise, not only for performance through concurrent flow control but with regards to determinacy and correctness as well.

12.7.4 Templates

The need for collections of usable software modules for scientific computing and system management is immediate and can not possibly be met with the limited available resources in the near future. An aggravating problem is the number of different host parallel systems and system environments for which such software is required. A possible compromise, advocated by some

workshop participants, is the heavy use of templates. A template, in this instance, is a description of a general algorithm rather than the executable object code or source code more routinely found in conventional libraries. In a template, many of the system dependent details can be supplied by the end user, configuring it for the specific system upon which it is intended to run. Templates exhibit two significant properties. First, templates are general; they can lead to ports to diverse machines. Second, they allow for anonymous collaboration. The expert algorithmist creates a template reflecting in-depth knowledge of a specific numerical technique or, in the case of a functional driver, the component (logical or physical) to be managed. The user of a template then provides the value added capability to the general template description that customizes it for the specific context or environment needed. Templates are not language specific and will no doubt be captured in some algol-like pseudo code (perhaps not even rigorously formalized) that is readily translatable into the target high level language. It was thought that for practical use, an example of one template instantiation into a real world language and a test suite should also be provided.

12.7.5 Software Exchange

Beyond constructing software to be reusable, means are needed to disseminate both the code and knowledge of the existence of the code. A simple and uniform method of accessing such data from widely disparate sites would greatly facilitate the evolution of MPP application and system software. Such a system is being developed, incorporating many data bases around the world and providing uniform interface to software library repositories with differing requirements. The HPCC National Software Exchange combined with an on-going software exchange experiment is to determine the viability and utility of such a global software system.

12.7.6 Source Codes

When dealing with diverse systems and research software, many subtle interactions with unforeseen consequences may occur, often to bad effect. Tracking down the exact nature of the cause can be difficult, even with all of the information available. Without source code for all constituent elements of the software environment, it is frequently impossible. If, as was the general consensus, sharing of experimental software will be an essential element of successful HPC system evolution, then two aspects of this context demand

that availability of source code be mandatory. First, experimental code, by its very nature, will be capable of unanticipated behavior under circumstances other than those explicitly imposed by the originators. Only access to the source code can provide the means for active users to ascertain the cause of a problem and exact a fix, recompiling and linking prior to new runs.

Even if no bug is experienced, recompilation may often be necessary because of the peculiar nature of a particular experimental parallel system which may, in part, be reflected by the local compiler. The second aspect of shared experimental software is that documentation is rarely provided in sufficient detail, or correctness, to provide all necessary information to system software integrators. The source code becomes the documentation and is therefore essential for the code to be useful outside the environment in which it was originally developed. With source code, experimenters are much more likely to be willing to take a chance and, therefore, the work of a few may have a positive impact on many. An added advantage is that with source code, an experimental software package may be extended by others, enriching its capability and robustness beyond that provided by the originators.

12.7.7 Test Suites

In any experimental environment, few things are stable. Yet a given problem should return the same results (if deterministic and driven by the same input stream) even as the underlying computing system is permuted. For shared software modules and tools, contributors should include test suites of input data sets and corresponding result data against which execution on experimental systems can be tested and verified. While it is naive to think that any suite or set of suites will be capable of finding all possible kinds of errors, the most likely and egregious errors caused by system software faults will be detectable through this means. Without it, even the most simple and obvious mistakes could go undiscovered for some time, confusing other observations and possibly corrupting other software components.

12.8 Resource Allocation and Management

Effective performance within the HPC arena is achieved through application program parallelism, hardware speed, and optimal mapping of computing

requirements to physical resources. Allocation of resources to program demands is a major component of the challenge of realizing the full potential of these experimental systems. System software, much yet to be developed, is the principal means of managing the parallel computing resources of MPP systems. Indeed, how the end-user perceives these systems is largely a consequence of how they are presented to the user by the intervening system software. Currently, because the user must be so intimately involved in every aspect of resource allocation decision-making, the machines are viewed as truly distributed ensembles of separate computers and networks. This contrasts markedly from a users perspective of a conventional uniprocessor in which almost no resource related decisions are required and attention can be given exclusively to efficiency of code itself. To close the gap between these two viewpoints (making MPP systems more application friendly), issues concerning the nature of the virtual machine that should be reflected by the systems software received much attention throughout the workshop. The recurring issues are highlighted below.

12.8.1 Shared Address Space

Direct access to programs and data on a distributed memory computer is available only locally to the node upon which the access request originates. Access to data on remote nodes is acquired through the explicit intervention of service routines executed on both processors coordinated via interprocessor messages conveying the request at a higher level protocol. The most basic system support relies on programmer specification of all such transactions. The environment presented to the programmer, therefore, is truly that of a multiple computer system. Conceptually, the variable and control state of a user's application makes up a single name space that is independent of its assignment to an MPPs distributed resources.

For reasons of programmability, scalability, and portability a *global reference space* should be presented at least at the source code level. This is often referred to as a "shared address space" but should not be confused with only those specific systems using snooping bus based cache coherency schemes. Message-passing mechanisms can be employed in systems that support global reference space semantics. Indeed, for MPPs this is probably essential for limiting communication contention through split transactions. However, the programming model presented is not that of message passing semantics. It is still important to understand the physical distribution of the objects and their relationship to tasks applied to them. This is necessary for

latency reduction through locality exploitation and register data passing in lieu of message packet transfers.

12.8.2 Role of Program, Compiler, Runtime, O/S

Once, the operating system was assumed to perform all resource management and allocation responsibilities within a computing system. With the advent of MPP system architecture, efficient exploitation of parallelism requires a readjustment of responsibilities among the programmer, compiler, operating system, computing environment, and runtime system. There are two reasons for this. The operating system mechanisms have to be fully general so they may support all conceivable needs of any program being executed by the system. For this reason, the mechanisms are often cumbersome. This is exacerbated by the need to trap to the kernel for each service call, performing a context switch in the process. The resulting large overheads make it inefficient to use any but the coarsest grain tasks, thus limiting the amount of parallelism available and consequent potential scalability.

The operating system should support program execution but stay out of the way of task scheduling and synchronization. These can best be performed by the compiler when sufficient knowledge is available at load time, or by a runtime system if dynamic mechanisms are necessary. Access to the runtime system is simply a compiler managed subroutine call. Runtime mechanisms are generally light weight and are chosen for the particular job they are to do within an application. Indeed, runtime library modules can be compiled with the application source code in some cases to provide the benefit of user program knowledge in compiler optimization and runtime servicing. It would be excellent if the compiler could solve the dusty-deck problem, detecting all parallelism, resolving all dependencies, and precluding all conflicts and race conditions. In many cases, the compiler can do much of this. However, at least currently, parallelism must be considered as part of program structure and the means of representing parallelism must be part of the semantic constructs of the programming formalism employed by the user. What is less clear is to what degree the programmer must also intervene in making explicit resource management decisions. This is discussed in more detail in the next section.

While the role of the operating system may be diminished in controlling some of the details of application program execution, its responsibilities will become more important in managing the distributed file name space and supporting heterogeneous computer system processing. MPP computers find

themselves in a rapidly expanding distributed file system that will shortly evolve into a National File System, i.e., a large hierarchical uniform file name space such as CMU's Andrew experiment. Operating systems will directly access and cache files from around the country in real time as well as supply local file access support to other remote facilities. When more than one computer is applied to the execution of a single program, the operating systems of all computers involved have to automatically integrate their logical functionality, reserve appropriate capacity on each system, and coordinate the program flow control. These requirements exist right now when integrating a graphics workstation with an HPC compute server.

12.8.3 Philosophy of Efficiency vs. Ease of Use

At one time, programmers explicitly managed data placement and movement in registers, main memory, and on disks (and drums). The advent of virtual memory, automatic cache management mechanisms, and sophisticated optimizing compilers largely eliminated the need for direct programmer supervision. This is not to say that a programmer can be totally ignorant of the implications of program structure on storage facility usage. But at a sacrifice of some performance, the memory system hierarchy can be largely transparent. The occasional cache and disk thrashing can usually be avoided through some care. With MPPs, memory access times experience a broad range of latencies as well as latency variance. The question of relative degree of transparency of memory and processor supervision by the programmer must again be addressed.

Programming MPPs is greatly complicated by the presence of multiple resources and the need to coordinate them. If such issues could be ignored by the programmer, the difference in ease of use would be dramatic. But wisdom drawn from experience indicates that the degradation in efficiency and performance also would be dramatic should such issues be ignored. The other extreme is full programmer specification of the mapping of program activities and data blocks to system processors and memory blocks, including explicit control of message passing and synchronization. Scientists seeking the fastest available systems have little sympathy with sacrificing performance for almost any reason. Yet, to the extent possible, software tools and analyzers must remove much of the burden of the programmer performing tasks that had conventionally been done by operating systems. Some associated performance degradation is likely. Therefore, programmer hooks must always be available to override control decisions when user intervention can

significantly enhance effective performance.

12.8.4 Synchronization and Scheduling

Except for the most complicated systems codes, uniprocessor application programs rarely are required to consider issues of synchronization and scheduling. Parallel processing brings these operations to the forefront of programmer consideration. Both synchronization and scheduling relate to the time ordering of concurrent program activities on multiple computing resources. A difference between the two is that synchronization is more closely associated with program flow control semantics and manipulated by the programmer whereas scheduling may be done entirely by the compiler, runtime system, or operating system. Synchronization relates the interdependencies of the program tasks and scheduling associates pending tasks with available resources. Scheduling also may take into consideration the logical locality of tasks and data sets. Thus, although not explicitly part of the program semantics, this implicit information may be necessary for effective scheduling and efficient resource management. Currently, spatial scheduling is usually done explicitly, often through program annotation of data partitioning. But as compilers become smarter and more closely tied to the runtime system, these responsibilities may become transparent to the programmer.

12.9 Robustness

Reliability takes on many facets in the exploitation of the potential of MPP systems. Hardware transient errors, system software errors, subtle user program stimulated timing or coordination errors, and even administrative errors all contribute to reduction of effectiveness of these systems in solving large end-user problems. Because of the complexities and experimental nature of these systems, reliability is more fragile than users have come to expect from their computing support environments. Yet, due to the relative scale of the problems being attempted and the difficulty in crafting an application problem on these systems, the relative impacts of failure and cost of recovery are substantially larger. In the near term, more robust systems will be realized through advances in system software that both reduce the likelihood of catastrophic error and ease the recovery cycle. Central to enhanced effectiveness is improved software (and possibly hardware) support for checkpointing.

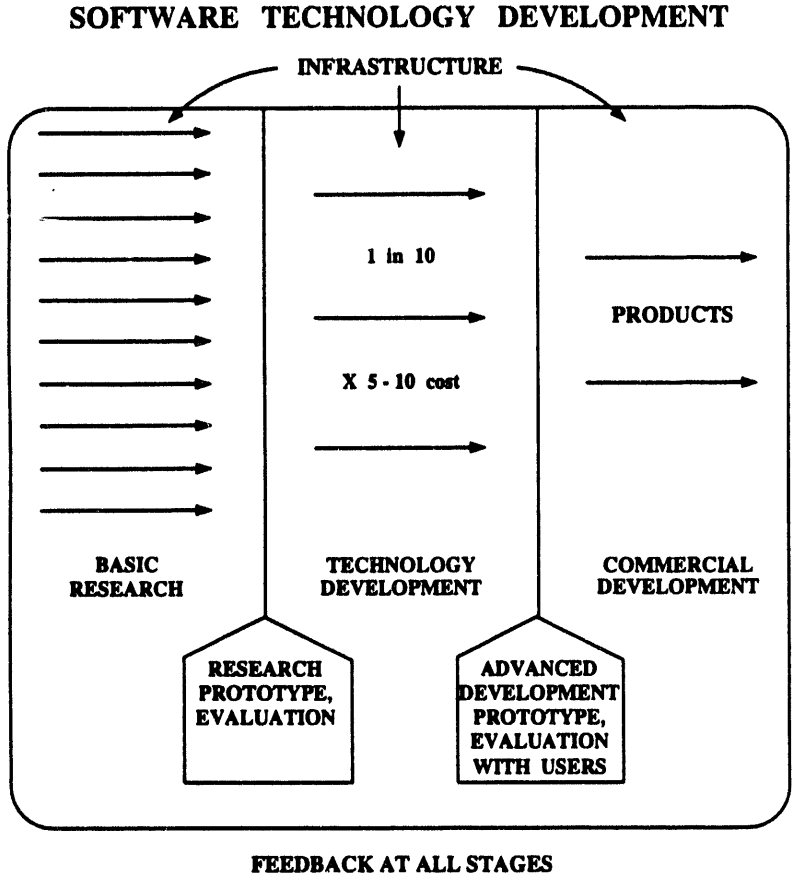
12.10 Monitoring System Behavior

Among the hottest topics at the workshop was the need for many forms of feedback from the system to the user to help in building correct and effective programs on MPPs. Those discussed extensively were debugging and performance monitoring support. Debugging parallel programs takes on a complexity beyond that experienced in a uniprocessor environment. Not only are there many processors independently capable of error, but the temporal dependencies among processors can invoke subtle and insidious bugs. While it is necessary for previous debugging facilities to be ported to these new systems, new kinds of debugging tools need to be developed to detect and isolate timing race conditions and message send-receive conflicts, as well as other timing-related difficulties.

Even correct programs may behave poorly due to ineffective program mapping. It is essential that parallel systems provide the means by which programmers may observe the program behavior to determine the sources of performance degradation. Such tools are not so easily devised. Software tools are intrusive and under certain conditions can grossly alter the execution flow of a program and at a minimum will change the timing somewhat. Hardware support for performance monitoring may be essential but is not easy to come by. Fortunately, there is a trend by vendors towards supplying some hardware hooks for collecting statistics on performance. This is particularly important when there is no software-accessible measurement of an important parameter like bus contention which falls below the semantic horizon of software observability.

12.11 R&D Priorities and Responsibilities (Funding Policies)

Much consideration was given to the problem of applying limited resources to achieving the systems capabilities needed to make HPC technology routinely applicable to scientific problems. Many practical concerns were recognized, especially those related to the sharing of experimental software. In spite of the best intents, early experimental software is infrequently in a form that can be used by other than the originating individual or group. The code is not robust, does not encompass the general cases, and has inadequate documentation. Yet, it is from these sources that the ground breaking work often comes. It is recommended that funding agencies, academic institutions, and



industrial vendors collaborate to facilitate a process by which research codes may eventually find wide use in the commercial computing community. A three-stage process is recommended; the above figure depicts the strategy. Small research grants would be provided, as is currently the case, to many researchers to try out new ideas. An evaluation process would select a small percentage of these. For those chosen, additional funds would be provided for development of prototype versions of the experimental codes. These prototypes would be used by friendly teams to shake out the ideas and implementation. After sufficient experience has been attained, a small number of those may be selected for commercialization, again sponsored at least in part by funding agencies. These final products would then be available to all users of MPP technology.

12.12 Points at Issue

Any conference dedicated to exposing the opportunities and challenges of an exciting new technology will be permeated with differing views, sometimes enriching understanding, and other times simply exposing (through conflict) collective confusion. Rather than obscure these differences, we take this opportunity to convey them for the sake of fidelity in capturing the experience of this workshop. Below are some, but by no means all, of the issues and positions that spawned debate and that will ultimately have to be resolved if MPPs are to replace prosaic supercomputing architecture in advancing computational performance.

12.12.1 Who Dictates Requirements

For these experimental machines, it is far from clear from whom the system specifications are derived. In certain well behaved cases, the application programmers can predict how extant codes will scale on future larger systems; however, in many instances program scaling into new performance regimes is simply unexplored territory. Vendors are constrained by the capabilities of their present device technology and processor architectures. So they must operate within a narrow window of implementation feasibility. Parallel programming and execution models are still evolving and it is not apparent which ones should ultimately achieve direct architecture support. Finally, even if added functionality can be provided, the cost in performance may be unacceptable to the application community. A healthy tension exists that will probably endure throughout the evolution of HPC systems.

12.12.2 Programming Model and Languages

Great uncertainty exists regarding the user interface to parallel systems. The parallel programming model and its representation as a computer language is complicated by at best modest underlying resource management systems software. Programmers are often driven to a hands-on approach; sometimes resorting to assembly level programming. This inhibits portability and extends software development time while severely limiting its utility lifetime. While Fortran derivatives appear to be the syntax of choice among the science community, embellishments are slow to come and tools are required now to harness MPP potential. Even with message passing or data parallel forms of Fortran, the user is limited to a constrained, albeit useful, parallel pro-

gramming model and must consider the physical distribution of the program subcomponents. At the other extreme are parallel programming languages that explicitly reveal the program's parallelism, but do not demand user intervention in resource management. While this allows an HPC to be treated as a single entity, it distances the programmer from the means of control that might be essential in optimizing performance. A built-in conservatism prevails because of the investment required to realize a new programming environment and the low probability that it will be widely embraced by the user community; the learning curve being a sometimes impassable obstacle. To paraphrase an unfortunate statement of a recent senior political leader: we want the status quo to remain just as it is.

12.12.3 When to Address Heterogeneous Processing

The question of how the heterogeneous processing problem should be addressed from a systems software perspective was one of considerable debate. On the one hand, it was felt that the homogeneous processing problem was still unsolved, was tough enough, and that heterogeneous processing would require even more effort. To focus resources on the heterogeneous issue would be to dilute the more valuable effort on homogeneous computing. However, a strong contrary opinion was voiced as well. It was pointed out that the heterogeneous system issue is integral to even current networked computing environments. MPPs and high-speed vector machines are served by large distributed file servers and are controlled from graphic workstations with visualization requirements. These distinct systems need to operate as a synergistic ensemble, working together on a single user problem. Without proper direction and coordination, any one could become a severe bottleneck, degrading the performance potential of the others.

Part of the discussion revolved around the level of seamlessness or transparency afforded by the system software supervising heterogeneous resources. One camp felt it necessary for the programmer to be protected from the possible complexities of such systems and that this capability was unlikely to emerge in the near term. The alternate view was that the heterogeneous structure should be exposed to the user, but through an object oriented model which describes the coarse structure of an application in terms of separate execution environments that form client-server relationships and support each other's needs. Such a program organization, it is imagined, would permit a plug-and-play programming approach that would readily map onto elements of a heterogeneous system. Between these too

views were numerous compromises that tended to seek a balance.

12.12.4 Degree of Current Successes

There was even some debate about how good/not good things are at the current time. This was spawned by the widely distributed (intellectually) and sparse nature of the collective experience base. Over the last decade, many different parallel programming models and systems, resource management and allocation strategies, and performance vs. programmability trade-off philosophies have been pursued throughout the academic, industrial, and government communities. The dilemma of how much has been achieved versus how much there is yet to achieve is akin to the half full/half empty bottle argument. Many experimental programming systems applied to a space of applications on one (or sometimes a few) machines have shown real potential. Many more models, languages, and even architectures have been proposed. The question would often be cast as, what a user can get today on a specific system to make his programming easier and more effective? It was recognized that select experimental systems must be commercialized if their potential benefits are to be embraced by the user community.

12.12.5 Role of Architecture

Surprisingly, as indicated in an early section, there was a basic expectation of the class of system that was the target for HPC system software. Yet, it was understood that many of the costs in managing computation with the current architectures severely limited the freedom for adjusting resource allocation to program demands on the fly. Overheads for communication and synchronization as well as data migration and coherency can easily become unacceptable on MPP architectures. More prosaically, bandwidth for I/O to external networks, distributed file systems, and graphics workstations is anticipated to be insufficient for many applications. Therefore, it is also recognized that new architectures at the processor and system level will be needed to provide more efficient mechanisms in support of advanced semantic policies of flow control and resource management.

Even if the opportunity to devise a new architecture existed, a financially daunting prospect, there is no consensus as to what capabilities it should embody. The same uncertainties that confront system software are equally germane to decisions about parallel architecture. There is even a school of thought from the RISC philosophy that says that all but the most basic

and widely used mechanisms should be in software: minimizing the cost of hardware, making possible the shortest cycle times, and giving the compiler complete control at the microcycle level. But, with the realization that caches are becoming a significant portion of the system total cost, the idea of cheap processors may be non-viable. What was not adequately discussed is how the experience base from developing application and system software for MPPs should be applied to the processor and system vendors to encourage architecture changes. It was considered by some that this was outside the scope of the workshop; others thought it relevant.

12.12.6 What are Reasonable Costs/Hits for Capabilities

The trade-off space between performance on the one hand and portability and programmability on the other is perceived by many to be wide. How much one is willing to trade for usability in terms of performance varies significantly among users and application requirements. One group asserted that 10% performance degradation over some imagined best performance would be unacceptable even for real improvements in programming environment. Others recognized that the time to debug and optimize an application was important, especially for experimental codes that have short life times. If an order of magnitude improvement in code writing or porting time could be achieved through high level programming tools, some would gladly accept a factor of two performance reduction. What many would like is the opportunity to make that decision on a per case basis, optimizing critical code, but having the opportunity to get real code up and running quickly if it is desirable. This is probably an important goal for system software development.

12.12.7 Will Templates Really Work?

Recognizing both the need and difficulty of software sharing to enhance the rate at which these systems become useful to a broad spectrum of problems, more than once group suggested templates as an intermediate form for algorithm exchange. There is a question about the likely savings of time or useful generality of the approach. It has been observed that a template, when offered to the community, should be accompanied by the source code of at least one actual instantiation of the template into a working program and a test suite of input/output data to verify correct ports. Whether this saves time or costs time is a question. From the writer's point of view, clearly

on top of the added effort of writing a working program, a carefully crafted abstraction of that program needs to be formulated at the same time. But it is not clear how much time would be saved by the user in conducting the port. For a template to have generality, it must be largely machine independent. But most of the work in achieving best performance for an application comes from making the detailed machine dependent decisions. A question arises as to how far the implications of a machine's structure and execution model must affect the algorithm in order for efficiency to be achieved. If it is too high, templates could become inconsequential. And yet, a programmer must start somewhere in the search for algorithms. The merit of program templates is still an open and important question.

12.12.8 Value of the Workshop and this Report

As a meta-comment, there were differing opinions about the degree of success of the workshop and the long term value of this report to the community. It was clear by the end of the workshop that there was much more to be done and that the workshop had at most scratched the surface. Some felt that the focus was too narrow, excluding important areas of concern or major alternatives to the specific system types being considered. Others felt that the focus was too broad, permitting insufficient depth in any single area. There were concerns that the list of invited attendees was biased toward certain viewpoints. Others felt that there were too many participants as it was and that fewer people, not more viewpoints, would have improved substantive interaction. And there was disappointment that a final answer had not emerged from this major collaborative enterprise. What really happened, of course, was that the goals of the workshop were satisfied and that the workshop was a success. Its intent was to bring together many experts across the wide array of related fields to consider the issues confronting effective use of massively parallel processing from the standpoint of software technology. The workshop was to identify research directions and set priorities. All of these things were accomplished. If some came away daunted by the task ahead, and disappointed that the workshop had not caused those challenges to be dissipated, then that too was an important outcome of the workshop and a contribution to the community. By the end, we learned that the need is real, the challenge is real, and this workshop had been an important beginning.

Chapter 13

Conclusions and Implications to HPC

13.1 Introduction

The Pasadena HPCC Systems Software Workshop proved to be an important and timely meeting. The radical change of supercomputer architecture from vector computing to massively parallel processing demands an equally fundamental change in programming methods and environments. Both the immediate needs of pathfinding computational scientists and the ultimate long term goals for a fully integrated distributed computing environment were examined in detail from diverse perspectives over the three day period. Approach, requirements, feasibility, and resources were considered in delineating the opportunities and challenges for system software research and development in the high-performance computing arena. The large number of high calibre participants guaranteed that the results of this workshop would be a critical and accurate appraisal of the current state of system software technology for massively parallel processing. The conclusions emerging from this intense exchange of ideas and experiences were both stimulating and sobering. Stimulating because they revealed the many possibilities opening up for the high performance computing community; sobering because they exposed the daunting difficulties that must be overcome and the substantial investment of resources and time required to attain the full promise of these systems.

A widely embraced long term HPC goal is to:

> Bring MPP system technology to a state of applicability and

> ease-of-use comparable to that of conventional contemporary computers while attaining sustained TeraFLOPS performance.

Recognizing the current disparity between hardware capacity and software capability, an immediate implication of the workshop sentiment is that a second short term goal must be aggressively encouraged as well, to:

> Provide immediate practical means for application of MPP capability to Grand Challenge problems.

The philosophy that emerged from the workshop was that synergism among system components and among professional organizations is crucial to success in achieving these goals. The complexity of current and future MPP systems demands that the knowledge and investment dedicated to implementing any part should be accessible to all developers and users. Coordination of R&D in system software and software tools is considered the only viable means of achieving the critical mass of talent and resources to address key problems in an acceptable time frame. Much thought was given to the tactics of capability integration, from the requirements for software tools interoperability to the mechanisms for collegial result sharing. The important pragmatic issues of funding, commercialization, application demands, and architecture limitations were considered throughout the workshop deliberations with particular emphasis on their influence on methods for achieving near term objectives. Underlying the urgency of the discussions was the prevalent dissatisfaction with the available tools for application programming on MPP systems. A second element of the adopted philosophy was referred to as gradualism or incrementalism, i.e., the small changes to existing tools that would produce near term useful results.

13.2 Summary of Key Findings

Major conclusions of the workshop are summarized here to highlight the issues that must be resolved to ensure the success of the HPCC Program. Their implications for potential follow-on actions are then discussed. These findings are presented in terms of the elements missing from, but required for, a fully effective programming and execution MPP environment. This summary is intended to provide the basis for prioritizing needs and specifying actions.

1. Today's MPPs have inadequate programming models for representing application parallelism and locality.

2. There is insufficient understanding of resource balance required for applications/systems at TeraFLOPS-scale processing. Also, the workshop had insufficient representation from the application community to provide a broad base of application experience.

3. Mathematical software libraries are largely unavailable for new computing systems.

4. Current MPPs provide limited user access to system state and behavior for debugging and performance optimization.

5. Disparate user interfaces and execution models across systems hinder portability.

6. Concerns about hardware/software reliability necessitate checkpoint/restart and exception handling.

7. Dynamic resource management/task allocation is usually left to the programmer; it should be achieved by integration of the user program, compiler, and runtime system.

8. Seamless heterogeneous parallel processing is measurably more difficult than managing ensembles of uniform processors. But heterogeneous systems are essential and techniques at least for overt explicit control are required in the immediate future.

9. I/O and file handling capability development are not keeping pace with processor performance evolution.

13.3 Strategy

A broad strategy encompassing the resources and shared goals of industry, academia, and government is required to address the needs exposed by the above findings in order to ensure success of HPCC. These needs arise largely due to the inability of vendors in the MPP arena to play the role conventionally attributed to computer companies. Specifically, computer manufacturers of production grade systems have provided complete resource management and programming environments for their customer base. But MPP vendors are delivering experimental systems with minimal environments because it is not clear what kind of software support is required or feasible to implement. Thus, the vendors require guidance from the users and

the computer science research community while the users require the largest possible systems to achieve research breakthroughs. This very healthy synergism will contribute to a rapid advance of HPC technology (this "friendly buyer" approach between the government HPC community and the vendors has considerable historical precedents and has been effective). But it requires unprecedented cooperation among all elements of the community. To paraphrase the sentiment of the workshop, now and in the foreseeable future, "business as usual" is synonymous with "going out of business". The following points make up the major tenets of an evolving strategy of cooperation that many consider to be the path to TeraFLOPS computing.

- Stimulate commercial delivery and support of all necessary system hardware and software components comprising complete MPP computing environments.
- Encourage active community dialogue to define technical challenges, identify approaches, and share results. Cross disciplinary efforts will be required to produce the necessary integrated environments.
- Foster coordinated multi-agency support for research and advanced development in areas of importance to shared goals.
- Research funding strategy should include many small research projects, a number of advanced development projects, and a few commercialization efforts.
- Establish criteria and methods for evaluating intermediate and end results of the program as well as means for feedback of assessments to research and manufacturer communities.

At the heart of this strategy is *sharing*: shared goals, shared resources, shared results. Implicit is loose coordination and cooperation across the community even as individual organizations quite rightly focus on their own self interest. In the scenario most likely to yield positive results, each organization emphasizes those aspects of the total problem appropriate to its own mission and shares the results with others. Redundancy can be wasteful—or it can be useful. Organizations should be prepared to leverage the work of others rather than building their own, despite the pernicious "Not Invented Here" syndrome. Beyond that, the community as a whole must discover where there is no coverage, and funding agencies should encourage appropriate entities to fill these gaps.

13.4 Implications for Applications

As much value as the Applications Working Group rendered to the quality of the workshop results, it is clear that more understanding of the requirements of applications on MPPs is necessary. Without this understanding, there is little certainty that near term software development efforts are adequately addressing the most important needs. An insidious trap occurs in a context such as this. We want to know how systems should evolve to support the needs of applications. We run experiments on current systems, imperfect as they may be. We select application problems that are suitable for execution on these existing machines and we collect measurements. What do we learn from this process? We usually learn how to build the *same* kind of machine we already have. Perhaps this scenario is slightly simplistic. But, the programs we are running tend to be those reasonably well suited for the *current* systems. One dimension of the problem is that the shape and form of applications studied need to be extended to encompass a broader class of problems.

Applications scaling problems are not well understood. As larger systems approaching TeraFLOPS capacity are realized using many more increasingly powerful processors, it is not understood how the relative balance of resources will change to best service these new scale problems. It is thought that larger machines will host larger versions of current problems. But, subtle scaling factors can introduce unexpected and undesirable performance degradation. The I/O requirements and secondary storage demands may scale linearly, much less, or much more; it's not currently known which. The critical path time of problems that are expanded in scale may increase. Increased memory latency on larger systems may alter cache behavior. With potentially larger address space, Translation Lookaside Buffers for virtual-to-physical address translation may experience more frequent flushing.

Because of the costs of task and data migration on current distributed memory MPPs, most application experience is with statically mapped and scheduled programs. If, however, program demand is data dependent and time varying runtime scheduling decisions may be necessary, perhaps on a continuing basis. The overhead requirements of such problems are not understood due to the limited hands-on experience with them. Yet, they may constitute an ever increasing portion of the total application program demand. The tradeoff between runtime load balancing and the inefficiencies load balancing may introduce is subtle and requires much more study.

The Applications Working Group concluded that optional system capabili-

ties need to be offered to the applications programmers with some indication of the performance cost likely to be incurred by their use. This conclusion reflects the nature of the current approach to application programming. Historically, applications programmers will do whatever is necessary to get the program running: dealing with all levels of system complexity if need be, even with the most primitive support tools. This approach will not serve the general community in the MPP regime. Even the biggest problems should be reasonably straightforward in their formulation for HPC system execution. It is incumbent upon the applications community to better articulate their requirements for programming environments and resource management so that systems designers can target the most critical needs in the immediate future. Where there are uncertainties due to lack of data, experiments should be formulated and sponsored for the express purpose of determining future resource demands.

Despite early hopes for automatic parallelizing compilers, programmer independent delineation and management of program parallelism is unlikely to be generally viable in the near future, at least with regards to execution on the current generation MPPs. Many important codes need rewriting, including those comprising heavily subscribed mathematical software libraries. This is not just because parallelism intrinsic to applications programs needs to be exposed, but also because alternate algorithms will be needed for more effective parallel execution. Unfortunately, the investment in labor required for such rewrites will be at risk because of the uncertain future of parallel programming languages and system architecture. While many contributors may be prepared to rewrite their codes once, it is not likely that they will be willing to do so multiple times as environments change. Without stable platforms ensuring longevity to which programmers can target new codes, rewrites are going to be limited to the immediate needs of the individual, rather than the broader needs of the community. However, not all serial subroutines will need rewriting. Given the prevalence of large grain tasks and the current inefficiencies of fine and medium grain parallelism, routines employed in the inner loops of application programs can probably be used *as is* since no practical gains would be achieved through parallelization.

13.5 Implications for Architecture

For the purposes of this workshop, an evident but unspoken assumption was that the MPP architectures to be used by the HPC community would be

vendor-provided systems derived from microprocessors developed primarily for the high performance workstation market. This assumption excluded SIMD architectures such as the CM-2 and MasPar-1. It also precluded the likelihood of MPPs with processors designed expressly for the purpose of engaging in collaborative/coordinated computation with other like processors, with the possible exception of the Tera Computer.

The SIMD issue touches on the question of diversity of execution models, and is important in its own right. The general issue of processor architecture vis a vis the requirements of parallel computation is at the core of the relationship between the HPC user community and MPP vendors. The costs and tradeoffs of processor architecture today make it difficult to justify investments in hardware and code for special interests that make up a relatively small part of the microprocessor consumer market. HPC falls into this category, although market share is expected to grow substantially through the decade. A second equally important aspect of this relationship, however, inhibits the realization of *symbiotic* processors, (i.e., processors designed to work together, forming a single holistic entity comprising many components). Simply put, the HPC community does not know exactly what it wants. It continues to experiment with various programming models, relationships among operating systems and runtime systems, compiler techniques, and I/O demands. With such uncertainty at all levels of parallel system utilization, inclusion of hardware support for any one approach imposes unacceptably high risk on chip suppliers.

While all these factors constrained workshop consideration of processor architecture as a valid degree-of-freedom for MPP evolution, several architecture-related issues were uncovered and discussed. Architecture support in MIMD architectures for the following functionality would be useful—or even necessary—if HPC system architecture continues on its current course.

- global shared reference space support mechanisms
- instrumentation for behavior monitoring
- synchronization mechanisms
- debugging; especially traps for timing conflicts/race conditions
- support for checkpointing and restart
- rapid context switching for latency hiding

- data stream (prefetching) access to reduce impact of cache miss

Some of these capabilities have already found at least simple representation in MPP architectures. The MultiKron instrumentation chip developed at NIST will be included in the Intel Paragon. The CM-5 incorporates a global synchronization network for efficient SPMD barriers. The BBN TC-2000 supported a shared address space. In general, however, the above capabilities are not available to the user. Some are not as high risk as others because they do not require additional circuitry on the processor chip itself, but rather, additional components at the system level. Increases in interprocessor communication have benefited greatly from new communication chips external to the processor. Still more experience will be required with runtime systems, and dynamic task scheduling, and data migration before it will be understood how architectural advances can best enhance efficiency.

One way in which early understanding of architectural alternatives can be acquired is by collaboration between the HPC applications community and architecture research groups in academia. A number of experimental systems have been devised that address these issues. But in many cases, architecture research groups do not have access to important real-world codes to test their ideas. Applications programmers have little interest in using such systems because the programming environments are ordinarily poor and the systems are often too small to provide competitive performance levels. Yet, these two communities need to be brought together if essential data is to be produced that can force architecture advances in commercial MPP systems.

13.6 Elements of a Future Course of Action

The general strategy defined earlier set overall directions for the HPC community to pursue development of environments for rapid and effective programming of massively parallel processing systems. The strategy outlined the principal initiatives and relationships to be established among the participating groups and agencies. While useful, it alone does not clarify the actions to be taken to ensure HPCC success. Some important elements of such a course of action are suggested here.

13.6.1 HPC Framework

Coordination and collaboration on a complex system such as MPP environments among distributed organizations require an agreed-upon *framework*

to which all components can be configured. Such an abstract infrastructure defines the general types of elements required to flesh-out the framework and the interfaces between them. If adhered to, the framework will ensure interoperability, portability, and source code longevity. It will also provide the intellectual basis for identifying critical *leverage points* that will yield the greatest benefit in the near term and on which funding agencies should concentrate resources.

The framework constitutes a system architecture. Many possible implementations can be realized for a given architecture and, indeed, many will be expected to evolve over time. But the replacement of any one component with a newer routine will work in concert with the remaining older components. Unlike most architectures, an MPP software environment framework must remain flexible to new functionality. Hooks to the system using the software bus concepts can permit new capabilities to be incorporated in much the same way that external I/O devices can be attached to a workstation backplane. But, more is implied. The evolving functionality may vary in the quantity and kind of information it provides to other components within the system. Not all data required by one element may be accessible from another. Components must be designed to interpret the nature of their connecting pieces, with protocols that adapt, and functions that configure themselves according to available resources. Thus, an MPP environment framework interface will be specified by a meta-protocol, i.e., an information exchange logical medium that negotiates its final form. Numerous examples of simpler versions exist in the networking and operating system community where installation and runtime techniques achieve compatibility among independently derived software components. The personal computer market place thrives on interoperability facilitated by de facto standards which permit upward compatibility as new capabilities become price affordable.

An all-encompassing MPP environment framework with its myriad interfacing meta-protocols might appear exotic, but pragmatic considerations dictate modest goals for the near term. A framework with interfaces that permit incomplete exchanges (i.e., some information is ignored and the lack of other information results in reduced functionality) is all that can be hoped for. But even this is sufficient to provide the commonality necessary for code longevity, interoperability, and extensibility.

13.6.2 Prerequisites

With the establishment of a target system software architecture, prerequisites for filling in the framework will need to be determined. Understanding, however, will be insufficient to immediately embark on implementation of all essential components. One result of the workshop was that not even all the requirements are understood at this time. Determining those requirements, or at least an operational subset, will provide the basis for defining a set of specifications for the framework, its logical interfaces, and the functionality of its constituents.

Not all the requirements known now can be satisfied with current knowledge and experience. Critical path research questions which impede the implementation of key components will be exposed. Such questions will provide target goals for near term studies. For those studies central to realizing a healthy production environment, encouragement should be provided by funding agencies.

It is likely that certain capabilities that are currently unavailable will be required to implement a full environment, and that these may be identified through the process of devising an HPC system software architecture. Identification of enabling technologies is yet one more key activity to be performed in the immediate future and is a direct consequence of the system software architecture derivation. For example, high bandwidth or low latency communication channels may not exist in a required form. Software engineering practices may not fully encompass the needs for development. These constitute enabling hardware and software technologies that will make practical the realization of effective MPP environments.

13.6.3 Primary Sources

The HPC community is a diverse collection of resources and talents which, if brought together appropriately, can accelerate the pace of high performance computing advances. The problems challenging such progress are as disparate and imposing as this research community is capable. The many demands implicit in realizing a fully functional production level MPP environment will be met with talent drawn from all organizations and agencies comprising the HPC community. Identifying the primary sources of capability to address the program needs is an important action. Talent and capability from government agencies, academic institutions, and hardware and software vendors can be matched with identified needs of the aggregate

program through appropriate mechanisms. These mechanisms include the High Performance Computing, Communications and Information Technology (HPCCIT) Subcommittee of the Federal Coordinating Council on Science, Engineering and Technology (FCCSET), professional organizations, consortia, workshops, colloquia, etc. A *requirements-capabilities* matrix can be developed to reveal potential means of addressing critical questions. This matrix will prove useful in focusing funding on crucial leverage points of activity for rapid results.

13.6.4 Working Groups

To monitor the progress toward deriving a complete and consistent system software architecture and achieving a production level MPP environment, working groups for each of the key components and focus points should be established under the sponsorship of the HPCCIT. Their members should be drawn from across the HPC community. To a significant degree, these working groups will be an extension and refinement of the workshop sessions that proved so effective in ferreting out the major issues. But, they will also reflect the component breakdown and context as prescribed by the emerging framework. These working groups will apply their detailed intimate knowledge to ensuring the quality of the infrastructure and ensuing component implementations. As occurred at the workshop, these working groups will have to interact to come to a fully satisfactory architecture which meets the needs and realistic constraints of a production environment.

13.6.5 Software Exchange

Software sharing is a theme central to attaining the stated goals. Mechanisms for facilitating sharing of information, software, and results are crucial. The HPCC National Software Exchange (NSE) experiment initiated by NASA is an important endeavor for delivering such global mechanisms to the HPC community in the immediate future. Early interest and participation by the community in this experiment will expedite its results and enhance the quality of the experience base.

13.7 Some Final Thoughts

This report has attempted to capture the full scope and breadth of issues considered and views expressed: usually strong, occasionally conflicting, but

always thoughtful. Because of the complexity of the issues and the systems they reflect, the report has endeavored to articulate the workshop conclusions from three different vantage points:

1. The raw and summarized representations of the findings from each of the seven working groups convened to explore the HPC system software question from their respective disciplines
2. A global synthesis of the major outcomes of the workshop to provide a single characterization of its results
3. An assessment of the implications of the workshop results to the high performance computing community and suggested direction of activities that it should consider pursuing to realize its near term needs and long term goals

The motivation for this effort was to provide a useful tool and point of reference for determining future actions by individuals, groups, and the HPC community. Its value is in its timeliness and breadth of representation. Its ultimate worth will be the initiatives it encourages.

We conclude here by revisiting the highlights and major results of the workshop from an action oriented viewpoint, briefly reviewing the critical findings from three perspectives:

- What we need,
- What we don't know, and
- What we have to do.

13.7.1 What we need

Qualitatively, we need performance, portability, and usability. Performance is foremost because it is the reason d'etre for the MPP class of system. Portability is essential for code longevity and scalability for easy migration to machines of different size and configuration. Portability demands interoperability between disparate and independently developed software subsystems residing on the same or separate networked hardware platforms. Usability encompasses programming models for representing parallelism and resource assignment pragmas, tools for observing and optimizing system behavior in

terms of correctness and performance, and robust recovery methods from system failures and software exceptions.

Specific capabilities urgently required are dbx-like debugging tools and profiling tools on at least on a per node basis. Means for checkpointing and restart are necessary for breaking up very long runs into manageable time frames and for robust recovery from all too frequent system failures. Message passing protocol standards and data parallel constructs in popular languages are required now for ease of programming and portability of source code. This involves to be agreed-upon extensions to Fortran and C (for system programming). Runtime service routines are required for efficient message passing, synchronization and context switching overhead, and (in the long term) load balancing. Operating system support and subroutine libraries are needed for convenient heterogeneous processing. Also, means for representing and performing parallel I/O must complement facilities for parallel computation.

13.7.2 What we don't know

From this workshop, the HPC community learned how much it has yet to learn. For applications targeted toward TeraFLOPS-scale MPPs, we don't know the relative balance of resources to achieve efficient execution. Programming models differ, sometimes markedly, as do the representation needs of parallel algorithms. We don't understand enough about which paradigms are appropriate to what application problems. Debugging parallel programs is greatly complicated by insidious timing errors and we don't know how to best capture, isolate, identify, and present these elusive bugs, let alone how to systematically eliminate them. Ultimate system behavior is a consequence of a collaboration between applications programmer, compiler, operating system, and runtime system. With the harnessing of algorithmic and processor parallelism on a massive scale, a change in emphasis in the roles and interplay of these entities is implied but the necessary balance is unknown.

13.7.3 What we have to do

In the broadest sense, we have to share; we have to nurture a sense of community; we have to standardize; we have to encourage and fund research; we have to evaluate and fund prototypes and commercialized key software subsystems; we have to experiment, evaluate, and learn; and we have to educate. At that point, the easy part is done. Because then we have to

work with the MPP vendors to evolve and enhance their architectures at the processor and systems levels to facilitate parallel computing so that we can begin the entire cycle all over again; but at a higher plane of efficiency, performance, and ease-of-use. Ultimately, we will approach the threshold of sustainable TeraFLOPS capability, perhaps wondering *what the big deal was?* because then it will have become easy.

More specifically, work to fill urgent needs such as debuggers and profilers has to be performed immediately. Longer term work in both areas needs to be initiated toward global system handling. Checkpointing and restart mechanisms need to be incorporated into MPP operating systems immediately. Inter-computer hooks for distributed inter-process communication need to be standardized soon and included in all networked computing systems.

A framework needs to be defined for system software supporting HPC systems. This system software architecture will provide an infrastructure specifying functional characteristics and interface protocols essential for interoperability and code longevity. We need to identify the critical components of this system software architecture and apply funding to the leverage points that will yield the greatest return for the community. Also, prerequisites for achieving the functionality of the various components must be determined, including research questions and enabling technologies. Funding agencies need to ascertain where their common interests lie and pool their resources to gain greater leverage on the problem space by concentrating mass. Evaluation criteria, benchmarks, and measurement models and techniques need to be devised to permit meaningful comparative studies that can be shared and appreciated by the community as a whole.

Methods for sharing information, software, results, and ideas are needed. These methods include automated global access to databases, and organizations within the community to oversee its evolution and monitor its successes. Working groups need to be established to monitor development of critical elements within the system software architecture. Follow-on workshops such as this one need to be organized to periodically sample the state of progress, knowledge, and consensus within the community for mid-course corrections. And we need methods to foster closer associations of the applications community and computational scientists with the computer scientists committed to fully realizing the manifestation of the HPCC system software architecture.

Appendix A

Working Group Position Viewgraphs

This appendix reproduces the slides that were presented by each working group on the first day of the workshop.

A.1 Applications Requirements

Presentation by Geoffrey C. Fox (Syracuse University)

What will Application Report look like?

- General Remarks
- Loop over several application areas
  - Requirements
  - Case histories in Parallelism
  - *N* Volunteers
    - Special Features of Codes
    - Performance needed
    - Functionality needed — SIMD v. MIMD
    - I/O — (a) visualization, (b) databases
  - Capture some of issues in benchmark set of "kernels"

  - What benchmarks cover reasonable set of algorithms/issue to certify software?

- Interaction of application areas with other working groups — six volunteers

Why are Application—Software Linkages Important?

1. Technical

   "Compiler" or "user" or some mixture of these, needs to "understand" problem structure to be able to parallelize it.

   This knowledge needs to be more precise than for other (non-parallel) architectures—penalty for failure high! (performance degraded by a factor of 1/Number of Processors . . .).

   This will be addressed at length in other working groups, e.g. by high performance Fortran directives, etc.

2. "Political" — Market Issues

   Parallel computing will only succeed if application scientists buy them!

   What are user desires and priorities?

Performance	**Portability**
• Physics graduate student wanting to get Nobel Prize with best QCD (Quantum Chromodynamics) Simulation	• Fortune 500 Company developing million-line code to last 20 years
• Redo many times	• Redo once • Real cost is critical

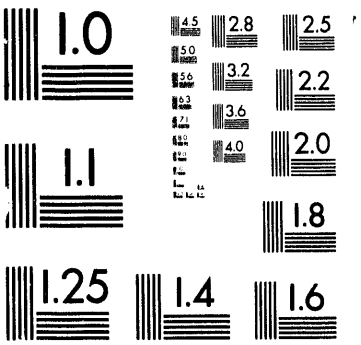
We have learnt about importance of edge over area!

edge: Interface between Parallel Computing → Real World of Computing

area: Software internal to HPCC

Need more attention to "edge software"
Surface Tension effects are holding back parallel computing?
Need to integrate Parallel Computing with larger fields of

- Personal Computers
- Workstations
- Distributed Computing
- DARPA SWTS


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0

3 of 3

Many Application Areas

- e.g., BMC^{3}IS or Manufacturing Systems
- Integrate components
  - AI
  - Science and Engineering
  - Real time constraints
  - Visualization (VR)
  - Databases

  naturally supported with different software paradigms
  - Application Specific Languages
  - *Data
  - *Functional
  - *Object

  *Parallelism
- e.g., need Linda, and Express, and Fortran D **not** either/or
- Application scientists don't often speak to computer scientists
- Even worse, one application group doesn't often interact with groups in a different discipline
- Adaptive multigrid developed for aerodynamics not used (yet) for study of colliding black holes
- Fast multipole method developed for astrophysics not used (much) in chemical molecular dynamics
- Neural network algorithm developed for vision not used in GIS
- Need to integrate computational science techniques between fields

A.2 Compilers and Languages

Presentation by Ken Kennedy (Rice University/CRPC)

Compilers and Languages

- User needs and priorities
- Current systems: strengths and weaknesses
- Paradigms and prototypes
- Standards
- Compilers

Needs

- Standard language
- Avoid low-level details
- High efficiency
- Powerful tools
- Libraries with standard language interfaces
- Protection of programming investment

User Priorities

- High performance
- Minimum effort
- Machine independence
- cost/performance

Problems with Current Systems

- Low level
- Machine specific
- Varied paradigms
- Low reliability
- Poor relative performance

Machine independence

- Current systems expose architecture
- Cost of reprogramming
- Lesson of vectorization
  - dusty deck problem unsolved
- Need: vehicle for portable parallel programming

Strengths

- Programming systems available
  - explicit message passing
  - Fortran 90 (data parallelism)
- Many programming systems available
- Autoparallelization for shared-memory

Paradigms

- Data parallelism
- Task parallelism
- Object parallelism
- Instruction parallelism

Research and Development I

- Data parallelism
  - Fortran D
  - Fortran 90
- Task parallelism
  - Schedule
  - PCN
- Object parallelism
  - C++

Research and Development II

- General shared memory
  - PCF Fortran
- Simulated shared memory
  - Linda
  - Virtual shared memory
- Functional languages
  - Sisal

Technology Development

- Three System Levels
  - University prototype
  - Usable prototype
  - Commercial product
- Problem: step from university to usable prototype

Compiler Challege

- Mapping to parallelism is complex
  - Deep program knowledge
  - Broad (interprocedural) knowledge
- Use mixed types of parallelism
- Compile good code for the nodes

Standards

- Promote machine independence
- Encourage manufacturers
- Essential to users
- Ensure interoperability of parts

Summary

- Exploration of paradigms
- Research on complex compilers
- Technology development of selected prototypes
- Standardization

A.3 Computing Environments

Presentation by Reagan Moore (San Diego Supercomputer Center)

Computing Environment Topics

- Data Support Systems
- Communication Systems
- Heterogeneous Computing Environments

Working Group Agenda

- Identification of system software issues and paradigm shifts for the next five years
- Identification of current software infrastructure and research efforts
- Identification of key technologies for future development

Computing Environment Hierarchy

- Parallel Computers
  - Distributed memory
- Metacomputers
  - Heterogeneous computing platforms within a computer center
- Metacenters
  - Multiple computing centers within a national machine room

System Software Issues

- Performance
  - Data distribution
    - Data/CPU no-locality
  - Appropriate resource selection
    - Functional decompositions
- Data Support for TeraFLOP Computer
  - Large files
  - Location transparency
- Data Distribution
  - Decoupling data space from CPU space
  - HPF assumes tight coupling
- Data Access
  - Move process to the data — RPC
    - Assumes CPU resources available
  - Move data to the process — use supercomputer
    - Always faster for sufficiently complex algorithm

Perspectives

- Application Performance
  - Shortest wall clock time for job to complete
- System Performance
  - Maximum utilization of resources

Data Support Systems

- I/O Scaling
- Archival Storage
  - Network attached peripherals
  - Third party data transfer
  - Caching file systems
- National File System
  - File sharing between file systems
- Scientific Database Interfaces
  - Data format standardization
  - Data format conversion tools
  - XDR/data compression tools

Examples

- Data Storage Requirements
  - Large file access
  - Response time for Gigabyte files
  - Location transparency
  - National file system
  - Petabyte storage archives
  - IEEE MSSRM — distributed
  - archival storage system
  - Associative data reference
  - Scientific database systems
  - Knowledge about the data is more important than the data

Data Storage Questions

- Is there a "standard" data hierarchy?
  - Distributed memory
  - Ram disk — SSD
  - Local disk
  - Remote disk
  - Tape robot
  - Scientific database
  - Metadata file systems
  - Archival storage system
- Can the amount of data be minimized?
  - Application shift to direct methods and code reexecution rather than data storage.
- Will future technologies make data storage devices cheaper than CPUs?
  - Holographic storage
  - Chemical/optical storage
- Should data storage be error free?
  - Bit error rates of 1^{-12} are too high

Communication Systems

- Gigabit/second Communication Links
  - High-speed protocols
  - Multimedia protocols
- Communication Media Standardization
  - HIPPI, fiberchannel, SONET
- Distributed Application Control
  - HIPPI RPC mechanisms
  - Synchronization of distributed programs
- Data Privacy

Communication Questions

- High-speed Transfers
  - Bit error rates of network (assuming no packet loss) same as storage device. Should error correction be a property of the data instead of the network?
- Data Privacy
  - Is knowing when data has been compromised sufficient? Authentication mechanisms relying on tickets have time windows during which encryption can be broken.
- Communication Hierarchies — What Comes Next?

Ethernet	T1
FDDI	T3
HIPPI	SONET
??	??

Heterogeneous Computing Environment

- Programming Support Environments
  - Decomposition metrics
  - Coupling metrics
- Resource Management
  - Remote system status
  - Error handling
  - Accounting
- Resource Control
  - Load balancing
  - Job scheduling

Heterogeneous Computing

- Should parallel computers be treated as CPU limited resources or as memory limited resources?
- Amdahl's Law
- Maximum number of processors usable for given efficiency is

$$N = \frac{1/E - f}{1 - f}$$

E = efficiency

f = parallelization fraction

E	f	N
10%	50%	19
	90%	91
50%	50%	3
	90%	11
	99%	101
90%	50%	1
	89%	2
	98.8%	10
	99.4%	20
	99.9%	112

Parallel computers are multiprocessors

Heterogeneous Computing

- Super-linear Speedup
  - Functional decomposition to distribute computation between multiple heterogeneous computer platforms
  - Implementation through:
    * Application level
    * Explicit decomposition
    * Program support environment
    * Compiler/Libraries
    * RPC servers
    * System level
    * Job scheduling

A.4 Mathematical Software

Presentation by Michael T. Heath (University of Illinois, NCSA)

- What will a scalable parallel math software library look like, and how will it be used?
- How will it differ in structure and content from conventional mathematical software libraries?
- Will portability be achievable at a reasonable cost in performance (or at any cost)?
- What standards will be necessary to ensure portability, to whatever degree is possible?
- What is the meaning of scalability, and is it truly achievable in practice?
- Can the complexity of parallel architectures and algorithms be hidden from users without sacrificing performance, and is a math software library a suitable vehicle for attempting to do this?
- What are the implications of parallelism for the accuracy, stability, and convergence of numerical algorithms?
- Can we, or should we, expect exactly repeatable results, given the potential nondeterminism introduced by parallelism?

Enabling Technologies

- Architectures
- Algorithms
- Data Structures
- Programming Languages
- Compilers
- Operating Systems
- Communication Systems
- Partitioning, Mapping, and Scheduling
- Software Tools

Applications Development

- Motivation and Market for Math Software
- Real Software Needs
- Operational Context

User Interface

- Hiding Complexity from Users
- Problem Specification and Representation
- Data Management
- Problem Solving Environments
- Mathematical Research
- Graphics and Visualization

Portability

- Scalability
- Flexibility
- Standards
- Computer Arithmetic
- Nondeterminism

Software Engineering

- Programming Paradigms
- Structure of Modules and Libraries
- Encapsulation
- Communication Primitives
- Adaptable Granularity
- Reusable Templates
- Hierarchical and Heterogeneous Systems
- Testing and Validation
- Error Handling and Reporting
- Instrumentation and Performance Analysis

A.5 Operating Systems

Presentation by Robert L. Knighten (Intel Supercomputer Systems)

Basic Assumptions

- Heterogeneous Networks of Computers
- Massively Parallel Processors with Physically Distributed Memory
- POSIX Compliant Operating System

Major Areas

- Memory Management
- Message Passing
- I/O and File Systems
- Support for Debugging and Performance Monitoring
- Resource Management
- Job Scheduling

Problems and Needs

- Performance
- Ease of Use
- Portability of Code, Developers, and Users
- Reliability
- Recoverability
- Debugging (including performance monitoring)
- Resource Management
- Large Scale I/O
- Memory Management

A.6 Software Tools

Presentation by Joel Saltz (University of Maryland)

Major Areas

- Performance Tools
- Debugging Tools
- Runtime Support for Irregular and Adaptive Problems
- High Level Programming Environments
- Tools Designed for Group Environments

A.6.1 Performance Tools

Application developers want tools to help make decisions involved in tuning performance

- Incremental changes to optimize performance in preexisting code
- Use performance predictions to guide application code development

Predict performance of code on new or on scaled-up hardware

- Most code development will probably be carried out on "small" local multiprocessors
- Codes may be designed with next generation hardware in mind
- Vendors could use performance projections to predict consequences of architectural changes

Many interacting factors influence performance of programs on scalable architectures

- Processor Architecture
- Time delays associated with accessing data residing in various places (e.g., cache, local memory, off-processor memory)
- Degree to which communication latency can be hidden
- Data reference patterns resulting from choices made in partitioning data or work
- Program Transformations

Performance information should be presented in ways that relate to users' language and programming tools

- Fortran D/High Performance Fortran compilers allow users to control data distribution
- Feedback is needed on performance impact of a programmer's choices about data or workload partitioning
- Compilers may sometimes attempt to optimize partitioning —
  - user needs to know what compiler has decided along with performance impact of compiler's partitioning decisions
- How to best display performance impact of partitioning?

Current User Priorities

- Source-level Debugging
- Debugger checkpointing to perform debugging experiments in the middle of long program executions
- Support for isolating transient errors (data races, indeterminate matching of synchronization primitives, send/receive matching)
- Hooks to apply scientific visualization tools from debugger without prior arrangement
- Watchpoints that pinpoint changes to a data value

Status of Systems Software

- Most commercial parallel debuggers are ill-equipped to address issues peculiar to developing parallel software
  - little support for examining and experimenting with issues related to communication and synchronization
  - no commercial debuggers support debugging of code with transformations that provide MIMD parallelism
  - no commercial debugging tools adequately support pinpointing of causes behind timing-dependent errors
  - inadequate support for watch points
  - a few debuggers provide support for lightweight instrumentation useful for low overhead conditional break points

Priorities for the Future

- Tools used to tune application codes need to be interactive
- Users need to get feedback on consequences of possible decisions concerning data layout, workload partitioning, loop restructuring
- New visualization methods needed to capture the effects of user decisions on interaction between data and workload partitioning and the performance of various levels of memory hierarchy
- Instrument to minimize impact of monitoring on performance. Possible that hardware support will be required

A.6.2 Debugging Tools

Goal — provide a level of debugging support to make development of a multiprocessor code as simple (or difficult) as development of a sequential code.

- Programming paradigm has great influence on errors introduced, required debugger functionality

  Explicit shared memory program:

  - complex synchronization conditions, data races, etc.

  Message passing program:

  - examining message patterns, pairing send and receive calls

  High Performance Fortran program

  - relate execution state back to source program specifying sequential thread of control constrained style may reduce errors

Applications developers want to avoid time consuming cycle of running and recompiling code when they tune performance.

- Possible Methods:

  1. A priori performance predictions based on compiler analysis of program text
  2. Performance tool generates program that would allow user to interactively explore wide range of partitioning alternatives
  3. Performance tools could checkpoint save state and repeatedly restart using modified program

Status of System Software

- Tools exist to monitor execution, collect performance related data, present data on (distributed memory architectures)
  - data presented from viewpoint of hardware and operating system (processor utilization, idle time, pattern of message traffic)
  - users can arrange to indicate when tasks are active
  - ability to relate performance data back to source code
  - but information about impact of data, workload partitioning must be inferred
  - neither compiler nor performance tools have shared name space

Priorities for the Future

- Support debugging of radically transformed programs
- Hooks for data visualization to provide users support in verifying correctness
- Support for isolating transient errors

Develop programmer support to establish confidence (or lack thereof) in the application as a whole.

- Verifying "correctness" of multidisciplinary parallelized application code

A.6.3 Runtime Support for Irregular and Adaptive Problems

Programs associated with many applications involve complex or irregular data access patterns.

To solve:

- Partition (dynamically) data and work
- Coordinate irregular patterns of interprocessor data movement
- Manage storage of, and access to, local copies of off-processor data

Applications developers want to be able to get good performance from their irregular applications **without having to deal explicitly with problem partitioning, load balancing, communication scheduling.**

Status of Software Systems

Distributed Shared Memory

- Move fixed sized data blocks (pages or subpages) in response to data access patterns
- Easy to use but inefficiencies can result when data access patterns do not conform to memory layout of pages
- Kai Li, Thierry Priol, Kendall Square Research

Linda

- Supports shared tuple space data accessed from tuple space in variable sized chunks
- Programmer must carry out preprocessing to obtain judiciously defined tuples

Fortran with irregular distributions, compiler embedded mappings

- Same advantages as any High Performance Fortran type language—user is protected from partitioning, load balancing details
- Several rudimentary compiler prototypes but no fully operational compiler
- Cannot (as yet) handle many important classes of irregular and adaptive applications.
  - Fortran D project, superb

Procedures acting on irregular distributed array reference patterns

- Procedures called from application code to partition data, work and to generate optimized communication schedules.
- Manage storage of, and access to, local copies of off-processor data
- Limited in scope of irregular problems handled
  - Fastgraph, PARTI

Problem solving environments for classes of sparse, irregular, adaptive problems

- DIME — unstructured mesh CFD

Priorities for the Future

- Methods to efficiently implement on scalable architectures the wide range of irregular and adaptive computations
- Transformations and runtime support to make it possible for compilers and problem solving environments to efficiently incorporate these methods
- Use high performance Fortran, problem solving environments, etc.
  - choose between strategies to be used in data and workload partitioning, communication optimizations, etc.
  - avoid having to get involved with details of dealing with low-level support for irregular problems

A.6.4 High Level Programming Environments/Tools for Specific Application Areas

Problem solving environments

- Big win for personal computers in non-scientific applications
- Mathematica, matlab, maple examples in the math/engineering/science domain
- Very little for scalable architectures

Tools should hide any networking

- Different portions of code should run transparently on different platforms

Standardized Object Oriented Libraries for specialized application areas

A.6.5 Tools Designed to be Used in Group Environments

Basic software engineering:

- Portions of complex project will be numerous and interdependent
- Coordinate dependencies
- Release level management
  - version control
- Test case management
  - log in and automatically use test cases when verifying software correctness

A.7 Visualization

Presentation by Lewis W. Tucker (Thinking Machines Corporation)

Visualization in High Performance Computing

- Goal: integration of visualization into large scale computations
- Why needed?
  - communication/understanding of results
  - application development and debugging
  - interpretation of results
  - insight

What do Users Want? They want it all . . . . .

- Real-time animation
- Volume visualization
- Polygon rendering
- Wireframe modelling
- Ray tracing
- Iso-contouring
- On their workstation
- Video-tape production
- Image archiving
- Graphical user-interface
- Data plotting capability
- Numbers behind the image

- Gigabytes of mass storage
- Fly through their data
- Interactive rendering
- Standard image formats
- Interoperability with desktop publishing
- Easy to use without programming or expert knowledge of graphics

. . . . . and they want it now.

Choices facing developers of visualization systems

- Emphasize distributed visualization systems to spread work between high performance supercomputers and workstations?
- Develop graphics libraries for parallel machines?
- Improve ease of use and production of publication-quality imagery?
- Target users remote of X Windows terminals?
- Promote standards for exchange of scientific datasets?
- Integrate video production capabilities?

Current Technology Trend

- Graphics workstations are dropping in price and increasing in performance
- Network bandwidths are slowly increasing in speed
- RAID technology promises high bandwidth mass storage capabilities
- Standards are emerging for exchanging graphical information
- Software systems for distributed visualization are gaining in popularity

Position Statement

- Ultimate target is the scientist's desktop.
- Systems must be both scalable, adaptable, and integrated into an environment of high performance systems, large mass storage units, graphics workstations and networks of various kinds.
- Must leverage as much "workstation" software as possible while building organizational structures for sharing and reusing software and systems.
- Recognize that HPC visualization systems have significantly different requirements in terms of dataset size, bandwidth, and interactivity.

Current Technology Trend continued

– but –

- The datasets produced by Grand Challenge problems on massively parallel systems are increasing in size at a much faster rate
- Network bandwidth in the typical computing environment is limited
- Visualization software and algorithms for parallel machines are still immature

Appendix B

Working Group Findings Viewgraphs

This appendix reproduces the slides that were presented by each working group on the final day of the workshop.

B.1 Applications

Presentation by Geoffrey C. Fox (Syracuse University)

Ground Rules

"Naive User" $\longrightarrow$ "Talented but inexperienced in parallel computing user" $\longrightarrow$ HPC Systems

This will be methodology for developing HPC applications in both

1. Grand Challenge Teams (NASA, DOE, NSF . . .)
2. Industry

$\longrightarrow$ **GRADUALISM WILL WORK**

Systems Software should try to help dedicated talented users

- Our findings come from a small unrepresentative group—need to be validated, e.g., by Grand Challenge teams which can confirm, deny or quantify our findings
- There is a spectrum of needs which can trade off between

Performance	.....	Portability
"Small" evolving codes		Large Institutional codes

- HPC is currently a small fraction of world computer market
- HPCCP is focussed on Grand Challenges
- Concern that need more coordination between HPC software and other efforts, e.g., software engineering

Template Codes

- Alternate to Libraries
- Communicate generic applications (parallel FFT ... Adaptive multigrid) between disciplines and between mathematicians and application
- Clear Documentation
- Test Cases, i.e., template should run on some computer
- Exact details, e.g., language, not so important (C++, Fortran, Pascal)
- Significant educational value courses → training sessions
- Canned Packages bound to be
  - either too general
  - too specialized

Need Standards

as in

- Message Passing
- High Performance Fortran where we have application experience

  But need to do research and gather experience in

  - Parallel I/O
  - POSIX Compliance

  What are the Application Requirements?
- Need world view and agreed terms?

$$\begin{aligned} \text{Sequential Process} &= \text{Program} \\ \text{Parallel } N \text{ Processes} &= \text{Program} \end{aligned}$$

- Need Agreed (set of) Paradigm/Virtual Machines

Gradualism

Far more emphasis on modest, robust software than on flashy complete tools/compilers/math libraries/operating systems/environments

We need to quantify needs but include

- Node debuggers
- Node compilers
- Porting tools ← need to analyze requirements

  e.g., as well as HPFortran compiler, need dependency analysis tools to aid user parallelism

Not clear that we need a

- Seamless heterogeneous metacenter before we have experience with
- Seamless homogeneous single machine

Interaction with Environment Group

- We do not know application requirements for ratio
  - CPU Power
  - Power
  - Memory
  - Disk
  - Archive Storage
- Not clear if realistic to extrapolate to TeraFLOP performance

Application Requirements

- Don't Ask us what we want as we will take anything you offer
  Rather, describe allowed trade offs, e.g.,
  - Virtual Shared Memory
  - Checkpointing (automatic)
  - Treatment of Races in Debugging
  - Zero Cost — Yes!
  - 10% Cost — No.

B.2 Compilers and Languages

Presentation by Ken Kennedy (Rice University/CRPC)

Outline

- User needs
- Priorities
- Strategies
- Research emphases

Needs

- Knowledge base about applicability of paradigms to problems
- Standardized languages
  - portable, vendor-supported, scientific, scalable
- Avoid low-level details
- High efficiency
- Language and paradigm interoperability
- Powerful tools
- Libraries with standard language interfaces
- Protection of programming investment
- Higher-level (domain-specific) languages

Program Priorities

- High performance
- Portability
- Usability
- Locality management
- Paradigm understanding
- Paradigm integration

Strategies

- Technology Development
  - research prototypes
  - advanced development prototype
  - commercial product
- Evaluation standards for all projects
- Standards to enhance portability
- Infrastructure development
  - software exchange via software repository
    - contract development
    - advanced development projects
  - test case repository for support of evaluations

Research Areas of Emphasis

- Robust environment for successful high-performance languages
- Advanced Development for qualified emerging research languages
- Optimize critical compilers for peak performance
- Define language features or compiler extensions for known critical weaknesses
- Continue funding of promising basic research efforts
- Paradigm evaluations (head-to-head)

B.3 Computing Environments

Presentation by Reagan Moore (San Diego Supercomputer Center)

Issues

- Understand minimal application requirements for computing environment
- Understand application scaling requirements for I/O bandwidth and I/O caching
- Understand application data integrity requirements
- Establish goals for HPCC software technology for next five years

Initial Requirements for porting/rewriting codes for parallel computers

- To Port Codes
  - Standards for parallel programming support environment
- To Rewrite Codes
  - Standard for parallel I/O
  - Standard for message passing
  - Working parallel programming support environment
  - Unix tools
  - Accounting

I/O Scaling Methodology for Predicting Archival Storage Requirements

- Data distribution between local disk/archival storage
  - Estimates from analysis of application-specific memory bandwidth scaling as a function of CPU power
    - Chemistry
    - Quantum Chromodynamics
    - CFD
    - Weather
    - Visualization

Data Support Systems

• I/O Scaling	(Research - 1)
• Archival Storage	(Vendors - 2)
• National File System	(Government lead - 5)
• Scientific database prototype	(Research - 5)
• Data format standardization	(Users - 2)
• Data Compression	(Research - 5)
• Data Privacy	(Research - 5)
• Data Integrity	(Research - 1)

- Archival Storage
  - Current file sizes — 200 MB
  - Future file sizes — memory size or 30+ GB
    - File retrieval time determines the acceptable I/O rate
- National File System
  - Current efforts AFS — CMU
  - Future efforts with DFS
  - Equivalent of ARPAnet

Communication Systems

- Gigabit/second networks (Government lead — 5)
  - Requirements for NREN
    - Latency tolerant applications
    - Bandwidth reservation
    - Levels of service
    - Data integrity
    - Packet switched vs circuit switched
- Computer center backbone I/O requirements
- Data Privacy — performance cost (Research — 3)
- Network algorithms (Research — 1)
- Standard data integrity mechanism for memory/network/storage (Research — 2)
- Parallel I/O across networks — standard file format

Computer Center I/O Requirements

- Upper limit < memory bandwidth
  - Local disk access
  - Checkpointing
  - Performance statistics
  - Archival Storage
- Bandwidth supportable by Operating System

Heterogeneous Computing Environment

- Programming support environment (vendor — 5)
  - APPL/ISIS/PVM Hence/LINDA/IPS-2
  - Requirement for application metadata
    - Decomposition — granularity
      - Automatic data parallel decomposition
      - Language drive — C++/LINDA/Object Oriented
      - Coarse grained decomposition
      - Homogeneous metacomputers
- Meta-manager (Research — 5)

Resource Partitioning for Meta-Centers

- Local Resources — controlled by run-time scheduler
  - Interactive users
  - Production usage
- Meta — Resources
  - Production usage
    - Tennis court reservation by meta-manager

Meta — Managers

- Existing Systems
  - DQS (Florida State) Heterogeneous queue management
  - NQS Exec (NASA) Manage RS 6000 cluster
  - Condor (University of Wisconsin) Workstation cluster management with job checkpointing queuing, and remote I/O access
- Future requirements
  - Accounting
  - Scheduling
  - Resource management
  - Load balancing
  - Remote status
  - Security

B.4 Mathematical Software

Presentation by Michael T. Heath (University of Illinois, NCSA)

Questions

1. Who are the users, and what do they want/need?
2. What is current status of math software?
3. What are priorities for future research?

Math Software Issues

1. Applications
2. Algorithms/Data Structures
3. User Interface
4. Portability/Scalability
5. Software Engineering
6. Enabling Technologies (e.g., compilers, languages, operating systems, architectures)

Plausible Paradigms for HPC Math Software

1. Minimum change to status quo
2. Reactive servers with byte stream interface
3. Problem solving environments
4. Reusable templates

Identified Needs

- Greater vertical integration
- Near-term results
- Rethink both library structure and user needs
- Availability of source code
- Education

B.5 Operating Systems

Presentation by Robert L. Knighten (Intel Supercomputer Systems)

Basic Issue

- The fundamental goal of a high performance computing system is to deliver high performance. The operating system must facilitate rather than impede this goal.

The Other Issues — I

- File Systems and I/O
  - Define basic parallel I/O modes
- Debugging and Performance Monitoring
  - Checkpoint/modify/restart
  - Control of event tracing

The Other Issues — II

- Heterogeneity
  - Priority — transparency
- Job Scheduling and Resource Management
- Memory Management
- Exception Handling

Checkpoint/Restart and Job Swapping

- The Hot Issue
- No Production Facilities for MPP systems
- Research Prototypes Under Development
- Research Needed Particularly for Networks

B.6 Software Tools

Presentation by Joel Saltz (University of Maryland)

Short term requests

- DBX
- Simple profiling info

Debugging

- Test coverage
  - how do we know whether program is correct?
  - what aspects of this can be automated?
- Debugging heterogeneous systems
  - uniform remote debugger interface
  - more ambitious ideas
- Post-mortems
  - what information to keep
  - where to keep (disk, memory)
- How compulsive do we need to be about timing information
  - sequences of events
  - real vs. virtual time
- Decent vendor support for the basics!!
  - funding agencies should insist on "minimal" level of support
  - dbx running on each node

- Different programming paradigms introduce different kinds of bugs
  - shared memory (race conditions)
  - distributed memory (send/receive matching, buffer overflow)
  - High Performance Fortran (single thread is advantage, but watch out for trapdoors)
- Abstraction of program state
  - debugging for 1000s of processors
  - visualization
  - programmed probing into program state
- Trace errors back to source
  - radically transformed code
  - link to compiler essential

Performance Tools

- Some performance issues
  - how would performance change if program was modified (e.g., change data distribution)
  - how good {speed, efficiency, utilization} is program X making of parallel machine Y
  - how would performance of program change if {problem size, #processors, memory access speed} changed?
- Types of performance tool users
  - domain experts
  - application software developers
  - system software developers
  - hardware developers
- Hierarchy of performance considerations
  - questions asked
  - information displayed
  - trace (information to be displayed)
  - invasiveness of measurements
  - hardware monitoring/instruction level simulation

Performance Tools II

- Layered analysis
  - memory hierarchy effects
    - * cache effects
    - * communication delays
- Role of Compiler
  - relating performance to program (portion of program text, data distributions, etc.)
  - compiler instrumentation to produce traces
  - compiler generates info for scaling predictions
  - performance predictions/measurements used as compiler feedbacks
- Display performance information — abstraction of program state
  - animation
  - representation in terms of programming paradigm

Additional Issues

- Tools for Transforming Source Code
  - Facilitate design of useful tools
    - * compilers
    - * performance tools
    - * ADIFOR
    - * runtime compilation
    - * debuggers
- Support of shared address space
  - specifically irregular problems but raises general issue
    - * HPF compiler
    - * runtime compilation support (Parti, Fastgraph)
    - * distributed shared memory
- Graphical User Interfaces/Problem solving environments
- Adaptation of basic software engineering to HPC arena

B.7 Visualization

Presentation by Lewis W. Tucker (Thinking Machines Corporation)

Motivation

- Select set of Grand Challenges (output and resolution):
  - Geophysical Apps.
  - CFD
  - Plasma Physics
  - Remote Sensing

TeraFLOP Computations Will

1. Generate terabyte to 100 terabytes datasets
2. Data streams of four to 400 MB/s
3. High Resolution stereo animation

   1 stereo pair at 1 K x 1 K pixels generated from 512^3 data requires 1 GByte data +17 GFLOPS processing on TeraFLOP Machine $\longrightarrow$ 2 K x 2 K pixels from 1000^3 data

Need — New Visualization Software

To support: browsing, animation, interactivity, distributed, extensible, parallel

Goal: distributed parallel heterogeneous visualization framework

Heterogeneous Computing Environment

- TeraFLOPS machine, large fast disk (RAID), tape archiving, hardcopy output devices, workstations, HIPPI framebuffers

Near Term Development* Needs to Focus on Two Main Problems:

1. Performance of Graphics Infrastructure 30%
2. Performance and Functionality of Graphics Libraries and Paradigms 70%

Infrastructure Items:

1. Parallel I/O
2. Fast and Large Filesystems

Other technologies the market is addressing (displays, VR, multimedia) IDEALLY **OS and Systems** Projects WOULD PROVIDE solutions. However, in practice solutions are inadequate!! To a lesser extent, MPP C needs attention. 90% of new visualization software is in C.

*5–7 years

Parallel I/O for Visualization ideally would include:

- Intelligent negotiation of:
  - Bandwidth
  - Data format
  - # Parallel data channels
  - Grouping and ordering of data blocks
  - Data compression
  - Buffer sizes
  - Use shared memory when possible

Module Functionality (users, developers)

- Parallel versions of (scalability is important)
  - Volume rendering
  - Polygon rendering
  - Contour surface or line generation
  - Tracer particles, smoke injection, ribbons
  - Grid generation
  - Interpolation
  - Animation

Module Structure

- Modules built from layers
  - Math libraries
  - Low-level graphics primitives (vendors)
  - High-level graphics primitives (tool developers)
- Need parallel version of everything

Summary

- Current problems are system infrastructure!!
- Visualization resources currently are solving systems problems $\Longleftarrow$
- Need MPP C, interfaces to Fortran D and Fortran 90
- Parallel I/O. Large fast file system (vendors, OS)
- Distributed graphics framework (vendor, gov, users)

Appendix C

Attendees List

Lee Holcomb, Workshop Chair (NASA)

Organizing Committee

Paul Smith, Chair (NASA)

Mel Ciment (NSF)

George Cotter (NSA)

Fred Johnson (NIST)

Gary Johnson (DOE)

Carl Kukkonen (JPL)

Fred Long (NOAA)

Jacob Maizel (NIH)

Joan Novak (EPA)

William Scherlis (DARPA)

Program Committee

Paul Messina, Chair (Caltech/Jet Propulsion Laboratory)

Jack Dongarra (University of Tennessee/ORNL)
John Dorband (NASA Goddard)
Brian Ford (NAG)
Geoffrey Fox (Syracuse University)
Mark Furtney (Cray Research)
Mike Heath (NCSA/University of Illinois)
Ken Kennedy (Rice University)
Bob Knighten (Intel SSD)
H. T. Kung (Harvard)
Steve Lundstrom (PARSA)
Robert Malone (Los Alamos National Laboratory)
David Mizell (Boeing)
Reagan Moore (SDSC)
John Riganati (SRC)
Joel Saltz (University of Maryland)
Thomas Sterling (USRA CESDIS)
Rick Stevens (Argonne National Laboratory)
William Tompkins (UTRC)
Lew Tucker (Thinking Machines Corporation)
Paul Woodward (University of Minnesota)
Jerry Yan (NASA Ames)

Applications Working Group

Geoffrey Fox, Chair (Syracuse University)
(315) 443-2163
gcf@nova.npac.syr.edu

Members

Rick Impett (SRC)

Don Leskiw (The Ultra Corporation)

Richard Metzger (Rome Laboratory)

Gary Montry (Southwest Software Structures)

Hugh Nicholas (Pittsburgh Supercomputing Center)

Marcia Pottle (Cornell University)

Sanjay Ranka, Deputy Chair (Syracuse University)

Manny Salas (NASA Langley)

John Salmon (California Institute of Technology)

William Tompkins (United Technologies Research Center)

Jeffrey Young (EPA)

Languages and Compilers Working Group

Ken Kennedy, Chair (Rice University)
(713) 527-4834
ken@rice.edu

Members

Jeff Brown (Los Alamos National Laboratory)

Nicholas Carriero (Yale University)

K. Mani Chandy (California Institute of Technology)

Marina Chen, Deputy Chair (Yale University)

Maya Gokhale (Supercomputing Research Center)

Jim McGraw (Lawrence Livermore National Laboratory)

David Mizell (Boeing Computer Services)

Burton Smith (Tera Computer Company)

Lauren Smith (NSA)

Larry Snyder (University of Washington)

Computing Environments Working Group

Reagan Moore, Chair (San Diego Supercomputer Center)
(619) 534-5073
moore@sds.sdsc.edu

Members

Charlie Catlett (National Center for Supercomputing Applications)

Gregory Follen (NASA Lewis)

Tom Henderson (NOAA)

Alan Kleitz (Minnesota Supercomputer Center)

Adam Kolawa (ParaSoft Corporation)

Barry Leiner (Advanced Decision Systems)

Christopher Maher (Pittsburgh Supercomputing Center)

Bernard T. O'Lear (NCAR)

Bernie A. Perella (NSA)

Peter Rigsbee (Cray Research, Inc.)

Mark Seager (Lawrence Livermore National Laboratory)

Bruce Shapiro (Frederick Cancer Research Development Center)

Operating Systems Working Group

Bob Knighten, Chair (Intel Supercomputer Systems Division)
(503) 629-4315
knighten@ssd.intel.com

Members

Eric Barszcz (NASA Ames Research Center)

David Black (Open Software Foundation)

David E. Culler (University of California, Berkeley)

Howard Gordon (NSA)

Steve Groom (Jet Propulsion Laboratory)

Dan Kopetzky (SRC)

Roger Lee (Thinking Machines Corporation)

Kai Li (Princeton University)

Lex Lane (University of Illinois)

Ed Lazowska (University of Washington)

Rick Light (Los Alamos National Laboratory)

Jishnu Mukerji (Unix System Laboratories)

Ed Upchurch (JPL)

Mike Wan (San Diego Supercomputer Center)

Mathematical Software Working Group

Mike Heath, Chair (NCSA/University of Illinois)
(217) 333-6268
heath@ncsa.uiuc.edu

Members

Ronald Boisvert (NIST)

Tony Chan (University of California at Los Angeles)

Carlie Coats (EPA)

James Demmel (University of California, Berkeley)

Jack Dongarra (University of Tennessee/ORNL)

Brian Ford (Numerical Algorithms Group, Ltd.)

Eric Grosse (AT&T Bell Laboratories)

Sven J. Hammarling (Numerical Algorithms Groups, Ltd.)

Lennart Johnsson (Thinking Machines Corporation)

Jorge Moré (Argonne National Laboratory)

Jean Patterson (Jet Propulsion Laboratory)

Anthony Skjellum (Lawrence Livermore National Laboratory)

Brian T. Smith (The University of New Mexico)

Francis G. Tower (NOAA)

Software Tools Working Group

Joel Saltz, Chair (University of Maryland)
(301) 405-2669
saltz@cs.umd.edu

Members

Donna Bergman (Cornell Theory Center)

Christian Bischof (Argonne National Laboratory)

Dianna Crawford (Cray Research Park)

John Mellor-Crummey (Rice University)

Frederica Darema, Deputy Chair (IBM T. J. Watson Research Center)

Dennis B. Gannon (Indiana University)

Ray Glenn (SRC)

Andrea Overman (NASA Langley)

P. Sadayappan (The Ohio State University)

Robert Schnabel (University of Colorado)

Roy Williams (California Institute of Technology)

Visualization Working Group

Lew Tucker, Chair (Thinking Machines Corporation)
(617) 234-1000
tucker@think.com

Members

Chuck Hansen (Los Alamos National Laboratory)

Ray Idaszak (North Carolina Supercomputer Center)

Paul Woodward, Deputy Chair (University of Minnesota)

Eric de Jong (JPL)

Other Participants

Kamal Abdali (NSF)
Brian Boesch (DARPA)
Art Cullati (EPA)
Robert Ferraro (JPL)
John Gary (NIST)
Leslie Hart (NOAA)
Tony C. Hearn (RAND)
Adolfy Hoisie (Cornell)
Barry Jacobs (NASA)
Gordon Lyon (NIST)
Robert Martino (NIH)
Erik Mettala (DARPA)
Merrell Patrick (NSF)
Bernardo Rodriguez (NOAA)
Bruce Shapiro (NIH)
Walter Shackelford (EPS)
Mike Steuerwalt (NSF)
Walter Stevens (CARB)
Francis Sullivan (NIST)
Robert Voigt (NSF)

DATE FILMED

1/31/94

END