Engineering Physics and Mathematics Division

Mathematical Sciences Section

**EARLY EXPERIENCE WITH THE INTEL IPSC/860
AT OAK RIDGE NATIONAL LABORATORY**

M. T. Heath

G. A. Geist

J. B. Drake

Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2009, Bldg. 9207-A
Oak Ridge, TN 37831-8083

Date Published: September 1990

Research supported by the Applied Mathematical Sciences
subprogram of the Office of Energy Research, U.S. Department of Energy

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
operated by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Contents

1	Introduction	1
2	Intel iPSC/860 Hardware	3
3	Intel iPSC/860 Software	4
4	Performance on Computational Kernels	5
5	Performance on Matrix Computations	11
6	Superconductivity Computations	13
7	Plasma Flow Computations	19
8	Atomic Physics Computations	21
9	References	24

**EARLY EXPERIENCE WITH THE INTEL IPSC/860
AT OAK RIDGE NATIONAL LABORATORY**

M. T. Heath

G. A. Geist

J. B. Drake

Abstract

This report summarizes the early experience in using the Intel iPSC/860 parallel supercomputer at Oak Ridge National Laboratory. The hardware and software are described in some detail, and the machine's performance is studied using both simple computational kernels and a number of complete applications programs.

1. Introduction

Today's leading supercomputers are capable of performing over one billion floating point operations per second (Gflops). The current generation of conventional supercomputers, typified by the Cray-2 and Cray Y-MP as well as a number of Japanese machines, attain such prodigious computational speeds by combining a small number (typically 4 to 8) of very powerful vector processors having a cycle time of a few nanoseconds (typically 4 to 10). For such an environment, efficient parallel implementations of application programs tend to be very coarse grained, meaning that the sizes of tasks executed by individual processors are relatively large. At the opposite extreme, another means of providing Gflop performance is through massive parallelism, in which a very large number (tens of thousands) of very small processors are employed. This approach is typified by SIMD architectures such as those available from Thinking Machines and MassPar. Due to the very limited power and memory of the individual processors, such machines require a very fine granularity of parallelism in applications programs.

An intermediate approach between these two extremes is that of medium-grain, distributed-memory multicomputers, in which a few hundred to a thousand 32- or 64-bit microprocessors are combined by an interconnection network. Such medium-grain parallel machines potentially have a price-performance advantage over either of the other two approaches in that they require fewer custom parts, instead employing mostly commodity parts whose development and manufacturing costs are amortized over hundreds of thousands produced for the personal-computer and workstation markets. The most successful instances of this approach to date have been parallel computers called "hypercubes," named for the topological structure of the network interconnecting the processors. The hypercube architecture, first practically realized at Caltech [19], has served as the basis for a number of commercial machines, some of which ultimately failed in the marketplace (Ametek/Symult and FPS T-series), but others of which have been notably successful (Intel and Ncube).

The aggregate performance of such a medium-grained machine is determined by the performance of its constituent microprocessors, the bandwidth and latency of its interconnection network, and the total number of processors. Given their relatively low-powered processors and limited memory, the first one or two generations of hypercubes were not bona fide supercomputers in that they were not yet competitive with the

fastest conventional machines at the time unless extremely large numbers of processors were used. Nevertheless, these early machines were valuable tools for computational scientists to learn to deal with parallelism in applications. Recently, with the advent of RISC designs and other technological advances, very high performance microprocessors (with cycle times as small as 25 nanoseconds) have become available and are now making their way into multiprocessor and multicomputer architectures. Consequently, this class of architectures has now moved into the Gflop performance range with the commercial release of the Intel iPSC/860 and the Ncube 6400 hypercubes, and is now competitive with any other class of general-purpose supercomputers.

Oak Ridge National Laboratory (ORNL) has had a long association with commercial hypercubes. ORNL was one of the first recipients of the Intel iPSC/1, iPSC/2, and original Ncube/ten (now called Ncube 3200) machines. These machines have been used for basic research in parallel algorithms as well as for a variety of applications at ORNL [10]. In January, 1990, ORNL took delivery of one of the two beta test machines of the new iPSC/860 hypercube produced by Intel (the other was delivered to NASA Ames Research Center). The iPSC/860 is also known as the Touchstone Gamma Prototype, since it represents an early phase of Intel's Touchstone project, whose development is funded in part by DARPA.

The purpose of this paper is to summarize ORNL's early experience with the Intel iPSC/860 machine. In the sections to follow, we will present an overview of both the hardware and software of the iPSC/860, performance data for some basic computational kernels, and results for some initial applications implemented on the machine, including comparisons with performance of the same applications on more conventional supercomputers. One of these applications, superconductivity, is discussed in some detail in order to gain an appreciation for the work done in adapting it for parallel execution. The work reported in this paper involved the efforts of many other people in addition to the listed authors. Contributions by key individuals will be noted in the appropriate sections.

The reader should keep in mind that our conclusions are based on experience with a beta release of the iPSC/860 during the three-month period preceding the first shipments of regular production models to customers. Thus, one should naturally expect more bugs and instability in both hardware and software than might be considered

tolerable in a mature product. It is also unrealistic to expect highly tailored and optimized development tools in such a new environment. Nevertheless, even during this early stage, the iPSC/860 has proven capable of world-class performance and shows great promise for tackling the grand challenges of computational science.

2. Intel iPSC/860 Hardware

Each computational node in the iPSC/860 consists of an Intel i860 processor plus memory and communication components. The iPSC/860 at ORNL has 128 such nodes, the maximum configuration available. Each computational node has 8 Mbytes of memory, for an aggregate total of one Gbyte of RAM. Each i860 processor features an integer core unit, pipelined floating-point units for addition and multiplication, a graphics unit, memory-management support, a large register set, separate instruction and data caches, and 64-bit data paths, all integrated into a single chip having about one million transistors [12,17]. With a clock rate of 40 MHz, each i860 processor has a peak execution rate of 32 MIPS integer performance, 80 Mflops 32-bit floating-point performance, and 60 Mflops 64-bit floating-point performance. Thus, the aggregate peak performance rate of the 128-processor iPSC/860 is over 7 Gflops (64-bit) and 10 Gflops (32-bit). It should be kept in mind, however, that peak execution rates are based on optimal conditions that are difficult to realize or sustain in practice. In particular, the peak rate for the i860 assumes an ideal instruction mix, cache utilization, data alignment, pipelining, etc. These issues will be discussed in greater detail below.

The processors in the iPSC/860 are interconnected by a 7-dimensional hypercube network in which "worm-hole" routing hardware is employed to provide efficient message routing between nonadjacent processors. The network essentially provides circuit switching (as opposed to packet switched, store-and-forward message routing), thereby effectively emulating a fully connected network, with very little penalty for nonlocal communication. The peak data transfer rate across the hypercube interconnection network between any two nodes is 2.8 Mbytes per second.

In addition to the 128 computational nodes, ORNL's iPSC/860 has four I/O nodes, each of which has an Intel 80386 processor and two 650-Mbyte (formatted) disks, for an aggregate total of over 5 Gbytes of disk space. These I/O nodes and the disks they support are directly accessible to the computational nodes over the interconnection

network. Peak data transfer rate between a single computational node and the I/O node disks is about 1.5 Mbytes per second. When more computational nodes access the I/O disks simultaneously, the aggregate throughput initially increases, peaking at about 3 Mbytes per second, but eventually degrades due to contention as still more processors are used. For more detailed performance data for the iPSC/860 on basic I/O, communication, and arithmetic operations, see [5].

Like most machines of its type, the iPSC/860 is not a stand-alone machine, but requires a host machine to serve as its interface to the outside world for program development, resource management, and external network access. The host machine, known in Intel terminology as the System Resource Manager (SRM), is an Intel 301 microcomputer, which features an Intel 80386/387 processor pair running at 16 MHz, 8 Mbytes of RAM, a 300-Mbyte disk, cartridge tape unit, and an Ethernet network connection. The SRM is attached to the hypercube network, and this link provides a peak data transfer rate of over 1 Mbyte per second.

3. Intel iPSC/860 Software

The user interface and software environment for the iPSC/860 reside primarily on the SRM. The SRM runs Unix System V, Release 3.2, with support for TCP/IP networking and the Network File System (NFS) via Ethernet. The disk space on the I/O nodes is managed by a separate Concurrent File System (CFS) that is not currently integrated with the SRM disk or NFS. A special shell is provided, however, for accessing and managing CFS files from the SRM.

The computational nodes in the iPSC/860 system run a simple operating system kernel called NX that supervises process execution and provides buffered, queued message passing (including communication to I/O nodes or the SRM). Like other MIMD hypercubes, the programming model for the iPSC/860 is based on adding explicit communication calls (send/recv) to serial code written in a conventional programming language (C or Fortran). At present, there is no automation provided to aid in parallelizing programs, but a node debugger is available.

Compilation of either C or Fortran for the i860 node-processor target takes place on the SRM. The cross-compilers currently available do not take specific advantage of any of the special features of the i860 processor (dual instruction mode, etc.) that give it its

unusually high performance. The result is that compiled code from high-level language source generally runs at about an order-of-magnitude lower performance than the peak rates expected for the i860. Specific performance data will be detailed below.

The i860 development tools currently available (compilers, linker, assembler, archiver) run very slowly on the SRM (much more slowly than their counterparts for the 80386/387 target), even for a single user. If multiple users run the i860 development tools on the SRM simultaneously, the SRM slows to a crawl. For example, at ORNL we have i860 application programs that cannot be built on the SRM in an eight-hour shift. Thus, although the computational performance of the iPSC/860 is competitive with conventional supercomputers, it is not yet in the same league in terms of the program development cycle. Intel and a number of third-party software houses are presently working on enhanced compilers and other development tools for the i860 that should be much more efficient, both in building programs and in executing them on the i860. In addition, another obvious route to alleviating some of the SRM bottleneck would be to move program development elsewhere on the network onto higher powered workstations via additional cross compilers, or onto the hypercube itself. Such improvements will be necessary before the iPSC/860 can become the same kind of everyday production workhorse that one expects of conventional supercomputers, such as the Cray series, where compilations seem almost "instantaneous."

4. Performance on Computational Kernels

Basic operations on vectors and matrices are common in all areas of scientific computing. These fundamental building blocks form the inner loops of many numerical algorithms and are a dominant factor in determining the performance of many numerically-intensive applications programs. The performance of these computational kernels is therefore of great interest on any computer architecture, and they tend to be among the first benchmarks run on any new processor. The definitions and user interfaces for these low-level operations have been standardized in the Basic Linear Algebra Subprograms (BLAS) [15], which in turn form the basis for portable implementations of higher-level matrix operations, such as solving systems of linear equations (see, e.g., [3]).

When implemented in a high-level programming language such as Fortran or C,

the BLAS can be made portable across a wide range of computers, but their performance rarely approaches the theoretical peak on any individual machine. The usual approach, therefore, is to implement custom-coded versions of the BLAS in assembler language for each particular processor, while maintaining the same user interface across all implementations so that programs that call the BLAS will remain portable, while retaining the speed advantage of assembler coding in their inner loops. High-level language implementations of the BLAS are still of interest, however, in that the performance gap between them and optimized implementations in assembler language serves as an indication of the effectiveness of the compilers for a given machine.

We have implemented a number of the most important BLAS in assembler language for the Intel i860 processor and compared their performance with standard implementations in Fortran and C. Both in writing these codes and in testing their performance, we were confronted with a number of options regarding methodology. The i860 processor has a number of features, and corresponding instructions in its instruction set, that potentially enhance its performance, but whose exploitation may limit the general applicability of the resulting code. For example, the "quad load" feature allows the fetching of 128 bits of data from memory with a single instruction, but only if the data are aligned on a "quad word" boundary (i.e., a byte address that is a multiple of 16). The use of this capability substantially increases the effective bandwidth between processor and memory, but in many applications it is impractical or impossible to meet the concomitant restriction on data alignment. Thus, in writing a general-purpose code, one must either forgo using this special feature entirely, or else detect those (possibly rare) cases for which it is applicable and exploit it only then. Clearly, this issue must be kept in mind when choosing benchmark tests and interpreting the results.

The "advertised" peak performance figures cited earlier for the i860 are based on ideal conditions, including alignment of data on proper word boundaries, perfect pipelining, no cache misses, an instruction mix that exactly matches the functional units of the processor, optimal use of dual instruction mode, etc. Full realization of these conditions in real programs for any sustained period of time is undoubtedly extremely rare, but they can conceivably be achieved in simple, artificial benchmark tests such as isolated tests of individual routines from the BLAS. However, we are confronted here by another thorny question of methodology regarding cache usage. An individual

call to a single BLAS routine on a high-performance processor is too brief to yield reliable timing results. The usual approach to such a problem is simply to replicate the test many times, perhaps several thousand, so that overall execution times can be measured accurately. Unfortunately, a far higher percentage of cache hits is likely to occur in such a replicated test than would be experienced in one-time usage of the routine, thereby significantly skewing the results. On the other hand, there certainly are instances in actual practice, some of which will be noted in the next section, in which data *can* be expected to remain in cache for sustained periods if the algorithm is carefully constructed. The reader should keep these comments in mind when interpreting this section's results, which were obtained through replication in order to produce accurately measurable execution times.

In a high-level language such as Fortran or C, the user has little specific control over cache utilization, but in i860 assembler, data traffic to and from memory can bypass cache at the programmer's option to obtain better overall cache utilization. The general principle, of course, is that reusable data should be cached, while nonreusable data should bypass cache so that it does not displace any reusable data that may already reside there. For example, one of the most important computational kernels in linear algebra is to compute the result of a scalar times a vector plus a vector, commonly known in BLAS terminology as **axpy**, $y = \alpha x + y$, where x and y are vectors and α is a scalar. Cache management is particularly important for this operation because of the different roles played by the variables involved. In particular, y is both fetched and stored, while x is only fetched, so it may be advantageous to cache y , while bypassing cache with x .

Figures 1 through 3 show results for the BLAS routines **saxpy**, **daxpy**, and **zaxpy** for single precision real, double precision real, and double precision complex data, respectively. In all cases the vector length is measured in words of the appropriate size for the precision involved. The execution rates shown were obtained using timing tests that make 10,000 successive calls of the basic routine, using a stride of 1 and with the same argument list for each call. Thus, after the first call, some data may remain in cache that is potentially reusable on successive calls, depending on the vector length involved. Some of the implementations shown use special instructions that load multiple words, but these require the input vectors to be aligned on special boundaries.

Figure 1: Execution rate for various implementations of **saxpy** on the Intel i860. Asterisk indicates routine requires aligned data.

The routines using the special instructions are indicated by an asterisk in the figures. These results are included for comparison purposes, but it should be kept in mind that not all applications will be able to meet the restrictions on data alignment, and hence may not be able to take advantage of the higher performance offered by the kernel implementations that use these special features. The maximum execution rates achieved with assembler-coded **axpy** routines are about 55 Mflops with single precision data, and about 27 Mflops with double precision data. The 2-to-1 performance ratio between the two precisions for this computation is presumably due to the fact that the i860 can issue a single precision multiply instruction on every clock cycle, but it can issue a double precision multiply instruction only on alternate clock cycles. These maximum execution rates represent about 50% to 70% of the advertised peak performance of the i860, depending on the precision. Fortran implementations of **axpy**, on the other hand achieve only 7% to 15% of theoretical peak performance, suggesting that the compiler is taking little or no advantage of the i860's high-performance features.

Figure 2: Execution rate for various implementations of **daxpy** on the Intel i860. Asterisk indicates routine requires aligned data.

Cache effects can also be seen clearly in Figures 1 through 3. Performance falls off

markedly when the vector size exceeds the 8 Kbyte cache size. This point is reached for vectors only half as large if both x and y are cached. And of course, the longer wordlength of double precision and complex data cause the cache to be saturated at a smaller vector size. As expected, caching only y gives better performance than caching only x . The best performance occurs when y is cached and x is "quad aligned," so that it can be piped from memory in 128-bit chunks. For very large vectors, the performance curves for all of the assembler routines converge to about the same level, which is little better than that of Fortran, suggesting that "strip mining" should be used to keep vector lengths within the cache size. Unlike the assembler routines, the Fortran routines are relatively insensitive to vector length and precision of data, but there is little virtue in this consistency.

Figure 3: Execution rate for various implementations of `zaxpy` on the Intel i860.

Figure 4 shows results for the BLAS routines `sdot`, `ddot`, and `zdot`, which compute the inner product of two single-precision, double precision, or complex double precision vectors, respectively. In our assembler implementations, one of the two vectors is cached, while the other is piped from memory, bypassing cache. Again we see some

fairly clear-cut cache effects similar to those seen earlier for **axy**. Note, however, that we do not see the factor of two difference in performance between single precision and double precision for **dot** that we saw for **axy**. The relatively higher performance for **zdot** is presumably due to the larger ratio of arithmetic operations to memory accesses for complex arithmetic.

The assembler kernels whose performance is reported in this section were written by T. H. Dunigan, R. E. Flanery, and G. A. Geist.

Figure 4: Execution rate for **sdot**, **ddot** and **zdot** on the Intel i860.

5. Performance on Matrix Computations

Although simple computational kernels such as those discussed in the previous section do a reasonably good job of capturing the flavor of the dominant inner loops of many scientific applications, their performance as isolated modules does not necessarily reflect their performance when used within the context of a larger, more complicated program. Cache management, in particular, plays a major role in determining overall performance and is much less straightforward to optimize in a real program with complicated data

structures, memory reference patterns, and multiple computational phases.

For benchmarking purposes, a useful compromise between simple computational kernels and fully-detailed application programs is provided by more substantial operations on matrices, such as matrix factorization to solve systems of linear equations. With their more complex memory-reference patterns, such matrix operations place a more realistic and demanding load on data paths between processor and memory, and contain enough computation that performance can be accurately measured without replication. These features probably account for the popularity of the Linpack benchmark, in which a linear system of order 100 or 1000 is solved as a basis for comparing floating-point performance of many computers [2]. At present, however, the Linpack benchmark code is limited to serial computers and a few shared-memory multiprocessors (with or without vector capability), so we have developed our own programs for matrix factorization for parallel execution on multiple processors of the distributed-memory iPSC/860.

We first consider the Cholesky factorization $A = LL^T$ of a symmetric positive definite matrix A , where L is a lower triangular matrix. There are three basic algorithms for implementing this factorization, corresponding to different ways of arranging the triply-nested loop that defines the computation (see [9] for a full discussion and an explanation of the terminology we use here):

- row-Cholesky, for which the inner kernel is `sdot`,
- column-Cholesky, for which the inner kernel is `saxpy`,
- submatrix-Cholesky, for which the inner kernel is `saxpy`.

Column-Cholesky and submatrix-Cholesky are both column-oriented and both use `saxpy`, but they differ in their memory-reference patterns. Column-Cholesky makes repeated calls to `saxpy` with the same y but different x vectors, whereas submatrix-Cholesky makes repeated calls to `saxpy` with the same x but different y vectors. Moreover, in a parallel implementation, column-Cholesky uses fan-in communication, while submatrix-Cholesky uses fan-out communication. Row-Cholesky makes repeated calls to `sdot` with one vector fixed and the other varying, and in a parallel implementation uses fan-out communication. These features suggest a different strategy for each of the algorithms in the use of cache by the underlying BLAS kernel.

The single-processor i860 performance of the three Cholesky algorithms is shown in Table 1. The variation in single-processor performance among the algorithms is due in part to differences in their effectiveness in exploiting cache. Note that we have not used "strip mining" in any of the algorithms to enhance the use of cache, so for large matrices the vectors involved in a given call to a BLAS routine may exceed the cache size. The row Cholesky algorithm based on inner products is clearly the most effective algorithm for this computation on the i860 processor, probably due to the fact that it requires no stores to memory (or cache) in accumulating the inner products, whereas the other two algorithms require stores in the inner loop. The performance of the serial Cholesky algorithms correlates well with the performance of their underlying BLAS kernels, as reported in the previous section.

BLAS	Precision	Row-Cholesky	Col-Cholesky	Sub-Cholesky
C	single	5.1	3.7	3.8
C	double	4.4	3.3	3.3
assembler	single	24.6	15.5	N/A
assembler	double	22.4	14.8	10.7

Table 1: Asymptotic execution rate in Mflops for Cholesky factorization on a single i860 processor.

Multicomputer performance of the row-Cholesky algorithms is shown in Figure 5, using all 128 processors of the iPSC/860. In the multicomputer case, the use of double precision also doubles the necessary communication volume (measured in bytes), in addition to incurring a slightly lower arithmetic execution rate. We see that performance exceeds 1 Gflop for single precision, and about 600 Mflops for double precision.

The work reported in this section was done by M. T. Heath, greatly assisted by the assembler kernels discussed in the previous section.

6. Superconductivity Computations

The discovery of high temperature superconductivity in 1986 has provided the potential for spectacularly energy-efficient power transmission technologies, ultra-sensitive instrumentation, and other devices. Each year new materials are found to add to the family of existing high temperature superconductors. In general these materials are difficult to form and use, and some of the superconducting compounds are unstable.

Figure 5: Execution rate of row-Cholesky factorization using 128 processors on the Intel iPSC/860.

These difficulties are exacerbated by the lack of an accepted theory explaining superconductivity at higher temperatures. To further our understanding of the behavior of solids in general and of superconductors in particular, quantum-mechanical laws have been incorporated into sophisticated computer algorithms to predict from first principles the structural, vibrational, and electronic properties of matter.

Present calculations of the electronic structure of real materials usually employ a mean field approximation in which each electron is viewed as moving independently in a self-consistent potential due to all of the electrons and nuclei. According to density-functional theory, it is possible to express the energy of any system of electrons and nuclei as a unique functional of the electron density [1]. Since this functional is not known exactly, it is usually approximated by that of a homogeneous electron gas. This local density approximation to density functional theory has been very successful when applied to metallic and semiconducting systems, but it appears inadequate to explain important physical phenomena such as optical band gaps and superconductivity found in transition metal oxides. More sophisticated treatments of the many-electron problem are possible, but have not been attempted previously because the Green's function and the susceptibility function needed to construct the electron self-energy are very difficult to calculate for real systems, especially those with narrow bands such as transition metal oxides.

The approach taken by researchers at ORNL is based on theoretical advances growing out of work on the Korringa, Kohn, and Rostoker coherent potential approximation (KKR-CPA) theory of alloys and magnetism [13,14]. The advantage in using the KKR-CPA approach is that it yields directly the Green's function for the system and thereby a direct way of calculating susceptibilities. The effects of disorder are treated in the CPA, which is an analytic technique for calculating the configurationally averaged Green's function [20]. The KKR approach is a natural context for implementing the CPA, because it is a Green's function method, and there is a natural separation between the lattice and the potential.

Researchers at ORNL and their colleagues have developed a self-consistent, semi-relativistic KKR-CPA computer code that can handle multiple atoms per unit cell. The code has wide applicability to situations in which some form of substitutional disorder plays an important role, including metallic alloys, high-temperature supercon-

ducting compounds, metallic magnetism, and metal-insulator transitions. There are three primary reasons for parallelizing this code.

- The KKR-CPA calculations are computationally intensive. A single KKR-CPA calculation commonly requires 10 hours of CPU time on a Cray-2. An estimated 1000 hours or more of Cray CPU time would be needed to complete a single self-consistent computational experiment. The turnaround time for such experiments makes them prohibitive on conventional supercomputers.
- The KKR-CPA algorithm embodies natural parallelism that can be exploited to increase computational throughput. The feature we exploit is the calculation of the density of states (DOS) at a given energy level. Computation of the DOS at over one hundred energy levels is required to determine the Fermi level. Each of these DOS can be calculated independently of the other energy levels.
- The availability of a parallel computer with Gflop performance, the iPSC/860, has made it feasible and attractive to develop an efficient parallel version of the KKR-CPA code.

In adapting the KKR-CPA code to a parallel setting, the modifications to the KKR-CPA code were made in such a way that the code can still be run on Crays, and smaller problems can be run on scientific workstations. Having only one consolidated code to maintain has made software modifications much simpler to implement than trying to keep three versions of the code up to date. Moreover, the user interface is identical across all the machines the code runs on, which has been an important factor in inducing scientists to use the code in a relatively unfamiliar parallel environment. Explicit parallelism is hidden from the user, with operations such as allocating a number of processors and loading programs onto these processors done automatically by the code. The number of processors to be used in a given computational experiment is specified in the input file, making it easy for the user to control the degree of parallelism for each run of the program. Organizing the code in this way required writing only a few new routines to be added to the existing serial code. None of the additional routines involved new calculations, so exactly the same computational routines are called in the serial and parallel versions.

A master/slave paradigm is used in the parallel implementation. In this scheme, one

processor controls work on the entire problem, and the remaining processors perform work requested by the master process. The master process in our implementation is called the pseudo-host and executes on one of the iPSC/860 nodes. We avoided using the SRM for the master process because of the computational imbalance between its 80386-based processor and the much more powerful i860-based computational nodes. The SRM is also burdened with executing the Unix operating system and program development tools for time-shared use by multiple users.

The KKR-CPA algorithm is organized in the following way. We start by inputting the atomic numbers of the species and initial estimates for the charge density and potentials. Since the Green's function for the system at any energy is independent of any other energy, this is a natural point in the algorithm at which to exploit parallelism. In the parallel implementation, the energies to be evaluated are held in a queue of tasks. The difficulty of each task is initially unknown, so a heuristic strategy is used to arrange the queue in order of approximately decreasing difficulty. Each idle processor selects the next task in the queue and returns the DOS to the master process, which computes the integral over all energies. Load balancing is achieved naturally, with all the processors remaining busy as long as there are tasks left in the queue.

The most computationally intensive portion of the tasks assigned to the processors is integrating the KKR matrix inverse over the first Brillouin zone. To evaluate the integral, hundreds or possibly thousands of complex double precision matrices of order between 80 and 300 must be formed and inverted. Each matrix corresponds to a different vertex of the tetrahedrons into which the Brillouin zone has been subdivided. The results of the integration are used to compute the Green's function for the system and the DOS for the given energy.

A further outer iteration is necessary to incorporate self-consistency of the charge density into the KKR-CPA code. This outer iteration involves integrating the Green's function over energy to obtain the charge density, which is used to derive the potential for the next iteration. Thus the entire process described thus far is repeated several times in the self-consistent version of the code, greatly magnifying the already substantial computational demands of the program.

Using the high temperature superconductor $\text{Ba}_{1-x}\text{K}_x\text{BiO}_3$ as a test problem, the consolidated KKR-CPA code has been run on several computers. The test problem

requires the calculation of the DOS for a fixed number of representative energies, without iterating to self-consistency. The average execution rate for a range of computers is shown in Table 2.

Machine	Processors	Mflops	Comments
DEC 3100	1	2	
IBM RS/6000	1	18	Model 530
Cray-2	1	49	
Cray Y-MP	1	130	Fortran only
Cray Y-MP	1	203	assembler BLAS
Cray Y-MP	8	1509	multitasking
iPSC/860	128	725	Fortran only
iPSC/860	128	1792	assembler BLAS

Table 2: Average execution rate for superconductor test problem on several computers.

The KKR-CPA code using only Fortran runs at a rate of about 130 Mflops on a single processor of the Cray Y-MP. Performance increases to about 203 Mflops when assembler-coded BLAS are used. The addition of multitasking to make use of all eight processors of the Cray Y-MP yields an aggregate performance of over 1.5 Gflops. The rate shown for the iPSC/860 includes the time to load the problem onto 128 processors, all communication, file I/O (four fairly large output files are generated), and dynamic load balancing overhead. The rate of 725 Mflops was attained using only compiled Fortran. This rate was increased to about 1.8 Gflops by using an assembler language `zaxpy` in the inversion routine and in the formation of the KKR matrix. The test problem used here is too small to attain the asymptotic execution rate of which the code is capable on the i860. Larger problems are expected to yield execution rates of approximately 2.5 Gflops.

The use of parallel computation on the iPSC/860 has led to more than an order-of-magnitude improvement in computational speed compared to a single processor of the Cray supercomputers previously used for the KKR-CPA code. From a research standpoint the improvement in turnaround time for computational experiments is even more substantial, since each subcube of the iPSC/860 is dedicated to only one user at a time, while the Crays are time-shared by many users. This greater computational power allows us to begin investigating many unanswered questions in superconductivity and materials science. For example, several studies are underway on the effects of alloying in the two perovskite superconducting compounds $\text{Ba}_{1-x}\text{K}_x\text{BiO}_3$ and $\text{BaPb}_{1-x}\text{Bi}_x\text{O}_3$.

The work reported in this section was done by G. A. Geist, W. A. Shelton, and G. M. Stocks. For further details, see the paper [8].

7. Plasma Flow Computations

One of the obstacles to the design of a magnetic-fusion reactor is an understanding of anomalous transport mechanisms that destroy plasma confinement. The onset of plasma turbulence can be studied numerically by considering the dynamics of the plasma edge. Detailed measurements are possible at the plasma edge using probes, so that experimental verification of numerical models is possible. The study of plasma edge turbulence faces many of the same challenges as the classical fluid turbulence problem. All the important time scales and length scales must be resolved in a numerical computation, and this strains the abilities of present supercomputers. In addition, since the plasma is not a perfect conductor, the turbulence can cause changes in the topology of the magnetic field. These topological changes are critical for plasma confinement.

A code for studying plasma instabilities based on the reduced magneto-hydrodynamic (MHD) equations has been in use by researchers in the Fusion Energy Division of ORNL for several years [11]. This code has been optimized for use on the Cray machines available at the National Energy Research Supercomputer Center. As a pilot study, the code was implemented on the Intel iPSC/1 hypercube [4]. The MHD equations in a toroidal geometry are discretized by a pseudo-spectral method, with derivatives in the time and radial directions approximated by finite differences while functions of the two angular variables are expanded in Fourier series. Derivatives in the angular variables are performed analytically. All quantities are stored in spectral form. The nonlinear convection terms are taken to be explicit in time, while linear terms are treated implicitly. The convolutions arising from the nonlinear terms are performed analytically, rather than using a fast Fourier transform. This allows for explicit study of mode interactions during the nonlinear evolution.

Parallelism is incorporated by a spatial domain decomposition of the radial coordinate. This approach preserves data locality for the computationally intensive convolution calculation. The implicit terms require solving multiple tridiagonal systems distributed across the processors. A ring-based, pipelined solution strategy is used for this phase. Results on the iPSC/1 indicated that the calculation is well suited to large-

scale parallel computation, attaining parallel efficiencies above 90%. But as a practical matter, on this early hypercube the run time was considerably larger than corresponding runs on Cray machines. The iPSC/1 is also limited by its relatively small memory of 512 Kbytes per processor. Only small problems having 20 to 30 modes fit in memory, whereas interesting simulations involve 500 or more modes.

The second generation Intel hypercube, the iPSC/2, offers an improvement in both processor speed and memory size, but still falls short of the Cray-2 in overall run time. The 4 Mbytes of memory per processor accommodates much larger problems, but a 500-mode case remains infeasible on the iPSC/2. The Intel iPSC/860, on the other hand, with 8 Mbytes of memory per processor and vastly improved computational speed, has the potential to run full-sized simulations with times comparable to any other supercomputer available. Figure 6 shows execution times on two generations of Intel hypercubes, the iPSC/2 and iPSC/860. Each data point represents the CPU time required to take 10 time steps using 16 processors.

Figure 6: Execution times of plasma flow code using 16 processors on two generations of Intel hypercubes.

Table 3 shows CPU times on the iPSC/860 compared with the corresponding single-processor Cray-2 time for a 300-mode case. The Cray code uses 60-bit precision, while

the iPSC/860 code uses 64-bit precision. The Cray code has unrolled loops and takes advantage of vectorization of the inner loop of the convolution calculation. We have begun to experiment with optimizing the inner loop on the iPSC/860. By rearranging the order of loop indices we can obtain a factor of two increase in performance. We find that the unoptimized inner loop executes at about 1.9 Mflops per processor. With compiler optimization, the execution rate increases to 4.5 Mflops. We expect to increase this speed significantly in the future by using assembler-coded routines.

Machine	Processors	CPU time	Comments
Cray-2	1	67	
iPSC/860	16	180	unoptimized Fortran
iPSC/860	16	130	optimized Fortran
iPSC/860	16	110	unoptimized, loops rearranged
iPSC/860	16	69	optimized, loops rearranged
iPSC/860	32	49	optimized, loops rearranged

Table 3: Times in seconds for 10 steps of the plasma flow code for a problem with 300 modes.

Since the timing results from these tests appear favorable for large scale computations simulating plasma edge turbulence, we are developing more extensive models, such as the KITE code [7], for use on the iPSC/860.

The work reported in this section was done by J. B. Drake and V. E. Lynch. For further details see the paper [16].

8. Atomic Physics Computations

The collisions of heavy ions at high energy levels can be simulated using a quantum electrodynamic framework, which is somewhat simpler and more tractable than the full coupling of quantum chromodynamics. One of the structures to emerge from the collision is a strongly coupled lepton-antilepton pair. This structure usually decays in less than 10^{-19} seconds. The simulation of the production and decay of leptons is a formidable computational challenge. The ability to simulate the production of such pairs is important in the design of experiments for colliders currently under construction. Until recently, most accelerator designers have worked in the domain where fundamental interactions can be decoupled from engineering considerations, with little or no involvement in the modeling of basic processes on supercomputers. It is now rec-

ognized, however, that beam stability and focusing in all advanced collider designs are strongly influenced by pair production. The theory of these processes is rapidly evolving for both high-energy heavy-ion colliders and for electron-positron linear colliders [18].

The fundamental model for heavy ion collisions is based on Dirac's equation [21]. The solution of this equation gives the motion of the leptons, which are assumed to move independently of the classical electromagnetic fields. Dirac's equation relates the time rate of change of the particle probability distributions to spatial derivatives of the distributions. This equation is linear, so that propagation of the distribution in time can be represented as an operator exponential. Methods developed for the simulation of lepton-pair production are also applicable to related non-relativistic problems in nuclear, chemical, surface and plasma physics.

A B-spline collocation method for Dirac's equation has been implemented on the iPSC/2 and iPSC/860, as well as on the Cray and the FPS T-series computers. The collocation method employs basis splines of user-selectable order. The high-order splines have better approximation properties than low-order splines or the simple interpolants typically used in finite difference and finite element discretizations. The B-spline collocation method thus has excellent convergence and accuracy properties. The method also allows implementation in a storage-efficient tensor product style, where the effects of the discrete operator in each coordinate direction are separated. The desired level of resolution is a $100 \times 100 \times 100$ lattice. The computations typically involve solving an eigenvalue problem for the initial minimum energy state and then taking several thousand time steps through a transient pair production and decay. The eigenvalue calculation is an iterative procedure using only the operator; the tensor form of the operator replaces explicit formation of a matrix representing the operator.

Parallelism is introduced by domain decomposition, with processors and data assigned in a two-dimensional grid or, with edges connected, a torus arrangement. Applying the discrete Dirac operator to a state vector requires three matrix-matrix products, one for each coordinate direction. Two of these directions, y and z , have data divided among the processors by the domain decomposition. The first matrix of the product represents the derivatives of the Dirac operator in the particular coordinate direction. This matrix is of order 100 for the desired resolution and can easily be formed and stored

on each processor, so that only the state vector is then divided among the processors. A "roll" algorithm similar to the well known parallel matrix-matrix product algorithm [6] is employed to accumulate the results. This phase requires nearest-neighbor communication on the grid of processors, accumulating the results for the y direction using rings of processors in one direction and then accumulating the results for the z direction using rings of processors in the perpendicular grid direction. The inner loop of the implementation is a **saxpy**, even though all of the state vectors are complex. Real and imaginary parts of the state vector have been rearranged to gain efficiency from the use of only real arithmetic.

Preliminary performance data for this code on the iPSC/860 are shown in Figure 7. The figure shows dramatic improvements in computational speed with the use of an assembler-coded **saxpy** for the inner loop calculation. The crossover in performance between the two codes based on assembler-coded **saxpy** is due to the lower efficiency of the the aligned code when the vector lengths are short, which is the case when a fixed-size problem is spread over more processors. Timing studies and operation counts show a dependence on the number of processors, p , and the grid resolution, n , that is proportional to n^4/p . Physically interesting results computed at the rates shown and at the desired resolution can be obtained with runs lasting approximately one day.

Figure 7: Execution rate of lepton pair code for a $16 \times 16 \times 16$ lattice.

Efforts to optimize the performance of the code for the iPSC/860 have thus far taken precedence over exploring the physics of pair production using the iPSC/860's computational power. The latter activity will begin in earnest during the next few months. The work reported in this section was done by C. Bottcher, J. B. Drake, R. E. Flanery, and M. R. Strayer.

9. References

- [1] U. Von Barth, Density functional theory for solids. In P. Phariseau and W. M. Temmerman, editors, *The Electronic Structure of Complex Systems*, pages 67-140, Plenum Press, New York, 1984.
- [2] J. J. Dongarra, Performance of various computers using standard linear equations software. Tech. Rept. CS-89-85, Dept. of Computer Science, University of Tennessee, Knoxville, TN, January 1990.
- [3] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *Linpack User's Guide*, SIAM, Philadelphia, 1979.
- [4] J. B. Drake, W. F. Lawkins, B. A. Carreras, H. R. Hicks, and V. E. Lynch, Implementation of a 3-D nonlinear MHD calculation on the Intel hypercube. Tech. Rept. ORNL-6335, Oak Ridge National Laboratory, Oak Ridge, TN, 1987.
- [5] T. H. Dunigan, Performance of the Intel iPSC/860 hypercube. Tech. Rept. ORNL/TM-11491, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.
- [6] G. Fox, et al. *Solving Problems on Concurrent Computers*. Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [7] L. Garcia, H. R. Hicks, B. A. Carreras, L. A. Charlton, and J. A. Holmes, 3-D nonlinear MHD calculations using implicit and explicit time integration schemes. *J. Comput. Phys.*, Vol. 65, pages 253-272, (1986).
- [8] G. A. Geist, B. W. Peyton, W. A. Shelton, and G. M. Stocks, Modeling high-temperature superconductions and metallic alloys on the Intel iPSC/860. *Proc. Fifth Distributed Memory Computing Conf.*, to appear.

- [9] A. George, M. T. Heath, and J. Liu, Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Algebra Appl.*, Vol. 77, pages 165-187, (1986).
- [10] M. T. Heath, Hypercube applications at Oak Ridge National Laboratory. In M. T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 395-417, SIAM, Philadelphia, 1987.
- [11] H. R. Hicks, B. A. Carreras, J. A. Holmes, D. K. Lee, and B. V. Waddell, 3-D nonlinear calculations of resistive tearing modes. *J. Comput. Phys.*, Vol. 44, pages 46-69, (1981).
- [12] L. Kohn and N. Margulis, Introducing the Intel i860 64-bit microprocessor. *IEEE Micro*, Vol. 9, No. 4, pages 15-30 (1989).
- [13] W. Kohn and N. Rostoker, Solution of the Schrödinger equation in periodic lattices with an application to metallic lithium. *Phys. Rev.*, Vol. 94, page 1111, (1954).
- [14] J. Korringa, On the calculation of the energy of a Bloch wave in metal. *Physica*, Vol. 13, page 392, (1947).
- [15] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, Vol. 5, pages 308-325, (1979).
- [16] V. E. Lynch, B. A. Carreras, J. B. Drake, and J. N. Leboeuf, Plasma turbulence calculations on the Intel iPSC/860 (RX) hypercube. *Internat. J. Supercomputer Appl.*, submitted.
- [17] N. Margulis, The Intel 80860. *Byte*, Vol. 14, No. 13, pages 333-340, (1989).
- [18] M. Month, Physics of particle accelerators. *A.I.P. Conference Proceedings*, Vol. 184, Ithaca, NY, 1988.
- [19] C. L. Seitz, The cosmic cube, *Comm. ACM*, Vol. 28, pages 22-33, (1985).
- [20] P. Soven, Application of the coherent potential approximation to a system of muffin-tin potentials. *Phys. Rev.*, Vol. 156, page 809, (1967).

- [21] A. S. Umar, J. Wu, M. R. Strayer, and C. Bottcher, Basis-spline collocation method for the lattice solution of boundary value problems. J. Comput. Phys., submitted, 1989.

END

DATE FILMED

01 / 15 / 91