# SPLASH: STANFORD PARALLEL APPLICATIONS FOR SHARED-MEMORY

Jaswinder Pal Singh
Wolf-Dietrich Weber
Anoop Gupta

Technical Report No. CSL-TR-91-469

April 1991

This research has been supported by DARPA contract N00014-87-K-0828. Authors also acknowledge support from an IBM graduate fellowship for Wolf-Dietrich Weber and an NSF Presidential Young Investigator Award for Anoop Gupta

# SPLASH: STANFORD PARALLEL APPLICATIONS FOR SHARED-MEMORY

**Jaswinder Pal Singh, Wolf-Dietrich Weber
and Anoop Gupta**

**Technical Report: CSL-TR-91-469**

April 1991

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 943054055

## Abstract

We present the Stanford Parallel Applications for Shared-Memory (SPLASH), a set of parallel applications for use in the design and evaluation of shared-memory multiprocessing systems. Our goal is to provide a suite of realistic applications that **will** serve as a well-documented and consistent basis for evaluation studies. We describe the applications currently in the suite in detail, and explore their behavior by running them on a real multiprocessor as well as on a simulator of an idealized parallel architecture. We expect the current set of applications to act as a nucleus for a suite that will grow with time.

**Key Words and Phrases:** parallel applications, architecture evaluations, **shared-**memory multiprocessors

# SPLASH: Stanford Parallel Applications for Shared-Memory

Jaswinder Pal Singh, Wolf-Dietrich Weber and Anoop Gupta

Computer Systems Laboratory
Stanford University, CA 94305

### Abstract

We present the Stanford Parallel Applications for Shared-Memory (SPLASH). a set of parallel applications for use in the design and evaluation of shared-memory multiprocessing systems. Our goal is to provide a suite of realistic applications that will serve as a well-documented and consistent basis for evaluation studies. We describe the applications currently in the suite in detail, and explore their behavior by running them on a real multiprocessor as well as on a simulator of an idealized parallel architecture. We expect the current set of applications to act as a nucleus for a suite that will grow with time.

## 1 Introduction

Designers of parallel systems are faced with a chicken and egg problem regarding applications software. Few real parallel applications exist to guide their designs, and users are unwilling to write new applications for systems that do not exist. The result is that studies done to evaluate system features often base their conclusions on "toy" programs that bear little resemblance to, or are only a part of, the codes people will actually run on these systems. In providing the Stanford ParalleL Applications for SHared-memory (SPLASH), we hope to meet the existing need for more complete applications. Drawn from several scientific and engineering problem domains, the applications are intended as a design aid for architects and software people working in the area of shared-memory multiprocessing.

The use of real applications for studying system performance, however is not without pitfalls. Dongarra *et* al. [I] discuss some of these in the context of sequential and vector computing. Besides the scarcity of applications, and the consequent difficulty of identifying a "representative" set, parallel computing introduces new limitations to the design and use of application suites, and accentuates some of the existing reasons for caution:

- The software technology for writing parallel programs is immature. It is unclear how well programs written with today's constructs will represent those that might be written in the future, and what the implications of this are for the effectiveness of evaluation studies performed today.
- The available programs might not represent the best parallelization of the problem they solve, but only one that is reasonable and convenient to implement. Even more significantly, large-scale parallel processing might call for very different algorithms than those implemented on smaller machines today.
- The relationships between applications and architectures take on new dimensions with parallelism. Managing data locality, for instance, becomes significantly more complicated, as do the interactions between scaling problem and multiprocessor sizes. The number of architectural variables is also much larger, making careful accounting for all assumptions more important as well as more difficult in understanding the impact of transformations.

Besides the above limitations that apply to any set of parallel applications, the SPLASH suite and input data sets have their own particular shortcomings. Firstly, the parallelism in some of the applications is limited by the nature and size of the input data sets. Since we use a simulator, rather than a real multiprocessor, to characterize the applications, runs with large data sets take too long. We expect people with real, high-performance parallel machines to run the programs with much Larger as well as more realistic data sets, and we provide these wherever possible. Secondly, the applications were written with small to medium-scale machines in mind, and may require restructuring to run efficiently on larger multiprocessors.

As a result of the above limitations, we believe that it is inappropriate to use the SPLASH applications as benchmarks for definitive, quantitative comparisons to prove one system superior to another (the way one might use the SPEC benchmarks [2] for microprocessors). However, the complete programs in the suite are real and representative, written in an architecture-independent way under the same programming model. We hope that the application suite will be useful to a parallel processing community that all too often finds itself using scattered and disparate toy programs. The use of a common set of applications will also enhance the comparability of results. We expect the suite to evolve with time, and that people will modify the programs for their evaluation studies.

For every application, the paper contains a description and evaluation. The description includes the problem being solved, the principal data structures used, profile information, the structure of parallelism in the program, and some of its static and dynamic characteristics. References to more detailed treatments of the algorithms used are also provided. The evaluation section contains performance results from an Encore Multimax as well as from an execution-driven simulator of an idealized multiprocessor architecture. The Multimax provides information about program execution on a real—albeit small-scale---- machine while the simulator allows us to explore program characteristics in a more architecture-independent fashion with larger numbers of processors. This detailed information about the applications should help users of the SPLASH programs to understand the results and limitations of their studies. The report can also serve as a common reference point for the applications used in a particular study, a convenience for authors and readers alike.

The paper is structured as follows. Section 2 details our method of distributing the application sources and documentation. Section 3 describes the programming model common to all the applications. In Section 4, we discuss the problem domains and solution techniques the applications represent, as well as their behavioral characteristics, to try to place them in the space of parallel applications. Section 5 describes the mechanisms used for evaluating the performance of the programs. Section 6 introduces the format of the individual application descriptions and Sections 7-12 presents each application in detail. Finally, Section 13 offers some concluding remarks.

## 2 Distribution

Application sources and makefiles for the Encore Multimax are obtainable by anonymous ftp from the internet host `mo jave. Stanford. EDU.` The root directory for the applications is `splash,` and every application is contained in a subdirectory entitled with its name as used in this paper. This paper, together with others referenced in it, serves as the documentation for the suite. Questions should be addressed to `splash@mojave.Stanford.EDU.`

## 3 The Programming Model

Most of the programs in this suite are written in C (one is in FORTRAN), using the `parmacs` macros from Argonne National Laboratory [3] for parallel constructs. The programs assume a number of tasks (Unix processes) operating on a single shared address space. Typically, the initial or parent process spawns off a number of child processes, one per additional processor to be used[1]. These cooperating processes are then assigned chunks of work using static scheduling, task queues or self-scheduled loops. The synchronization structures used are locks and barriers. Since Unix process creation and destruction are too expensive to be done frequently, processes are spawned once near the beginning of the program, do their work, and then terminate at the end of the parallel part of the program. Most of the programs were written with machines like the Encore Multimax in mind: bus-based multiprocessors with per-processor caches and uniformly accessible shared memory.

[1] Note that for this reason we use the words process and processor interchangeably in this paper.

# 4 Overview of Application Characteristics

In this section, we briefly discuss the domains of applications covered **by** the SPLASH suite and provide a summary of the execution characteristics of the programs. This information should help the user to select the applications appropriate for her studies, and it provides some idea of the coverage achieved with SPLASH.

Some basic information about the different applications is presented in Table 1. Following the terminology of Dongarra *et al.* [1], our applications can be characterized as whole *applications.* In the absence of the actual workloads that will be run from day to day, individual whole applications are considered to be the most representative benchmarks for evaluation studies, since they include interactions that are not represented in smaller pieces of code but will be present in the real workload of the machine.

Table 1: Basic information about the applications.

| Application | Author | Lang. | Lines | What Application Does |
|---|---|---|---|---|
| ocean | J.P.Singh | Fortran | 3300 | simulate eddy currents in an ocean basin |
| Water | J.P.Singh | C | 1500 | simulate evolution of a system of water molecules |
| MP3D | J.D.McDonald | C | 1500 | simulate rarefied hypersonic flow |
| LocusRoute | J.Rose | C | 6400 | route wires in a standard cell circuit |
| PTHOR | L.Soule | C | 9200 | simulate a digital circuit at the logic level |
| Cholesky | E.Rothberg | C | 2000 | Cholesky factorize a sparse matrix |

We divide our axes of characterization into two categories. The first of these is concerned with the domain of scientific activity that the applications represent. The second deals with behavioral characteristics of the applications.

## 4.1 The Applications and their Domains

Table 2 shows the problem domain of each application, the category of applications it is representative of, and the particular methods or algorithms that are dominant in it. Four of our applications (Ocean, Water, MP3D and Cholesky) are scientific computations and two (LocusRoute and PTHOR) are from the area of computer-aided design. We believe that the current set of applications provides a reasonably wide variety of applications. It is our plan to continue to enhance the coverage of the programs in the furture.

Table 2: Problem domains and techniques represented.

| Application | Problem Domain | Representative of | Algorithms Used |
|---|---|---|---|
| ocean | oceanography | finite differencing CFD, regular grid | near-neighbor, SOR |
| Water | molecular dynamics | N-body | direct, with cutoff radius |
| MP3D | aeronautics | particle-in-cell methods | Monte Carlo |
| LocusRoute | VLSI CAD | standard cell routing | iterative refinement |
| PTHOR | logic simulation | distr. time discrete event simulation | Chandy-Misra |
| Cholesky | matrix factorization | sparse Cholesky factorization | supernodal fanout |

## 4.2 Behavioral Characteristics

In this subsection we outline the behavioral characteristics along which we classify the SPLASH applications. More detailed treatments for each application can be found in Sections 7-12. The characteristics we describe are:

- **Partitioning and Scheduling:** How is the parallel work partitioned among processes? For several of our applications, at least one axis could be found along which to do load-balanced partitioning and scheduling statically (e.g., Ocean, Water, MP3D). In others, the work done is much more dependent on the particular input data set, and dynamic scheduling is implemented with task queues (LocusRoute, PTHOR).

- **Synchronization:** What kinds of synchronization are used to preserve dependences in the application? The statically scheduled programs use only barriers, either explicitly or implicitly in distributed loops. Programs that use task queues preserve dependences through the order in which tasks are enqueued as well as explicitly with barriers. All the programs use locks to provide mutual exclusion.

- **Granularity of parallelism:** What is the grain size of the parallel computation? In task-queue based programs that exploit parallelism at a single level (PTHOR, LocusRoute), a natural way to measure grain size is the average number of cycles needed to execute a task taken off the queue. This granularity may, of course, depend on the input data set and may also have a significant variance within a given set. Quantifying granularity is a little more complicated in the statically scheduled numerical programs (Ocean, Water, MP3D). Here, several computations are performed, each on every particle or point in the problem domain, with barrier synchronizations between some of these large computations. The granularity is therefore best thought of as being proportional to the number of particles/points divided by the number of processors used. However, there typically is some sharing of data by the parallel processes between barrier synchronizations. Another type of granularity relevant to all the applications is the size of critical sections proctected by locks, which is small in all our cases.

- **Computational Scalability:** Does the parallelism in the application scale well for large input sizes? We assume an ideal memory system, thus characterizing only algorithmic or computational speedup. Real memory systems might reduce the scalabilities significantly. With the ideal memory system and large input data sets, all our applications except PTHOR scale very well.

- **Locality of Data Referencing:** How regular and predictable are the access patterns to the various data structures? What kinds of locality does the application afford, and which of these does the program exploit? Given the machine model for which the programs were written (see Section 3), most of them are designed to preserve access locality for *individual* processors in at least one major data structure, in order to improve cache hit rates and thereby lower communication requirements. None, however, make any attempt to exploit geographic locality *between* processors. For example, in Water and MP3D which simulate particles in a region of space, efforts are made to ensure that the same processor always deals with the same particles. However, there is no attempt to allocate particles in adjacent regions of simulated space to adjacent processing nodes on the machine, in order to keep the interactions local in machines that have non-uniform 'distances* between processors. Some of the applications have two or more data structures that are heavily referenced in very different ways. Scheduling for locality in one data structure can make accesses to another non-local (for example, the array of particles and the array representing physical space cells in MP3D); optimizing for both can compromise load-balancing. In these cases, locality is incorporated in accesses to at least one of the major data structures, typically the one for which load-balanced scheduling follows more easily. No attempt is made in the source code to control the memory allocation of shared data structures, as would be desirable for machines in which main memory is distributed among processors.

Table 3 summarizes the first four behavioral characteristics for every application. Locality is more difficult to summarize in a table-especially before the applications and their data structures are even described— so we leave its discussion to Section 6.

The second column of Table 3 reveals a range of scheduling mechanisms, from the fully static to the fully dynamic. In the table, by semi-static scheduling we mean that a task queue model is used, but tasks that use the same data are preferably assigned to the same processor every time. The third column describes the dominant forms of synchronization used to preserve dependences in the program. Note that locks are used in all cases to provide mutual exclusion. An imprecise measure of application granularity is shown in the fourth column; more detailed characterizations are given in Sections 7-12. A wide range of granularity is present, although none of the programs are extremely fine-grained. For the scientific programs whose granularity is proportional to the input data size, it is listed in the table as being large. The computational scalabilities are summarized in the last column of Table 3. Most of the programs scale well with an ideal memory system. Cholesky can scale well given a large input data set. PTHOR speedups are also dependent on the input data set. However, typical circuits being simulated generally have limited activity and thus provide limited parallelism.

Table 3: Summary of some behavioral characteristics.

| Application | Scheduling | synchronization | Granularity | Comp. Scalability |
|---|---|---|---|---|
| ocean | static | barrier | large | good |
| Water | static | barrier | large | good |
| MP3D | static | barrier | large | good |
| LocusRoute | semi-static | task queues, barrier | small | good |
| PTHOR | semi-static | task queues, barrier | small | limited |
| Cholesky | dynamic: task queue | task queue | large | limited; input dept. |

Table 4 summarizes the memory referencing behavior of the programs, all run with 32 processors on the simulator. The programs uniformly exhibit more read-sharing activity than write-sharing.

Table 4: Reference statistics by application.

| Application | Number of Processors | Input Data Set | Cycles (M) | Reads (M) | % Shared | Writes (M) | 9b Shared |
|---|---|---|---|---|---|---|---|
| ocean | 32 | | 2.46 | 11.0 | 88 | 3.17 | 68 |
| Water | 32 | LW112 | 2.16 | 17.1 | 21 | 6.93 | 7.5 |
| MP3D | 32 | 3000 mols | 1.57 | 4.60 | 71 | 2.74 | 80 |
| LocusRoute | 32 | Primary1 | 2.73 | 13.9 | 57 | 3.37 | 31 |
| PTHOR | 32 | risc | 4.59 | 53.9 | 54 | 11.6 | 21 |
| Cholesky | 32 | tk14 | 2.05 | 10.1 | 80 | 3.05 | 42 |

Having characterized the applications, we now describe the methodology used to quantitatively evaluate the behavior of the applications.

## 5 Evaluation

To characterize application behavior, each parallel program was run on an Encore Multimax and an execution-driven simulator of an idealized architecture. The Encore Multimax (chosen due to availability) provides performance results on a real multiprocessor, but is limited to twelve processors. The simulator provides results as well as behavioral statistics for a larger number of processors.

### 5.1 The Encore Multimax

The Multimax we use has twelve NS32332 processors--each with an associated NS32081 floating point unit—connected by a shared bus [4]. Each processor is nominally rated at 2 MIPS, and the bus has a peak bandwidth of 100 Mbytes per second. A pair of processors shares a 32-Kbyte direct-mapped cache with a 4 byte line size. Since the processors are quite slow relative to the rest of the system, the memory and interconnection system is not a significant bottleneck on this machine. Timing measurements were made with no other user applications running on the machine.

### 5.2 The Simulator

There are two parts to the simulator we use: the Tango reference generator [6] which runs the application and produces a parallel memory reference stream, and a memory system simulator which processes these references and feeds timing information back to the reference generator. The simulator runs on a DECstation 5000. The programs were compiled with the cc (Mips Computer Systems 1.31) and f77 (Mips Computer Systems 1.31) compilers, using the -02 level of optimization.

The simulator assumes a perfect memory system for timing purposes (i.e. all memory referencing instructions take a single cycle to complete), yielding what might be called computational or algorithmic speedups without any degradation due to the memory system. We do, however, functionally simulate infinite processor caches for the purpose of tracking miss statistics. The caches are kept coherent by an invalidation-based protocol. Infinite caches are chosen to exclude misses due to limited cache capacity or mapping interference. The only misses observed are due to cold-start and invalidations. For some applications, we eliminate cold misses as well by gathering statistics only after the initialization phase is complete. Note that infinite caches tend to bias miss rates favorably toward a uniprocessor execution, by taking away the advantage of a growing total cache space as the number of processors is increased. Cache lines are 4 bytes long, to avoid false sharing problems. The simulator keeps track of the number and types of references, as well as of misses, invalidations and time spent waiting at synchronization points.

Because a perfect memory system is assumed in measuring execution times, miss rates do not affect performance. Departures from ideal application speedups can be ascribed only to larger wait times at synchronization events, and such overheads as parallelism management and redundant computation. Larger synchronization wart times are caused by load imbalances (in the case of barriers) or contention to *enter* a critical section (in the case of locks), but not by contention for synchronization variables themselves. We present one or both of the following kinds of speedups for every application: *normalized* speedups *are* measured with respect to an efficient sequential program (running on one processor of the parallel machine), while *self-relative* speedups are measured with respect to an execution of the parallel program using a single processor. For each application, we outline the factors that prevent it from attaining ideal speedups.

The large run-time overhead of using the simulator forces us to limit the execution time of the applications. For some classes of applications, this is easy. For programs that simulate physical processes over time, for example, we might execute only a handful of time-steps, rather than thousands. Assuming that succeeding iterations are identical in application behavior, running for a small number of iterations does not introduce much error into our measurements. For other classes of applications, we are forced to reduce the data set to allow the simulation runs to complete in a reasonable amount of time. ‘This may have undesirable effects, such as reducing the degree of parallelism and altering the interactions with the memory system on a real multiprocessor.

## 6 The Application Descriptions

In the following sections we describe each SPLASH application in some detail. We begin with a description of the problem to be solved and then provide details about the parallel program, including principal data structures, program structure and profiling information. We also provide information about how the program is run, what inputs it requires and what output it produces. The final subsection of each application section contains results obtained from running the application on the Encore Multimax and on the simulator, and a brief discussion of these results in the light of application characteristics.

## 7 Ocean

This application studies the role of eddy and boundary currents in influencing large-scale ocean movements. A cuboidal ocean basin is simulated, using a discretized quasi-geostrophic[2] circulation model. Wind stress from atmospheric effects provides the forcing function, and the impact of lateral friction with the ocean walls is included. Bottom friction is set to zero in this application, although its effects can easily be incorporated as well. The sequential program was received from the National Center for Atmospheric Research in Boulder, Colorado.

The simulation is performed for many time-steps until the eddies and mean ocean flow attain a mutual balance. The computational demands are heavy, owing to the large number of time-steps required, the size of the ocean basin, and the increased accuracy obtained by finer discretization. However, the grid-based application is well suited to parallelism.

[2] Geostrophic: relating to the deflective forces caused by the rotation of the earth.

The work done every time-step essentially involves setting up and solving a set of spatial partial differential equations, details of which can be found in [7]. The continucus functions are transformed into discrete counterparts by first-order finite-differencing, and the resulting difference equations set up and solved on two-dimensional fixed-size grids representing horizontal cross-sections of the ocean basin. We simulate a square grid of size 98-by-98 points, and we use the same (constant) resolution in both dimensions. The original sequential program used a block cyclic reduction algoiithm to solve the elliptic equations (see [7]); the parallel programs use an iterative method: Gauss-Seidel with Successive Over Relaxation (SOR) [7, 8].

## 7.1 Principal Data Structures

The principal data structures are two-dimensional arrays holding discretized values of the various functions associated with the model's equations. These include the streamfunctions at the middle of the two horizontal ocean layers and at the interface between them, streamfunction values at the previous time-step (needed to avoid numerical instability in computing the friction terms), the driving functions in the equations, their component terms, and various others. In all, there are 25 such double precision floating point grids, most of them allocated as two-dimensional arrays, but some in pairs in three-dimensional arrays. The two-dimensional array size is statically allocated to be 128-by-128. How much of each array gets used depends on the size of the problem simulated, but the total allocated data space is about 3.2MB.

## 7.2 Structure of the Parallel Program

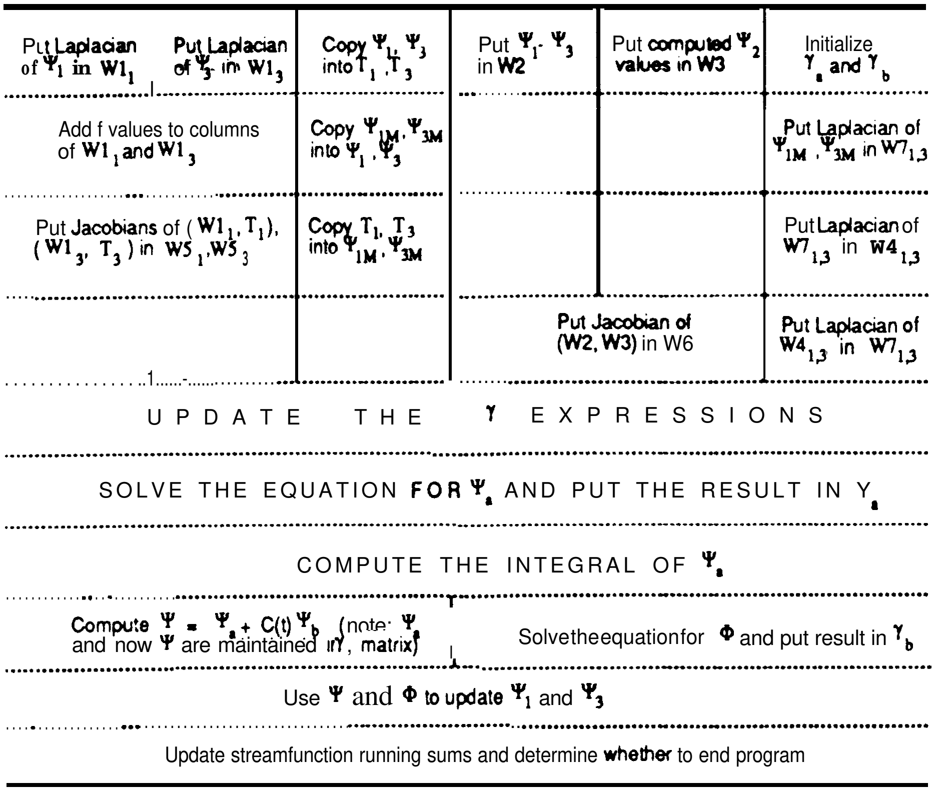
The program begins with initialization and some one-time computation, which includes computing the external forces and solving a single elliptic equation. After this, the outermost loop of the program iterates over a fixed number of time-steps. The high-level structure of the parallel program within a time-step is shown in Figure 1. Grid computations in the same horizontal section in this figure are independent of one another. Those in the same vertical section follow a thread of dependence. Note that only a limited and fine-grained pipelining can be exploited across time-steps, and is ignored here. The $\Psi$ grids in Figure 1 represent the streamfunction values being solved for. $\Psi_1$ is the value at the middle of the upper layer of the ocean, and $\Psi_3$ at the middle of the lower layer. $\Psi_{LM}$ and $\Psi_{LM}$ are streamfunction running sums, while $\Psi_a, \Psi_b$ and $\Phi$ are mathematical functions used in the solution process. The y grids are the right hand sides in the equation system, and the $W$ are temporary workspace arrays. The Jacobians and Laplacians are near-neighbor computations (9-point and 5-point stencils, respectively) with different input and output arrays, while the equation solution is an in-place near-neighbor iteration to convergence with a 5-point stencil. An understanding of the equation system and solution method can be obtained from [7].

### 7.2.1 Partitioning/Scheduling and Locality

The program proceeds from left to right within every horizontal phase of Figure 1. Every grid task in the figure is partitioned among all processors. The non-boundary elements of a grid are split into equal-sized chunks of adjacent columns[3], each of which is assigned to a different process. The scheduling is static: Every process determines the first and last columns of its partition at the beginning of the program and works on those columns of every grid. Boundary elements are partitioned as follows: The processor that works on column `j` of an N-by-N grid also does the computation for the elements `(j ,1)` and `(j, N)`. What remain are the four corner elements, and these are split between the processes with the smallest and largest identifiers. We should mention here that our parallel solution method does not follow the SOR ordering exactly, violating it at inter-partition boundaries.

Locality is provided by ensuring that the same columns are always assigned to the same processor. Near-neighbor computations are kept more local by partitioning in chunks of adjacent columns rather than interleaving processors among columns: The only communication now is at inter-partition boundaries, and can be further reduced by partitioning in square subblocks rather than column strips (not implemented here). All the computations simply sweep through entire grids, yielding no opportunity for blocking within a grid computation.

[3] Partitioning in subdomains that are as close to square as possibk is an attractive alternative that minimizes the interprocessor communication inherent to the application.


Put Laplacian of $\Psi_1$ in $W1_1$
Put Laplacian of $\Psi_3$ in $W1_3$
Copy $\Psi_1$, $\Psi_3$ into $T_1$, $T_3$
Put $\Psi_1$ - $\Psi_3$ in W2
Put computed $\Psi_2$ values in W3
Initialize $\Upsilon_a$ and $\Upsilon_b$
Add f values to columns of $W1_1$ and $W1_3$
Copy $\Psi_{1M}$, $\Psi_{3M}$ into $\Psi_1$, $\Psi_3$
Put Laplacian of $\Psi_{1M}$, $\Psi_{3M}$ in $W7_{1,3}$
Put Jacobians of ( $W1_1$, $T_1$), ( $W1_3$, $T_3$ ) in $W5_1$, $W5_3$
Copy $T_1$, $T_3$ into $\Psi_{1M}$, $\Psi_{3M}$
Put Laplacian of $W7_{1,3}$ in $W4_{1,3}$
Put Jacobian of (W2, W3) in W6
Put Laplacian of $W4_{1,3}$ in $W7_{1,3}$
UPDATE THE $\Upsilon$ EXPRESSIONS
SOLVE THE EQUATION FOR $\Psi_a$ AND PUT THE RESULT IN $\Upsilon_a$
COMPUTE THE INTEGRAL OF $\Psi_a$
Compute $\Psi = \Psi_a + C(t)\Psi_b$ (note: $\Psi_a$ and now $\Psi$ are maintained in $\Upsilon$ matrix)
Solve the equation for $\Phi$ and put result in $\Upsilon_b$
Use $\Psi$ and $\Phi$ to update $\Psi_1$ and $\Psi_3$
Update streamfunction running sums and determine whether to end program

Note: Horizontal lines represent synchronization points among ail processes, and vertical lines spanning phases demarcate threads of dependence.

Figure 1: Parallel structure of Ocean.

### 7.2.2 Synchronization and Granularity

Mutual exclusion, enforced with locks, is required in obtaining a process identifier and in only one other situation in this application: when every process accumulates its private sum into a shared sum in computing a matrix integral. This situation is relatively infrequent, occuring once in the one-time computation, and once every time-step. Synchronization is also needed to preserve dependences between grid computations; this is accomplished by inserting barriers at the end of every phase in Figure 1. Barrier synchronization is overly restrictive here, and can be replaced by more direct inter-processor synchronization if so desired. Barriers are also used when a globally determined value (such as an integral or a flag indicating convergence) is subsequently to be used by all processors.

The granularity of Ocean is measured as the amount of computation per processor between successive barrier synchronizations. While the different grid computations in the program take different amounts of time, the granularity in all cases is proportional to the number of grid points per partition; that is, to the ratio of the number of points and the number of processors. Since we expect this ratio to be large on shared-memory multiprocessors, we call the program large-grained. The granularity of the few critical sections in the program is very small: merely a few machine instructions.

## 7.3 Profile of a Uniprocessor Execution

Most of the execution time of the program is spent in near-neighbor Jacobian and Laplacian computations and in solving the elliptic equations. The initialization and one-time computation are negligible. Figure 2 shows the variation, with problem size, in the amount of time spent in the solver and in the rest of the program. Notice that the uniprocessor execution time of the program grows linearly with the number of grid points.

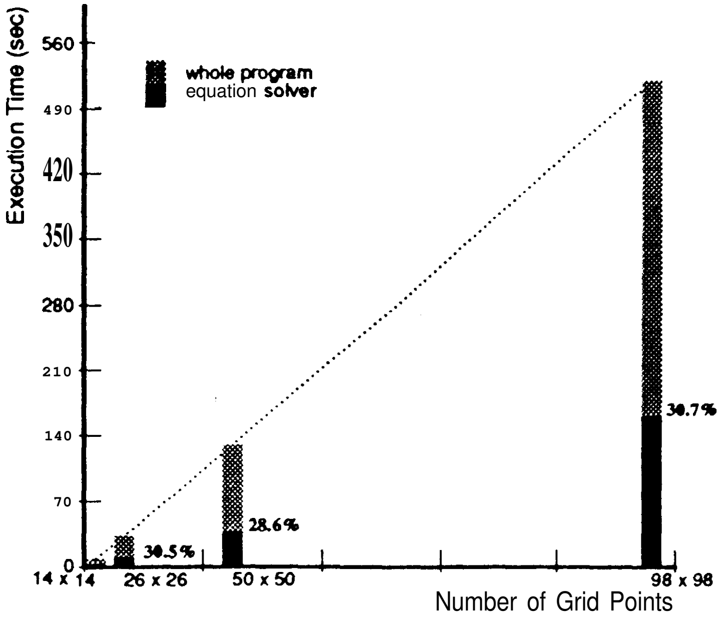


Figure 2: Ocean: uniprocessor execution time versus number of grid **points.**

## 7.4 Running the Program

The program is run by typing the command: `OCEAN.` Execution parameters are specified in an input file.

### 7.4.1 Input and Output

Ocean reads four input values from the standard input device, in the following order.

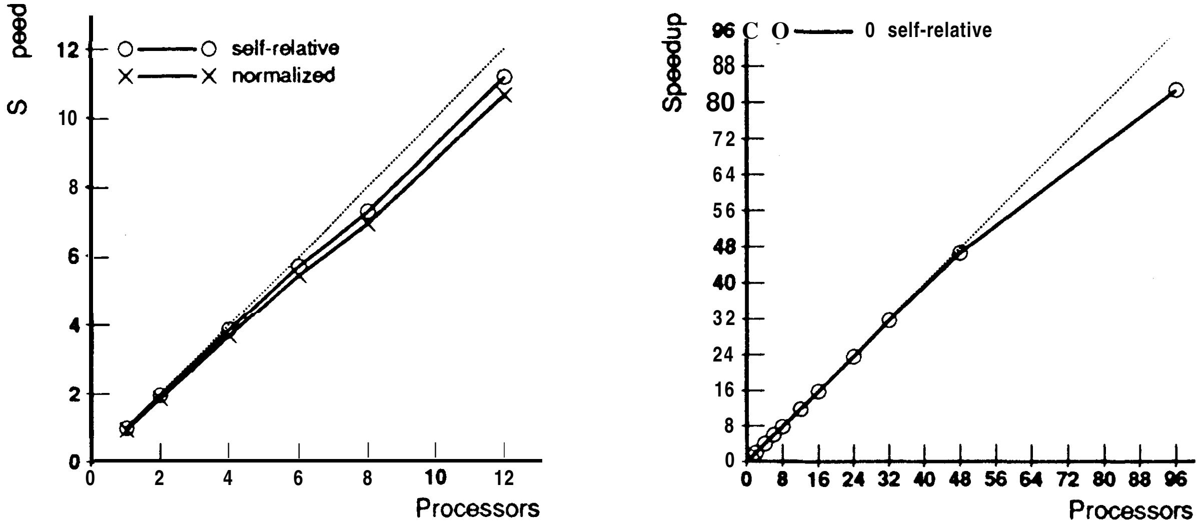
- the number of processes to be used.
- the size of the grid (assumed square) to be simulated `(IM).` This size is specified as the number of elements in either dimension, ***not*** the total number of points on the grid. Unless the PARAMETER statements in the source are modified and the program recompiled, `IM` must not be greater than 128; it must never be less than 2. The application also assumes that `(IM-&-the` matrix dimension excluding boundary elements-is a multiple of the number of processors used.
- the tolerance for convergence of the iterative equation solver (suggested value = $10^{-7}$).
- the SOR parameter $\omega$ for the equation solver (suggested value = 1.15).

The program prints out some timing results as its only output. Typical simulations in the application domain would be performed over hundreds of simulated days. However, since the work done in every time-step is essentially identical, significantly fewer time-steps can be used for parallel benchmarking. The program currently simulates 6 time-steps (2 days).

## 7.5 Results

We simulate a grid size of 98-by-98 points. The timer is started when the first time-step begins, omitting process creation and cold-start cache misses in the one-time computation, and is stopped just before printing the output at the end of the program execution. Note that the execution time of the SOR equation solver depends on the tolerance used to determine convergence. Since the sequential program uses a direct solver rather than SOR, normalized speedups are sensitive to the tolerance used in the parallel program.

Normalized and self-relative speedups obtained on the Multimax are shown in the left graph of Figure 3. The speedup scales quite linearly with the number of processors. Beyond six processors, the speedup is reduced to some extent by the artifact of two processors sharing a cache, thus reducing the effective cache space of each and increasing mapping collisions.



Encore Multimax

Simulator

Figure 3: Ocean Speedups: Multimax and Simulator.

Measurements obtained with the simulator are shown in the right graph of Figure 3 and in Table 5. The speedup is almost ideal upto 48 processors, and falls off a little when only one column is assigned to every processor, making the overheads of synchronization and parallelism management more significant. With 48 processors, only 1.2% of the execution time is spent waiting at barriers, and only 0.1% at critical sections. With a realistic memory system on a modem multiprocessor, cache miss rates would significantly reduce the speedup from the ideal memory system results shown here (a quick comparison of speedups will show, however, that this is not the case on the Multimax). Almost all the misses shown in the table are due to nearest-neighbor computations at inter-partition boundaries. The narrower the partitions, the more the misses and the smaller the speedup. The miss rate can be reduced by using square sub-blocks-rather than column strips-as partitions, or by using other locality-enhancing transformations. Note that the problem size we have used is considered quite realistic (being significantly larger than the 26-by-26 size hard-coded into the sequential program we received from NCAR). On high-performance multiprocessors, however, it is reasonable to consider problem sizes that would sweep the caches of a small machine on every grid computation.

## 8 Water

This N-body molecular dynamics application evaluates forces and potentials in a system of water molecules in the liquid state. The computation is performed over a user-specified number of time-steps, hopefully allowing the system to reach a steady state. Every time-step involves setting up and solving the Newtonian equations of motion for water molecules in a cubical box with periodic boundary conditions, using Gear's sixth-order predictor-corrector method [9]. The total potential is computed as the sum of intra- and intermolecular potentials. To avoid computing all the $\frac{n^2}{2}$ pairwise interactions among molecules, a spherical cutoff range is used with radius equal to half the box length. More recent algorithms to speed up the computation of N-body interactions (such as the Greengard-Rokhlin method [10]) are not used. Double-precision accuracy is required for this simulation,

Table 5: Ocean Information (Simulator).

| Number of Processors | Miss Rate (%) | synchronization Waiting Time (%) |
|---|---|---|
| 1 | 0.00 | 0.00 |
| 2 | 0.13 | 0.56 |
| 4 | 0.33 | 0.58 |
| 6 | 0.53 | 0.61 |
| 8 | 0.72 | 0.62 |
| 12 | 1.10 | 0.65 |
| 16 | 1.47 | 0.70 |
| 24 | 2.22 | 0.81 |
| 32 | 3.04 | 0.94 |
| 48 | 4.43 | 1.35 |
| 96 | 8.88 | 3.66 |

which can be used to predict a variety of static and dynamic properties of liquid water. Further documentation of the program can be found in [11], and details of the physical models in [12, 13, 14].

The sequential program, written in FORTRAN, is one of the Perfect Club set of supercomputing benchmarks [15]. The parallel program is written in C, with significantly modified data structures.

## 8.1 Principal Data Structures

The main data structure used in the Perfect Club benchmark is a large, one-dimensional array `called VAR`. The array is divided into eight sections, each representing a variable in the system: the first seven for the displacement ($f(x)$) and its first six derivatives, and the last for the computed forces. Every section is further subdivided into three spatial directions, with every direction having an entry for every atom of every molecule. For better cache behavior with large line sizes, the parallel program restructures VAR to be an array of structures, each holding all the data for one molecule. Every molecular structure has a three-dimensional array, indexed by variable type, spatial direction and atom number within the molecule, as well as a smaller array with three entries per molecule. The data memory requirement of the program is about 750 bytes per molecule. For our problem size of 288 molecules, the shared data set size of the application is about 220 Kbytes.

## 8.2 Structure of the Parallel Program

The program begins with some initialization and one-time computation. This includes reading the initial displacements and velocities of the molecules to be simulated. Additional processes are then spawned. In the sequential program, the equations of motion are solved by iterating over the loop in Figure 4 for a user-specified number of time-steps, with the total energy of the system being computed and output once every few time-steps (again a user-specified number).

Only very fine-grained pipelining is possible across time-steps; none of it is implemented. The work in a time-step can be divided into a set of tasks, each a computation over all molecules in the system. After an interprocedural analysis of high-level dependences between tasks, and the use of static scheduling to minimize explicit inter-task synchronization, the computation within a time-step can be mapped on to a set of fully parallel phases.

### 8.2.1 Partitioning/Scheduling and Locality

Every box in the flowchart of Figure 4 is partitioned among all processes. For the intramolecular computation, parallelism within a piece of work is mainly implemented by assigning every processor a set of molecules that

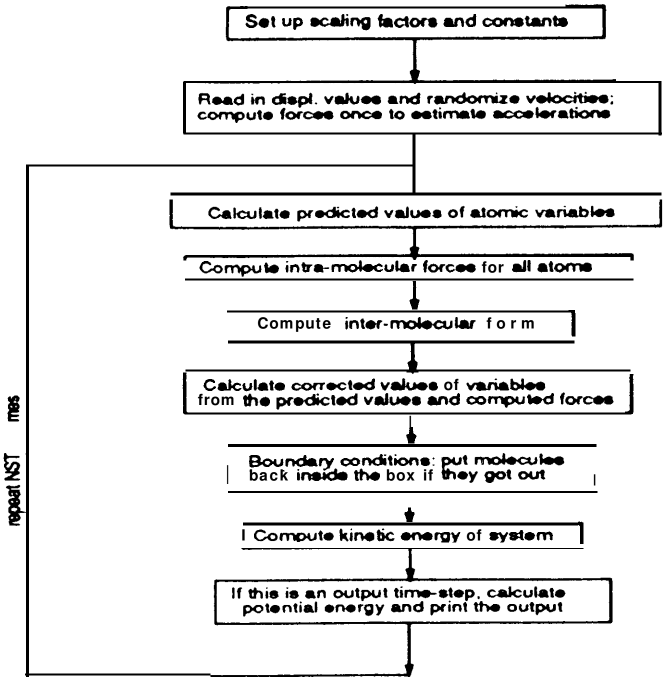

Set up scaling factors and constants
Read in displ. values and randomize velocities; compute forces once to estimate accelerations
Calculate predicted values of atomic variables
Compute intra-molecular forces for all atoms
Compute inter-molecular form
Calculate corrected values of variables from the predicted values and computed forces
Boundary conditions: put molecules back inside the box if they got out
Compute kinetic energy of system
If this is an output time-step, calculate potential energy and print the output
repeat NST mes

Figure 4: Structure of **the Water program.**

are located next to each other `in the VAR` array (which does not imply that they are physically close together in space). Locality is incorporated by assigning a processor the same partition in every computation; that is, static scheduling is used. In the intermolecular case, a processor is responsible the interactions of each molecule in its partition with the $\frac{n}{2}$ molecules following it, with wraparound at the end. Every one of the $p$ processors therefore examines $\frac{n^2}{2p}$ interactions of the pool of $\frac{n^2}{2}$, including as few nonlocal interactions as possible. Note that although a cutoff radius used in determining whether to actually compute an intermolecular interaction, all pairs of molecules have to be examined since no spatial data structure is maintained. The cutoff radius also gives rise to another form of locality: locality in physical space. Since molecules have greater interactions with other molecules that are close to them in space than with those that are without the cutoff radius, we would like the partitioning to assign roughly an equal number of actually computed interactions to every processor. Since every interaction updates the force locations of each interacting molecule, we would also like molecules assigned to the same processor to be physically close as well. This version of the application, however, makes no explicit attempts to incorporate locality in physical space. If the molecules in the input data set are not appropriately ordered, both load-balancing and data localitv can be compromised

### 8.2.2 Synchronization and Granularity

Barriers are used to maintain dependences where static scheduling doesn't suffice. The barriers used to separate the parallel phases can be replaced by more specific processor-to-processor synchronization, but this is not implemented in the current program. Barriers also separate successive time-steps. Other than in obtaining process identifiers, locks are required in two situations:

- A global running sum is computed every time-step, and is updated several times during the intramolecular and intermolecular force computations. In the parallel program, every process has its own private running sum, and locks are used to accumulate this into the shared sum at the end of the time-step.
- Since different processes may simultaneously be computing intermolecular interactions involving the same molecule, updates to the force locations of all atomsmust be mutually exclusive. This implementation uses locks at the granularity of individual molecules.

Granularity in Water is measured as the amount of computation per processor between successive barrier synchronizations. Since the number of molecules simulated is likely to be much larger than the number of processors used, the granularity is large ($O(\frac{n^2}{p})$ in the intermolecular interactions). All the critical sections in the program are quite small: no more than a few machine instructions.

## 8.3 Profile of a Uniprocessor Execution

Almost all the execution time of the program is spent in computing intermolecular interactions, this computation being $O(n^2)$ in the number of molecules as opposed to $O(n)$ for the intramolecular computation. The initialization and one-time computation are negligible. Figure 5 shows the variation, with problem size, in the amount of time spent in intermolecular interactions and in the test of the program, timing the **program as** in Section 11.5.

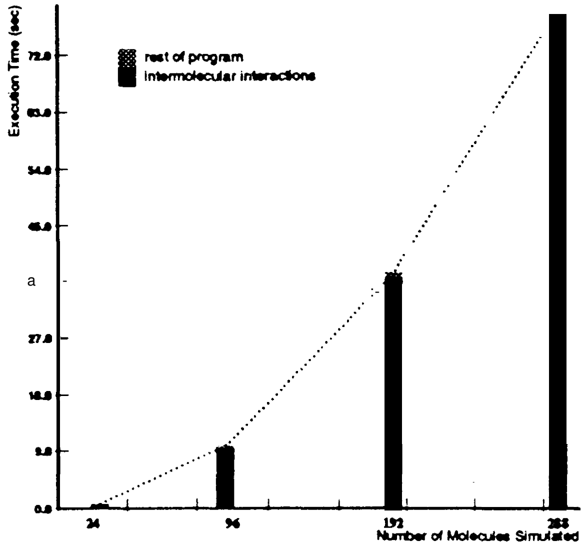


Figure 5: **Water:** uniprocessor execution time versus number of grid points,

## 8.4 Running the Program

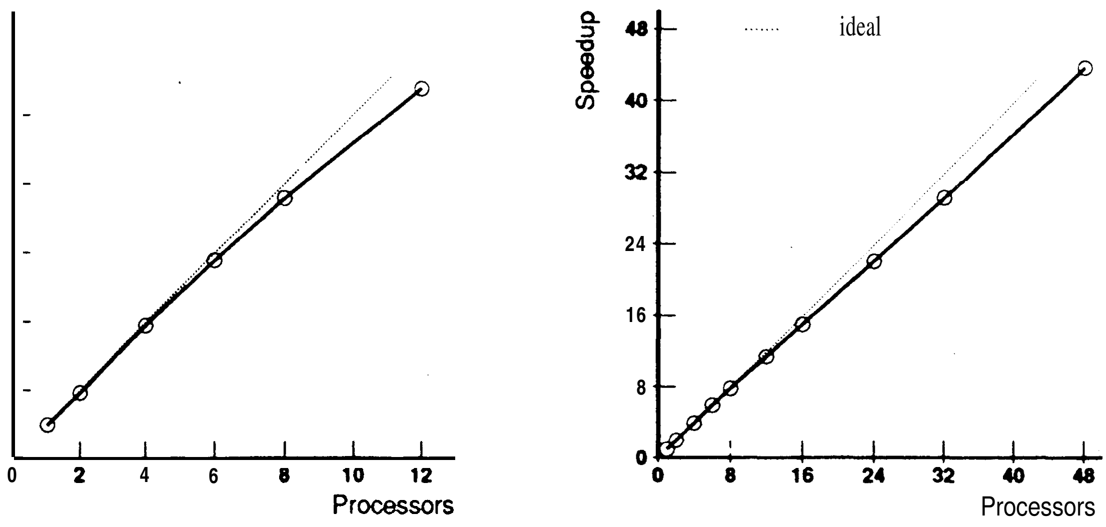
The program is run by typing the command: `WATER` Execution parameters are specified in an input file.

### 8.4.1 Input and Output

Two input files are read, their names being hard-coded into the program. `LWI` **12** is a file that contains the initial displacements for all atoms in all three spatial directions. This file--taken from the Perfect Club-provides data for 343 molecules, listing **first** all the x displacements, then all the y and all the **z.** Since the medium being simulated is liquid water, the distribution of displacements is quite uniform. Note that the cutoff radius and the length of the "box" containing the water molecules are computed from the number of molecules **used,** so that one should take this into account when generating one's own input files. In particular, note that it does not make much sense physically to use the input file provided for numbers of molecules too much smaller or larger than 343. The program provides the option of not reading an input file but generating its own initial displacements as long as the number of molecules **used** is a perfect cube. These displacements place the molecules on **a crystalline** lattice, which is an unrealistically uniform distribution. The file `random. in` contains pseudo-random numbers used to initialize particle velocities. The program also reads some execution parameters from the standard input device. These include the number of molecules to be simulated, the interval (in time-steps) at which to compute potential energy and print results, the number of processors to be used, and the parameter that **specifies** whether or not the displacements should be read from an input file, among others. All input parameters are described in the code listing, and a sample input file (called sample.in) is provided. The program writes its output to a file called LW06. Some timing results are also written on the standard output device.

## 8.5 Results

The problem size we use in our measurements (288 molecules) is almost as large as the Perfect Club provides input for (343 molecules). It is essentially the largest size, still divisible by all the numbers of processors we use, that can use the input data set provided with the Perfect Club benchmarks. We expect real uses of the application to simulate a much larger number of molecules. The timer is started when the second time-step begins and is stopped just before printing the output at the end. Process creation and most cold-start cache misses are omitted since we simulate only 2 time-steps rather than the 100 in the original sequential program. Note also that the potential energy calculations, performed once every 10 time-steps in the original program, are omitted in our measurements. They are, however, structured very similarly to the intra- and inter-molecular force computations. Only self-relative speedups are presented; normalized speedups are not significantly different.



Encore Multimax

Simulator

Figure 6: Water Speedups: Multimax and Simulator.

Speedups obtained on the Multimax are shown in the left graph of Figure 6. Measurements obtained with the simulator are shown in the right graph of Figure 6 and in Table 6. Load imbalance is not a significant problem, even though a cutoff radius is used in computing interactions, since the distribution of water molecules in the liquid state is fairly uniform. Nonuniform particle distributions would complicate the relationship between data locality and load balancing. With 48 processors, the average processor spends only 0.2% of its time waiting at barriers, and negligible time at critical sections. Miss rates are very small with infinite caches, although shared miss rates are significantly higher. This is because only about 20% or the references are to shared data. With more realistic problem sizes, however, the miss rates due to replacement in finite caches-as computations sweep across the molecules---can become significant.

# 9 MP3D

MP3D solves a problem in rarefied fluid flow simulation. Rarefied flow problems are of interest to aerospace researchers who study the forces exerted on space vehicles as they pass through the upper atmosphere at hypersonic speeds. Such problems also arise in integrated circuit manufacturing simulation and other situations involving flow at extremely low density.

Table 6: Water Information (Simulator).

| Number of Processors | Miss Rate (%) | Synchronization Waiting Time (%) |
|---|---|---|
| 1 | 0.00 | 0.00 |
| 2 | *0.04* | 0.19 |
| 4 | 0.63 | 0.32 |
| 6 | 0.98 | 0.18 |
| 8 | 1.05 | 0.26 |
| 12 | 1.11 | 0.39 |
| 16 | 1.16 | 0.37 |
| 24 | 1.24 | 0.33 |
| 32 | 1.33 | 0.29 |
| 48 | 1.48 | 0.20 |

Under such conditions, the discrete particle nature of the medium becomes significant, so these studies cannot rely on traditional fluid flow models (such as the Navier-Stokes equations) which assume a continuous medium. Monte Carlo methods, such as the one used in MP3D, have been developed as an alternative. These methods simulate the trajectories of a collection of representative molecules, subject to collisions with boundaries of the physical domain, objects under study, and other molecules. After a steady-state is reached, statistical analysis of the trajectory data produces an estimated flow field for the configuration under study. To obtain accurate results, such methods require large amounts of computation. Vectorized and parallelized codes have, therefore, been developed [17].

MP3D employs five degree-of-freedom simulation of idealized diatomic molecules in a three-dimensional ***active space.*** There are three translational freedoms and two rotational energy modes. The active space is a rectangular tunnel with openings at each end and reflecting walls on the remaining sides. The object being studied is represented as a set of additional boundaries in the active space. Molecules generally flow through the tunnel in the positive $x$ direction. Exiting molecules are reused after being thermalized to the free stream temperature and randomly distributed near the entrance to the tunnel. The thermalization velocities are copied from a small ***reservoir*** of molecules that are kept at the upstream temperature. The reservoir molecules are subject to motion and collisions among themselves, but are isolated from the active molecules by being located in a separate ***reservoir space.***

For the purposes of efficient collision pairing, the active space is represented as a three-dimensional ***space array*** of unit-sized ***cells.*** Molecules can move among cells, but are only eligible for collision with other molecules occupying the same cell at that time. Molecular collisions are statistically determined using a computed collision probability, conservation laws, and a table of collision outcomes, representing all 3840 permutations of the degrees of freedom of the molecules.

MP3D was developed and initially parallelized in the Aeronautics and Astronautics department at Stanford. Several enhanced versions have since been developed, and a restructuring study is described in [16].

## 9.1 Principal Data Structures

The amount of data accessed by MP3D is largely determined by the number of molecules simulated. The user specifies the initial number of active molecules as an argument to the application, and over the course of a simulation run the number of active molecules typically increases by about 25%. The data requirements are also affected to a lesser extent by the dimensions of the active space and the number of the internal boundary conditions. There are static limits on all the above parameters.

Two large arrays of structures account for more than 99% of the static data space used by MP3D. The first one stores the state information for each molecule, and occupies 36 bytes per molecule. The second one stores the properties of each cell in the active space, and requires 40 bytes per cell.

## 9.2 Structure of the Parallel Application

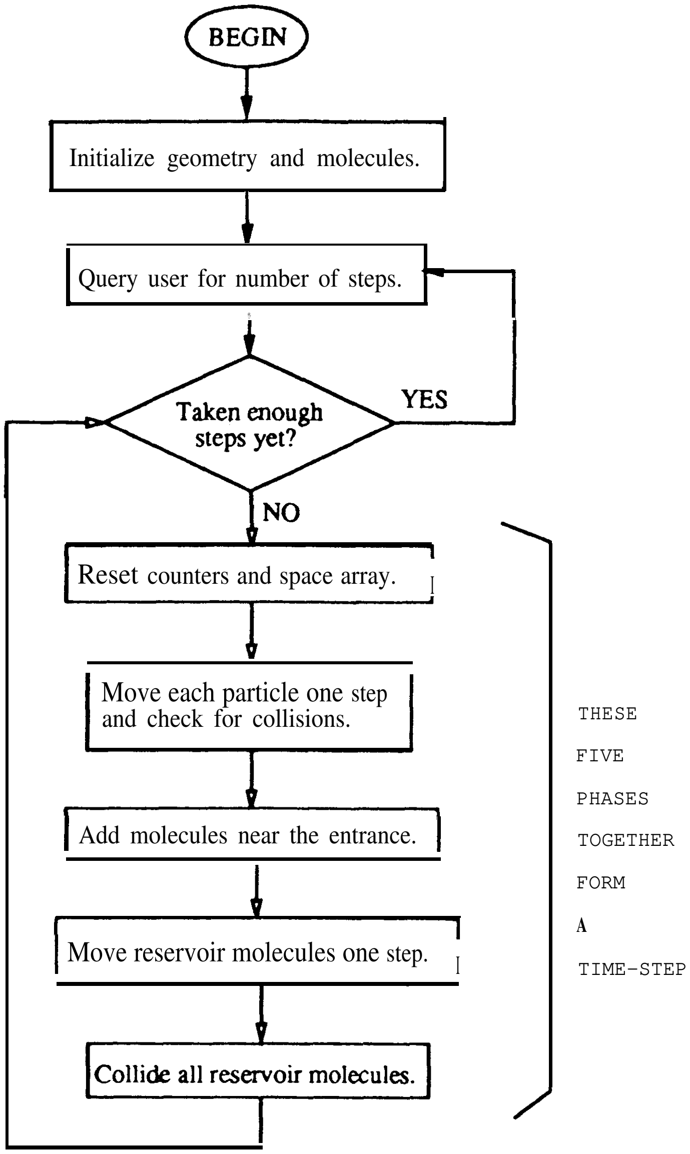

BEGIN
Initialize geometry and molecules.
Query user for number of steps.
Taken enough steps yet?
YES
NO
Reset counters and space array.
Move each particle one step and check for collisions.
Add molecules near the entrance.
Move reservoir molecules one step.
Collide all reservoir molecules.
THESE
FIVE
PHASES
TOGETHER
FORM
A
TIME-STEP

Figure 7: Flowchart of MP3D.

A high-level flowchart of MP3D is shown in Figure 7. Execution begins with the initialization of the data structures. The number of molecules is read from the command line, and the geometry of the problem is read from a separate input file. The application enters a user interface, which determines the number of time-steps to advance the simulation. It then iterates over the time-steps and returns to the user interface.

Each time-step has ***five*** basic phases: ***initialize, move, add, reservoir-move,*** and ***reservoir-collide.*** In the ***initialize*** phase, the space array, the collision counters, and the cell population counters are reset and the collision probabilities for each cell are computed. In the Move phase, the molecules are moved according to the equation $\vec{x} = \vec{x} + \vec{v}\, A\, t$, ***the cell*** statistics ***are*** updated, and collisions with boundaries and other molecules are performed incrementally.[4] In the *add* phase, molecules exiting the active space are recycled and new molecules initialized at the entrance. Reservoir molecules are moved and collided in the ***reservoir-move*** and ***reservoir-collide*** phases, respectively.

[4] **The incremental collision scheme is not sufficiently random, and in recent codes this technique has been replaced by a separate collision phase.**

### 9.2.1 Partitioning/Scheduling and Locality

The work is partitioned by particles and statically scheduled on processors. A given particle, therefore, is always moved by the same processor. In contrast, the access patterns to the space cells depend on the positions of the molecules and thus show much lower processor locality. It is possible for two processors to access a given space cell during the same time-step, if each is moving a molecule in that cell.

### 9.2.2 Synchronization and Granularity

Barriers that are used to synchronize between phases account for most of the synchronization needed. There is only one significant dependence within a phase: in the add phase, the computation of the number of molecules to be added must be completed before the add **loop begins. This** version of the code normally has no locking activity, although it does have a number of minor race conditions, most notably in the global event counters and the space array. The likelihood that these races will significantly affect the results is small, but they can be eliminated at some performance cost by compiling with the LOCKING option.

Granularity in MP3D is determined by the amount of work a processor does between barrier synchronizations. This granularity varies with the phases and is largest in the *move* phase. Since the granularity is proportional to the number of particles assigned to each processor, it is typically very large.

## 9.3 Profile of a Uniprocessor Execution

An execution of a 3000-molecule, 5-step test problem was profiled on a Digital DECstation 3100 using the Unix *pixie* and *prof* utilities. With a full-scale problem, the number of molecules and the number of time-steps would be much larger. The initialization time would then be relatively insignificant. The Table below shows the breakdown of time spent in the simulation section.

Table 7: MP3D Simulation Section Profile.

| Phase | % of parallel section |
|---:|---:|
| move | 93.0 |
| reset | 5.0 |
| reservoir-move | 0.85 |
| reservoir-collide | 0.88 |
| add | negligible |

The majority of the execution cycles were spent in the move phase. The time spent in the add phase was negligible. Of the cycles spent in the move phase, only 0.75% were for collisions between molecules and only 1 .0% were for collisions with boundaries. This is probably because of the trivial input geometry and the small number of molecules used, and is probably not representative of realistic MP3D runs.

## 9.4 Running the Program

MP3D takes three command-line arguments. The first two are the initial number of molecules and the number of processors, in that order. The third argument, which is optional, is the name of the input file containing the problem geometry. If no file is specified, the application attempts to open a file *named test.geom* in the current directory.

The application is designed to be interactive, with **a very** simple command-line interface. Typing a number causes the program to iterate for that number of time-steps. Typing **'q'** causes some statistics on cell populations to be printed. Typing 'e' terminates the application. The desired command sequence may be stored in **a** file for batch-mode operation.

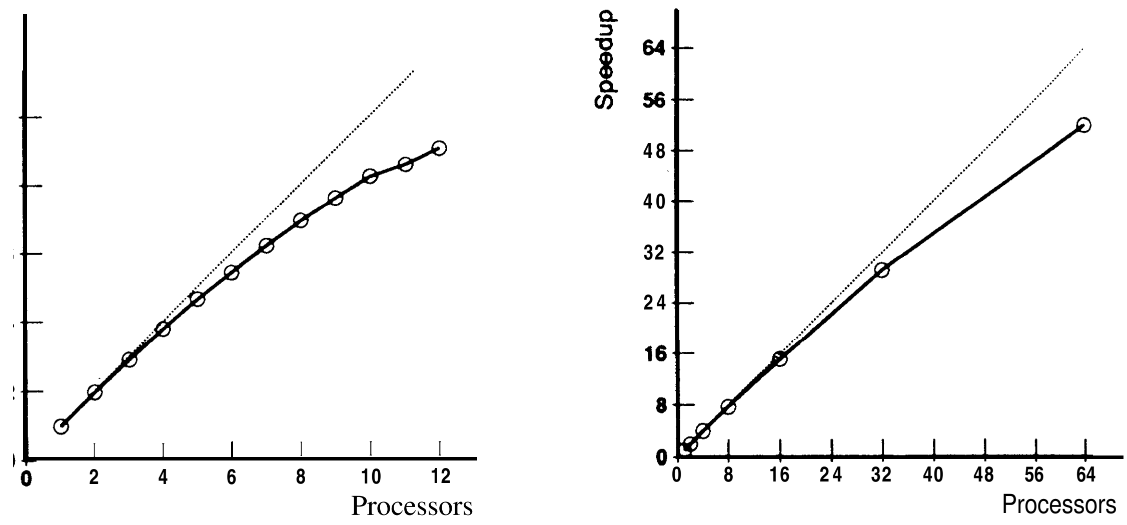
### 9.4.1 Input and Output

The file *test.geom* that we provide contains the geometry for the flat sheet problem used throughout this report, which is described in Section 9.5. Other geometries may be substitued for this. After each period of simulation, the application prints the execution time.

## 9.5 Results

Execution times for MP3D are measured from the point immediately after all processes are created to the time when the last created process completes. All speedups reported are self-relative.

We only use 3000 particles, to reduce simulation time. The geometry used is a 14x24x7 (2646-cell) space containing a single flat sheet, which is placed at an angle to the free stream. This geometry includes 27 internal boundary conditions. We would, however, advise against running MP3D with such a small problem to make useful conclusions. In practice, this application would be run with as many molecules as could be fit in the machine's memory, **and** with more realistic geometries. The Multimax runs were for 50 time-steps, the simulator runs for only 5.



Encore Multimax

Simulator

Figure 8: MP3D Speedups: Multirnax and Simulator.

Table 8: MP3D Information (Simulator).

| Number of Processors | Miss Rate (%) | Synchronization Waiting Time (%) |
|---|---|---|
| 1 | 0.73 | 0.00 |
| 2 | 11.22 | 0.33 |
| 4 | 16.66 | 1.04 |
| 8 | 19.42 | 2.53 |
| 16 | 20.89 | 3.04 |
| 32 | 21.63 | 3.35 |
| 64 | 22.09 | 5.02 |

Speedups obtained on the Multimax are shown in the left graph of Figure 8. Note that there is significant deviation from ideal speedup by the time 12 processors are used. A speedup curve on the simulator is shown in the right graph of Figure 8. For 64 processors, the speedup ratio is 52. Table 8 presents miss rates and synchronization waiting times. In the 64-processor run, only 0.76% of the time is spent waiting on locks, and 4.26% at barriers. With a real memory system, miss rates would be the limiting factor on the speedup gained by MP3D. In the 64-processor run, the overall miss rate was 22.1%. These misses are almost entirely invalidation misses. Since each molecule is assigned to a fixed processor, the space array is responsible for most of the misses. Assigning regions of the space array to different processors, as was done in [16], is one approach to reducing the number of space array misses.

With a realistic problem, we would expect the total data space used by the application to be much larger than the available cache space. This means that the application basically sweeps the caches during each time-step, resulting in high miss rates due to replacement.

# 10 LocusRoute

LocusRoute [18, 19, 20] is a commercial quality VLSI standard cell router. It is used to evaluate standard cell circuit placements by routing them efficiently and determining the area of the resulting layout. To minimize area, the program tries to route wires through regions (routing cells [5]) which have few other wires running through them. It calculates a cost function for each route being considered for a wire, and uses the route with the least cost. The cost function is the number of wires already in the routing cells that this wire will pass through.

LocusRoute affords parallelism at many levels. A circuit is made up of many wires that can be routed in parallel. Routing a wire requires determining paths for all the independent two-pin segments of the wire, again potentially in parallel. To choose the best path for each two-pin segment, several routes for the segment must be evaluated; this route evaluation can also be performed in parallel. Thus, depending on the number of processors available and the task granularity supported efficiently by the target machine, different parallelizing strategies can be chosen.

## 10.1 Principal Data Structures

LocusRoute's main data structure is the cost array. This array keeps track of the number of wires running through each routing cell of the circuit. The vertical dimension of the array is the number of routing channels in the circuit, and the horizontal dimension is the width of the circuit in routing cells. Figure 9 shows a standard cell circuit and one of its wires, with the corresponding cost array. The highlighted portions of the cost array will be updated if this route is chosen. Each cost array element consists of two integers: the numbers of wires running horizontally and vertically through the routing cell. The amount of shared memory required for the cost array (in bytes) is therefore 8 times the number of routing cells in the circuit. For Primary2.grin, the largest of the benchmark circuits with 1290 routing cells in each of 20 routing channels, this is about 200 Kbytes.

Data describing the wires' pin positions and current routes is also stored in shared memory. For Primary2.grin, with 3817 wires, the data describing the (fixed) pin positions of the wires amounts to about 1 Mbyte. However, since this data is only written once as the input circuit is read, it does not have a large effect on the memory referencing behavior of the application. A wire's current route is stored as a 20-byte structure for every straight segment of the wire. A two-bend[6] route has 3 straight segments. There are $n - 1$ two-bend routes for a wire with $n$ pin-groups [7]. The amount of route memory required is therefore roughly proportional to the number of pin-groups in the circuit, and amounts to about 600 Kbytes for Primary2.grin. This data is also not accessed nearly as often as the cost array, making the cost array by far the most important determinant of memory referencing behavior.

[5] A routing cell is a fixed-width section of a routing channel for wires in a standard cell circuit. Its width is usually the minimum width of a logic element.

[6] LocusRoute allows a maximum of two bends in a two-pin wire segment

[7] A pin-group is a set of physical pins representing a single logical pin; any pin in a group may be chosen for a given wire route.

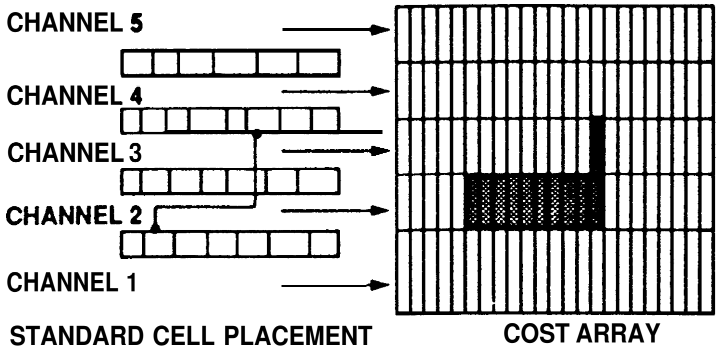


Figure 9: LocusRoute: Standard Cell Circuit and Corresponding Cost Array.

## 10.2 Structure of the Parallel Program

The program begins by reading the input files, initializing data structures, and spawning additional processes (see Figure 10). Every wire laid down increases the cost functions for wires later routed through the same routing cells. Thus, the quality of the routing is dependent on the order in which wires are laid down. Note that this order is nondeterministic in the parallel program. To reduce the dependence of routing quality on the order of routing, two iterations are performed[8]. During the second iteration, the previous route of the wire is "ripped up" (the corresponding cost array entries decremented) before the new route is done. For every wire, the processor computes the minimum spanning tree of the points being connected, in order to break the wire down into two-point segments. Each two-point segment is routed by generating possible permutations among equivalent physical pins in the two pin-groups, and evaluating the quality of the routes for different permutations. Finally, the processor chooses the lowest-cost route, and increments the cost array entries along it. When both iterations of wire routing have been completed, a single processor writes the final routes chosen for the wires to an output file and computes the quality of the final routing.

### 10.2.1 Partitioning/Scheduling and Locality

Task partitioning and scheduling can be manipulated by changing parameters in an input parameter file. The user can set the number of processors working on wire, segment, and route parallel tasks. Since the wire tasks are of the coarsest granularity, it is generally most effective to devote all the processors to routing entire wires. The other axes of parallelism will be potentially more useful as the number of available processes approaches the number of wires to be routed The results shown in Section 10.5 were gathered using all processors as wire processors, and the discussion that follows focuses on this approach.

Partitioning can have several variations. The results shown in Section 10.5 use a "geographical'* method of wire partitioning. In this method, the circuit is divided into regions, as specified by the user in the parameter file. Every region has a task queue associated with it, and processes are assigned to regions as part of the initialization. Wires are placed in the task queue of the region which contains the wire's leftmost pin. A process always checks for work in its own task queue first, but may check other task queues when its queue is empty. Geographic scheduling improves the caching behavior of the program by increasing locality. It also improves the quality of the resulting routing (see section on synchronization). If geographic scheduling is not chosen as an input parameter, the default is to have a single task queue for wires which is accessed by all processors. In general, the single task queue yields slightly poorer performance and quality.

[8] More iterations can be performed, but they typically do not improve the quality d the routing significantly. The number of iterations can be specified in a parameter input file.

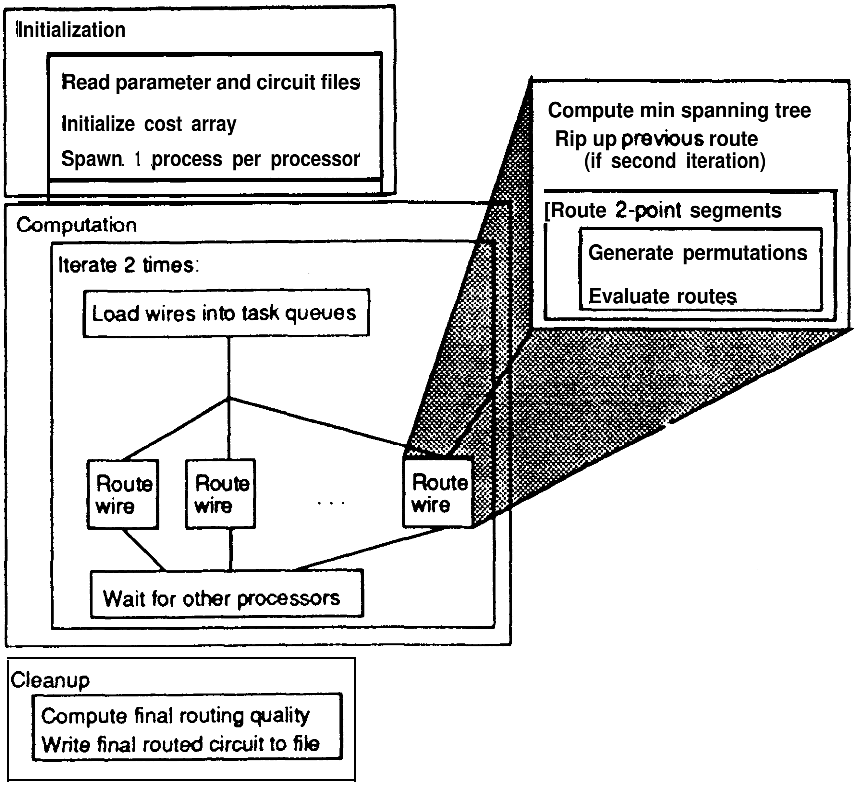


Figure 10: Execution Flowchart for LocusRoute.

### 10.2.2 Synchronization and Granularity

The principal use of synchronization in this version of LocusRoute is for task queue management. One lock is used per task queue to guarantee mutual exclusion. A barrier is used to separate the iterations of wire routing, but no barriers are used within an iteration. Note that parallelism allows several wires to be routed simultaneously, thus altering the order of a uniprocessor execution. LocusRoute can also tolerate the further relaxation of not locking the cost array (as in this version), thus potentially using stale information at times and causing a small degradation in routing quality. The geographic partitioning and scheduling we use tends to minimize this quality degradation and obviates the need for locking.

The granularity of LocusRoute is the time taken to route a single wire. This time is proportional to the area of the wire's bounding box and is only in this way dependent on the size of the input circuit (small circuits are unlikely to have wires with very large bounding boxes). Measurements on two circuits of different sizes revealed very similar granularities, each averaging about 104 simulator cycles and varying by less than 5% around this number.

## 10.3 Profile of a Uniprocessor Execution

Route evaluation is the most time-intensive activity in LocusRoute. The time required to evaluate possible routes for a wire is proportional to the area of that wire's bounding box. Table 9 shows a breakdown of uniprocessor execution tune spent in the major routines. The first input circuit (bnrE) has 511 wires in a 341 grid by 10 channel area. The second circuit (Primary1) is larger, with 1266 wires in an 481 by 18 area.

## 10.4 Running the Program

The program is run by typing the command:

```
LocusRoute CktInputFile ParameterFile OutputFile [#WireProcs [#RouteProcs]]
```

Table 9: LocusRoute Execution Profile.

| | bnrE | Primary1 |
|---|---|---|
| Routine | % of total | % of total |
| RegularEvaluateRoute | 32.7 | 60.0 |
| ProcessDensity | 7.9 | 5.8 |
| RipUpCostArray | 4.4 | 3.3 |

The two optional arguments allow the user to override the number of processors specified for wire and route parallelism in the parameter file.

### 10.4.1 Input and Output

The CktInputFile specifies the size of the standard cell circuit being routed, the number of wires in the circuit, and the positions of the pins each wire connects. Three circuits are provided with the application: bnrE with 511 wires and a 341-by-10 cost array, Primary1.grin with 1266 wires and a 481-by-18 cost array, and Primary2.grin with 3817 wire-s and a 129%by-20 cost array. The ParameterFile contains the parameter names for which the user wants to override the default values, together with the new values. The parameter file we provide uses only geographic scheduling and divides the ciruit into 15 regions. If more than 15 processors are used, several processors will be assigned to the same region.

LocusRoute generates an OutputFile describing the routes determined for the wires. At the end of the computation, a single process calculates the exact "height" of the circuit, in wires. This number is printed on the standard output device as the *Exact Density* Sum, and is the final measure of route quality.

## 10.5 Results

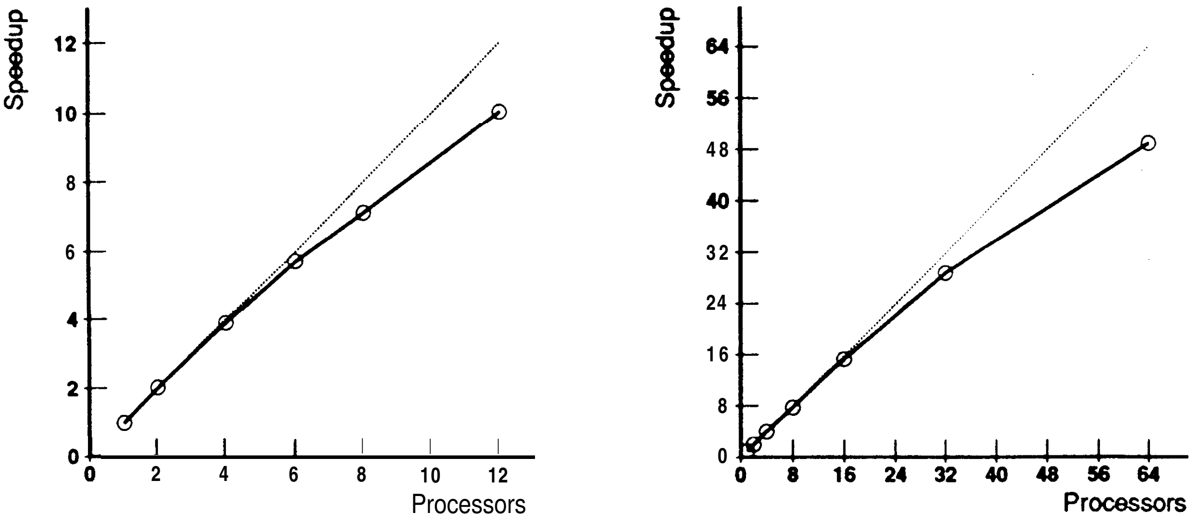
We present results for the Primary 1 grin circuit. The timer begins after all processes are spawned, and stops just before computing the final routing quality. Note that cold-start cache misses (especially for processes other than the original one that initializes the cost array) are not excluded from the timing measurement, since only about two iterations will be performed in a realistic run of the program. Only self-relative speedups are presented.

Speedups obtained on the Multimax are shown in the left graph of Figure 11. Measurements obtained with the simulator are shown in the right graph of the figure and in Table 10. Synchronization and load-balancing are minor limitations when the number of processors is much smaller than the number of wires; however, input-dependent load imbalances start to degrade performance as the number of processors increases. With a real memory system, the main limitation to speedup is the miss rate on accesses to shared memory. The bulk of the shared memory accesses are directed to the cost array during the route evaluation, wire lay-down, and wire rip-up phases. As the number of processors is increased, these accesses are more likely to interfere with one another, causing invalidations and subsequent cache misses.

# 11 PTHOR

PTHOR is a parallel, distributed time, event driven simulator. Its purpose is to verify the behavior of a digital logic circuit, given a description of the circuit and its input. The circuit is modeled as a collection of elements interconnected by wires or nodes. Conceptually, the elements are functional blocks with inputs, outputs, and internal state. The elements can be as simple as 2-input logic gates, such as AND-gates and OR-gates, or as complex as entire CPU's. The interconnecting nodes (wires) hold one of a small number of discrete values, such as high, low, undefined, and floating.

PTHOR uses a variant of the Chandy-Misra [21] distributed-time algorithm (denoted CM). The CM algorithm will be described only briefly here; for an in-depth treatment see [22]. While the standard event-



Encore Multimax

Simulator

Figure 11: LocusRoute Speedups: Multimax and Simulator.

Table 10: LocusRoute Information (Simulator).

| Number of Processors | Miss Rate (%) | Synchronization Waiting Time (%) |
|---|---|---|
| 1 | 0.20 | 0.00 |
| 2 | 0.70 | 0.00 |
| 4 | 1.42 | 0.00 |
| 8 | 2.12 | 0.00 |
| 16 | 3.17 | 0.00 |
| 32 | 4.47 | 0.01 |
| 64 | 5.08 | 0.02 |

driven algorithm maintains a common value of the current simulated time for the entire circuit, CM allows every element to advance its own value of time independently of other elements. As a result, different elements might have different notions of the current simulated time. An element receives time-stamped events on its inputs and, when it has events in each of its inputs, it computes its output, possibly sending out events on its outputs. In this implementation, an element is activated for evaluation whenever it receives a new input event. The activation places it on the task queue of a processor which will compute its new output behavior. The implementation of the algorithm allows the possibility of deadlocks, which are manifested by no processor having any work pending in its queue.[9] When deadlocks do occur, they are resolved and the element evaluations continue as before.

## 11.1 Principal Data Structures

The main data structures include the element and nodes structures, and the distributed task queues. Every node also has an event list, a list of time-stamped events oustanding on that node. Them is one task queue per processor. Since there is usually a large number of elements and nodes, the memory requirement of the application is essentially linear in the size of the circuit, measured in elements or wires.

Every node structure occupies 12 bytes. Every event also takes up 12 bytes, and the number of events

[9] Note that this has nothing to do with a deadlock in the physical circuit being simulated.

allocated depends on the circuit, input vectors, and length of the simulation. **For** the **relatively** small `risc` circuit used in our measurements, approximately **8,000** events are allocated **Every** element structure takes up **about** 110 bytes plus some element dependent storage (e.g. pointers to input node arrays). **For** a typical element in the `risc` circuit, the additional element-dependent storage averages about **180** bytes per element. Thus, the data set size for a simulation of this small circuit is about 1.6 Mbytes.

## 11.2 Structure of the Parallel Program

The program begins by reading in the circuit **netlist,** creating additional processes, initializing the task queues and node values, and reading in the stimulus file of input vectors. This is all done by a single process. All processes then repeat the following until the behavior of all wires is known **upto** the maximum simulation time:

- Evaluate elements until a simulation deadlock (all task queues empty).
- Perform deadlock resolution to activate more elements.

### 11.2.1 Partitioning/Scheduling and Locality

Partitioning in PTHOR is done in terms of elements, and scheduling through a task queue **per** processor. Locality is incorporated by assigning every element a preferred processor that it should be evaluated on. Whenever an element is activated, it is placed on its preferred task queue. Preferred queues are assigned to the elements in a round-robin fashion when the circuit is read in, to distribute the elements evenly among processors. No attempt is made to assign elements that are physically contiguous in the circuit, or that share wires, to the same processor. While this would enhance data **locality,** it might compromise load-balancing by assigning processors to distinct regions of the circuit Input events for an element's evaluation are determined by checking the event lists on all its inputs. When a processor's task queue is empty, it takes elements from another **processor's** queue (typically, an element is evaluated on its preferred processor about 95% of the time).

### 11.2.2 Synchronization and Granularity

**Every** element has a flag associated with it to ensure that it is not activated and placed on a queue more than once (owing to multiple changing inputs) at any given time. Locks are used to ensure the required mutually exclusive access to elements. There is little contention for these locks. Every work queue also has a single lock for insertions and deletions. Most of the time, these locks also have very little contention. **The** main synchronization points in the application occur at the deadlocks, when all processors must synchronize. These are the only points at which barriers are used in the program

**Granularity** in PTHOR is **measured** as the time taken to evaluate an element. It depends mostly on the nature of the elements in the circuit. **The** `risc` circuit is composed of only logic gates and one-bit registers, elements that don't have too many inputs and aren't very time-consuming to evaluate. The granularity of this circuit was found to average 580 simulator cycles and to not have a very large variance (about 80% of the element evaluations being within 50 cycles of the mean).

## 11.3 Profile of a Uniprocessor Execution

For a one-processor simulation of the `risc` circuit with the run-time parameters described in Section 11.5, the average length of the task queue is about 102 elements. That is, if each element executed in one unit of time, and there were no overhead associated with synchronization, the exploitable parallelism of the application would be 102. The average number of element evaluations between deadlocks is 260. **Thus** there are usually a couple of iterations of evaluating elements and activating more elements before a deadlock occurs. 2,000 elements are evaluated in the average simulated clock cycle. Of the total execution time, 7% is spent performing **deadlock** resolution.

## 11.4 Running the Program

The command line for running PTHOR looks like: `pthor -t 5000 -n <num-procs> -g 100 -i`. The first argument (-t) specifies the number of time ticks to simulate the circuit for. Next is **-n**, for the number of processors. The **-g** argument specifies half of the clock period. Finally, **-i** turns on incremental deadlock resolution. The arguments shown here are suggested ones for the `risc` circuit.

### 11.4.1 Input and Output

We provide two input circuits and input vector sets with this program. One is the small `risc` circuit described above, and the other (`dash`) is the directory controller for the Stanford DASH Multiprocessor. The `risc` circuit has 5,060 elements, while `dash` has 24,611 elements. Since these circuits are actually compiled into the executable, the file `risc. c` or `dash. c` (as desired) must be copied into `csim. c` before compilation.

The necessary inputs to **PTHOR** are:

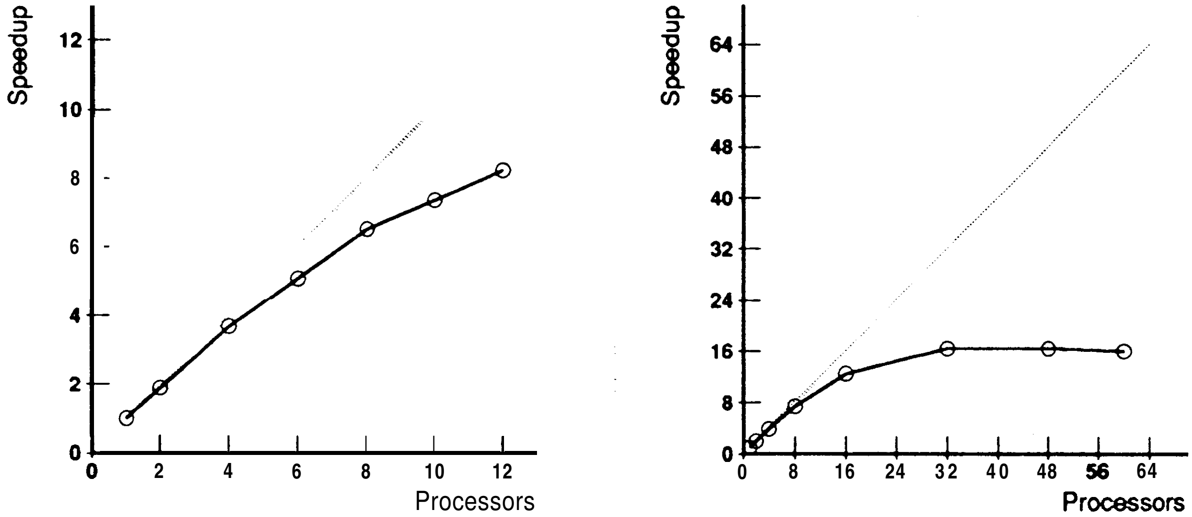
- The circuit **netlist**: specifies the element types and interconnection of the elements of the circuit, and is contained in the file `csim.min`.
- The input stimulus: supplies external signals to the circuit such as clock and reset, and is contained in the file `stim`

Note that `risc.min` and `risc.stim`, or `dash.min` and `dash.stim` (as desired), must be copied into the files `csim.simand stim` beforeexecution.

The program prints various statistics on the standard output device. The the total execution time excluding initialization is contained **at** the end of the output in the line `xtot=time`.

## 11.5 Results

We use the `risc` circuit in our measurements, although the larger `dash` circuit is expected to provide more parallelism. The circuit is simulated for 5000 time ticks, with 200 ticks comprising a clock cycle in the circuit. The timer is started when the first set of evaluations begins (after the stimulus file is read), and stopped at the end of the program. Only self-relative speedups are reported.



Encore Multimax

Simulator

Figure 12: PTHOR Speedups: **Multimax** and Simulator.

Table 11: PTHOR Information (Simulator).

| Number of Processors | Miss Rate (%) | Synchronization Waiting Time (%) | Avg. Queue Length |
|---:|---:|---:|---:|
| 1 | 0.33 | 0.45 | 102.6 |
| 2 | 2.20 | 1.08 | 55.0 |
| 4 | 3.29 | 1.60 | 21.1 |
| 8 | 4.12 | 1.64 | 13.2 |
| 16 | 5.70 | 1.56 | 5.0 |
| 32 | 7.79 | 1.63 | 2.9 |
| 60 | 8.76 | 2.42 | 2.5 |

Speedups obtained on the Multimax are shown in the left graph of Figure 12. Measurements obtained with the simulator are shown in the right graph of Figure 12 and in Table 11. The speedups observed are limited by the number of elements available for concurrent execution. With 60 processors there are, on average, only 2.5 elements for each processor to execute between deadlocks (see average queue length in the table). This number diminishes inversely with the number of processors. Since an element evaluation is very simple, 2.5 evaluations is too small a unit of work compared to the overhead of a barrier synchronization. Larger circuits might provide a little more parallelism, but in all realistic circuits parallel activity is likely to be limited. The extra work done by the processors, as measured by extra element evaluations due to parallelism, is not an important speed-limiting factor.

# 12 Cholesky

This program performs parallel Cholesky factorization of a sparse positive definite matrix. That is, given a positive definite matrix $A$, *the program* finds a lower triangular matrix $L$, such that $A = LL^T$. *Sparse systems* involving positive definite matrices arise quite frequently in a number of domains, including structural analysis, device and process simulation, and electric power network problems. Such systems are typically solved by performing a Cholesky factorization of the matrix, and the factorization is often the bottleneck in the overall computation.

This program is not a general purpose parallel sparse Cholesky factorization package. Instead, it concentrates on the most time-consuming components of the factorization. In general, Cholesky factorization typically proceeds in three steps: ordering, symbolic factorization, and numerical factorization. The ordering step symmetrically reorders the rows and columns of $A$ *to* reduce the amount of fill in the factor $L$. *Our* program does no reordering. A matrix must be reordered, if necessary, before being passed to the program. The second step, symbolic factorization, determines the non-zero structure of the factor matrix. This step typically accounts for a small fraction of the overall factorization runtime. While our program does perform this step, it is done on a single processor and is not measured The third step, numerical factorization, determines the actual numerical values of the non-zero entries in $L$. This step is typically the most time-consuming, and is parallelized in our program.

The numerical factorization approach we use is very efficient, due to the use of supemodal elimination techniques. The approach is a dynamic version of the *supernodal fan-out method* [26], an enhancement of the fan-out method of [24]. Supemodes are sets of columns with nearly identical non-zero structures, and a factor matrix will typically contain a number of often very large supernodes.

## 12.1 Principal Data Structures

The primary data structure in this program is the representation of the sparse matrix itself. A matrix is stored by columns, using a data structure almost identical to the one used in SPARSPAK. In C, the data structure is:

```
typedef struct {
  int n, m;
  int firstnz[], startrow[], row[];
  double nz[];
} SMatrix;
```

Fields `n` and m give the number of columns and non-zeroes in the matrix, respectively. The `firstnz[]` field holds, for each column, a pointer to the first non-zero element of the column. These pointers are references into the `nz` array, where the non-zeroes are stored. All non-zeroes belonging to a particular column are stored contiguously within this array. Since the matrix is sparse, the data structure must keep track of the rows in which the non-zeroes reside. The row number of a particular non-zero is available through the `row[]` and `startrow[]` fields. Row numbers are stored in a compressed manner in order to conserve space. Details of the compression and other issues relating to the data structure can be found in [25]. Of the two matrices we use, BCSSTK14 occupies 420 Kbytes unfactored and 1.4 Mbytes factored, while the corresponding numbers for BCSSTK15 are 800 Kbytes and 7.7 Mbytes, respectively.

## 12.2 Structure of the Parallel Program

The primary operation in column-oriented sparse Cholesky factorization is the addition of a multiple of one column of *A* into another in order to cancel a non-zero in the upper triangle. This operation is typically referred to as a column modification. If the grain size for the parallel computation were chosen to be a single column modification, the overheads associated with task creation and distribution wculd be too large. In supernodal Cholesky factorization, the column modification operation is replaced by a supemodal modification operation, where a column is modified by all the columns of a supemode at once. Task overheads would still be too large even if a single supemodal modification were chosen as the grain size. The task grain we choose is the set of all supemodal modifications performed by a particular supernode.

### 12.2.1 Partitioning and Scheduling

The parallel sparse factorization computation proceeds as follows. With each supemode, a count is kept of how many modifications have yet to be done to columns in that supernode. We call this the ***incoming*** count. A shared global task queue holds all supernodes whose incoming counts have gone to zero. These supernodes have received all modifications that will be done to them, and are therefore ready to perform modifications themselves. A free processor pulls a supemode task off the task queue and performs all supemode modifications done by that supemode. In the course of performing these supernodal modification, the incoming counts of the destination columns' supernodes are decremented to reflect the modifications done to them. If a supemode's count goes to zero, it is placed on the global task queue.

### 12.2.2 Synchronization

The only interactions between processors occur when they attempt to dequeue tasks from the global task queue and when they attempt to perform a number of simultaneous supemodal modifications to the same destination column. Both of these cases are handled with locks.

## 12.3 Profile of a Uniprocessor Execution

Almost all the sequential runtime is spent performing supemodal modifications.

## 12.4 Running the Program

The program is run by specifying the number of processors to be used and the file containing the matrix to be factored. The command `cholesky -p4 bcsstk14` would factor the matrix BCSSTK14 using 4 processors. The only other option is the `-o` command line option, which causes the program to output the factor matrix.

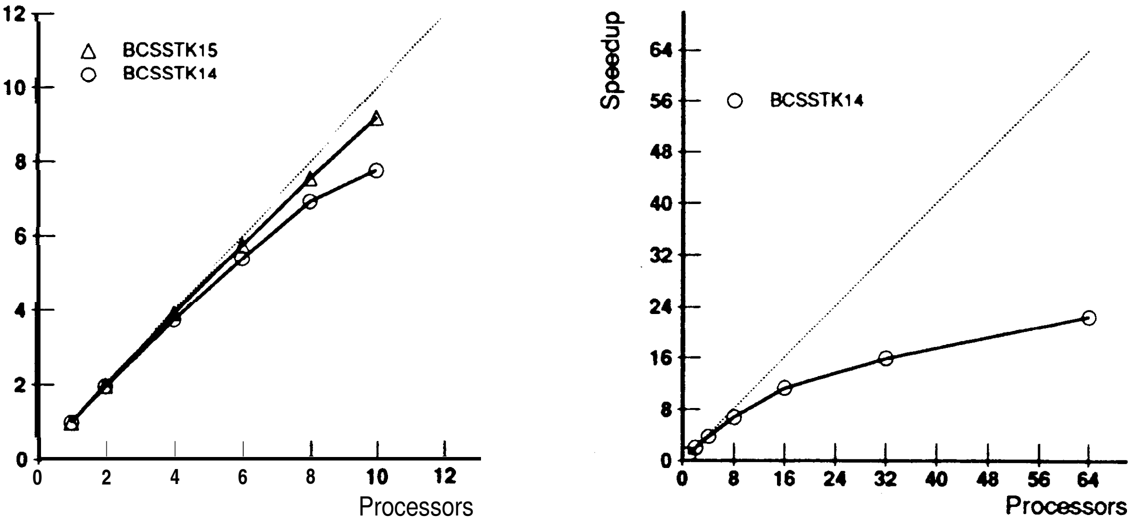
### 12.4.1 Input and Output

The program reads only the non-zero structure of the *A* matrix from the input file, choosing its own non-zero values. The program also verifies that the computed factor is correct once the factorization is complete. Two input matrices are provided. Both come from the Boeing/Harwell sparse matrix test set [23]. bcsttk14 is a 1806-by-1806 matrix with 30,824 non-zeros in the matrix and 110,461 in the factor, it has 503 distinct supemodes, the largest of which contains 135 columns. The corresponding numbers for the larger matrix bcsttk15 are 3948-by-3948, 56934, 647274, 1295 and 211, respectively. Both matrices have been reordered using the minimum degree heuristic [25].

The program prints some numbers describing the matrix and the execution. It also outputs the execution time and MFLOPS rate of the factorization.

## 12.5 Results

The results we present are for the factorization of the two Boeing/Harwell matrices included with the program. Only the numerical factorization phase is measured. The main determinant of parallel performance appears to be the number of floating point operations performed. This number depends on both the size of the input matrix and its sparsity pattern.



Encore Multimax

Simulator

Figure 13: Cholesky Speedups: Multimax and Simulator.

Normalized speedups obtained on the Multimax are shown in the left graph of Figure 13. Note that the larger matrix (BCSSTK15) indeed yields better speedups in this case. Speedups with the simulator are presented in the right graph of Figure 13. No results are given for matrix BCSSTK15 because of the enormous amount of time required for its simulation. The simulated speedups for the smaller matrix are quite similar to the speedups observed on the Encore within the range of 1 to 8 processors.

Miss rates and synchronization waiting times observed on the simulator for the BCSSTK14 matrix are shown in Table 12. Miss rates increase steadily as more processors are added. The bulk of misses is due to invalidations. The reason the speedups are poor for large numbers of processors is clear form the synchronization waiting times. BCSSTK14 is a relatively small matrix with limited available concurrency. This leads to load imbalances, high synchronization waiting times and low speedups. We would expect much larger speedups when factoring larger matrices.

Table 12: Cholesky Information (Simulator).

| Number of Processors | Miss Rate (%) | Synchronization Waiting Time (%) |
|---|---|---|
| 1 | 1.28 | 0.06 |
| 2 | 3.55 | 1.45 |
| 4 | 4.96 | 3.63 |
| 8 | 6.56 | 10.19 |
| 16 | 7.80 | 22.19 |
| 32 | 9.14 | 42.01 |
| 64 | 10.91 | 52.38 |

## 13 Concluding Remarks

The Stanford Parallel Applications for Shared-Memory are a set of real applications for use in the design and evaluation of parallel systems for shared-memory multiprocessors. They were originally targeted at bus-based multiprocessors and exploit parallelism at the medium to large granularity. As long as inherent limitations of these applications and their interaction with various architectures are kept in mind, they can be very useful in providing a consistent, realistic suite for evaluation studies. We look forward to expanding the set with other parallel applications from the user community.

## 14 Acknowledgements

We would like to thank the following people for cleaning up the parallel programs, running them, and writing initial versions of the individual application reports: Steve Goldschmidt (MP3D), Margaret Martonosi (Locus-Route), Ed Rothberg (Chole-sky) and Larry Soule (PTHOR). Okokon Okon helped with runs for the Ocean code.

## References

[1] J J. Dongarra, J.L. Martin and J. Worlton, "Evaluating Computers and Their Performance: Perspectives, Pitfalls, and Paths," IBM Research Report 12904, April, 1987.

[2] "SPEC Benchmark Suite Release 1.0," October, 1989.

[3] E.L. Lusk and R.A. Overbeek, "Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-scheduling DO-Loop, and Askfor Monitors," Tech. Report No. ANL-84-51, Rev. 1, Argonne National Laboratory, June 1987.

[4] "Using the Encore Multimax," Tech. Mem. No. 65, Rev. 1, Math. and Comp. Sci. Division, Argonne National Laboratory, Feb. 1987.

[5] JJ. Dongarra, J. Bunch, C. Moler and G. Stewart, "LINPACK Users' Guide," SIAM Pub., Philadelphia, 1976.

[6] H. Davis, S. Goldschmidt and J.L. Hennessy, 'Tango: a Multiprocessor Simulation and Tracing System," Tech. Report No. CSL-TR-90-439, Stanford University, 1990.

[7] J.P. Singh and J.L. Hennessy, "Parallelizing the Simulation of Ocean Eddy Currents," to appear in ***Journal of Parallel and Distributed Computing.*** Also Tech. Report No. CSL-TR-89-388, Stanford University, Aug. 1989.

[8] G.H. Golub and CF. Van Loan, *Matrix Computations,* Second Edition, Chap. 10, The Johns Hopkins University Press, 1989.

[9] C.W.Gear, *Numerical Initial Value Problems in Ordinary Differential Equations,* Prentice-Hall, New Jersey, 1971.

[10] L.F. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems,* MIT Press, Cambridge, 1988.

[ 11] J.P. Singh and JL. Hennessy, "Automatic and Explicit Parallelization of an N-body Simulation," submitted for publication.

[ 12] G.C. Lie and E.Clementi, "Molecular-Dynamics Simulation of Liquid Water with an *ab initio* Flexible Water-Water Interaction Potential," *Physical Review,* Vol. A33, pp. 2679 ff., 1986.

[ 13] 0. Matsuoka, E.Clementi and M. Yoshimine, "CI Study of the Water Dimer Potential Surface," *Journal of Chemical Physics,* Vol. 64, No. 4, pp. 135 l-61, Feb. 1976.

[14] R. Bartlett, I. Shavitt and G. Purvis, "The Quartic Force Field of $H_2O$ Determined by Many-Body Methods that Include Quadruple Excitation Effects," *Journal ofChemical Physics,* Vol. 7 1, No. 1, pp. 281-29 1, July *1979.*

[15] M. Berry et. al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," CSRD Report No. 827, Center for Supercomputing Research and Develpment, Urbana, Illinois, May 1989.

[16] David R. Cheriton, Hendrik A. Goosen, and Philip Machanick, "Restructuring a parallel simulation to improve cache behavior in a shared-memory multiprocessor: A first experience, 1990," to appear in Proc. International Symposium on Shared-Memory Multiprocessing, April 1991.

[17] Jeffrey D. McDonald, " A direct particle simulation method for hypersonic rarified flow," CS 411 - Final Project Report, Stanford University, March 1988.

[18] J.S. Rose, "LocusRoute: a parallel global router for standard cells," Proc. 25th Design Automation Conference, pages 189-195, June 1988.

[19] J.S. Rose, 'The parallel decomposition and implementation of an integrated circuit global router," ACM Sigplan Symposium on Parallel Programming: Experience with Applications, Languages and Systems, pages 138-145, July 1988. Sep. 1990.

[20] J.S. Rose, "Parallel global routing for standard cells", *IEEE Trans. Compurer-Aided Design of Circuits and* Systems, September 1990.

[21] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *Comm of the ACM, 24:* 11, pages 198-206, April 1981.

[22] Larry Soule and Anoop Gupta. "Analysis of parallelism and deadlocks in distributed-time logic simulation," Technical Report CSL-TR-89-378, Stanford University, March 1989.

[23] I. Duff, R. Grimes, and J. Lewis, "Sparse matrix test problems," *ACM Transactions on Mathematical Software,* 15:1-14, 1989.

[24] A. George, M. Heath, J. Liu, and E. Ng, "Solution of sparse positive definite systems on a hypercube," Technical Report TM-10865, Oak Ridge National Laboratory, 1988.

[25] A. George and J. Liu, *Compurer Solution of Large Sparse Positive Definite Systems,* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[26] E. Rothberg and A. Gupta, "Techniques for improving the performance of sparse factorization on multiprocessor workstations," Proceedings of Supercomputing '90, November, 1990.