SPLASH: STANFORD PARALLEL APPLICATIONS FOR SHARED-MEMORY

Jaswinder Pal Singh
Wolf-Dietrich Weber
Anoop Gupta

Technical Report No. CSL-TR-92-526

June 1992

This research has been supported by DARPA contract N00039-91-C-0138.

SPLASH: STANFORD PARALLEL APPLICATIONS FOR SHARED-MEMORY*

Jaswinder Pal Singh, Wolf-Dietrich Weber, Anoop Gupta

Technical Report: CSL-TR-92-526

June 1992

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, discuss and compare some of their important characteristics-such as data locality, granularity, synchronization, etc.-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.

Kev Words and Phrases: parallel application suite, shared memory, documentation, application characteristics, evaluation.

*This report replaces and updates CSL-TR-9 1-469, April 199 1.

Copyright © 1992

by

Jaswinder Pal Singh, Wolf-Dietrich Weber, Anoop Gupta

SPLASH: Stanford Parallel Applications for Shared-Memory

Jaswinder Pal Singh, Wolf-Dietrich Weber and Anoop Gupta

Computer Systems Laboratory

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, discuss some of their important characteristics— such as data locality, granularity, synchronization, etc.-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. [1] 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 #at 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 sixes. 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 7-13 presents each application in detail. Finally, Section 14 offers some concluding remarks.

2 Distribution

Application sources and makefiles for the Encore Multimax are obtainable by anonymous ftp from the intemet 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. A version of the paper can be obtained from Stanford University as Technical Report No. CSLTR-91-469. The most up to date version will always be maintained in the ftp directory on `mojave.Stanford.EDU`. Questionsshould `beaddressed` to **`splash@mojave.Stanford.EDU`**, and people who want to be added to or deleted from the mailing list for updates should send electronic mail to that account

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

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

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.

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
Barnes-Hut	E.Bruni, D.Roger, J.P.Singh	C	2750	simulate evolution of galaxies
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. Five of our applications (Ocean, Water, Barnes-Hut, MP3D and Cholesky) are scientific computations and two (LocusRoute and PTHOR) are from the area of computer-aided design. We plan to enhance the coverage of the programs in the future.

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	short-range N-body	direct, with cutoff radius
Barnes-Hut	astrophysics	hierarchical N-body	Barnes-Hut
MP3D	aeronautics	particle-in-cell methods	Monte Carlo
LocusRoute	VLSI CAD	standard cell routing	iterativerefinement
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 7-13. 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). In others, the work done is much more dependent on the particular input data set and may change with time, so they require dynamic partitioning and scheduling. In the Barnes-Hut application, dynamic partitioning is explicitly implemented by the programmer in a separate partitioning phase; in LocusRoute and PTHOR, tasks are scheduled using dynamic task queues.
- **Synchronization:**
  - What kinds of synchronization are used to preserve dependences in the parallel program? The scientific applications (Ocean, Water, Barnes-Hut, MP3D) mostly use global barrier synchronization, while those that use task queues (LocusRoute, PTHOR, Cholesky) preserve dependences either through the order in which tasks are enqueued as well or sometimes with barriers. All the programs use locks to provide mutual exclusion. Some also use regular variables as flags for event-driven synchronization.
  - Does the synchronization relax the ordering of events from that used in the sequential program? Or is it too conservative, in which case what is the synchronization we would really like to express? In most of the scientific programs that use barriers, this is done primarily for programming convenience. The actual dependence structure of the programs does not call for global synchronization, but only finer-grained synchronization between small subsets of processors. To the extent that this sort of synchronization does not incur too large an overhead, it can be used to enhance performance by overlapping computations that are currently separated by barriers. This may be useful when different partitions in a phase of parallel computation take different amounts of time to execute, either due to computational load imbalances or due to differences in their interactions with the memory system. However, in some cases the synchronization needed is more global in nature, and the only fine-grained synchronization that could replace barriers is at the level of individual data elements.
  - How does the amount of synchronization vary with the size of the problem or machine?
- **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 with a single number is a somewhat less meaningful in the scientific programs (Ocean, Water, Barnes-Hut, MP3D). Here, several large computations are performed, each on every particle or point in the problem domain, with global synchronizations between some of these computations. The granularity therefore grows with the number of particles/points (as well as with other application parameters such as the accuracy of the numerical methods used) and is inversely proportional to the number of processors used However, we find that the granularity usually decreases under realistic models of scaling the problems and machines. Another type of granularity relevant to all the applications is the size of critical sections protected by locks. Since these critical sections usually involve updating shared scalar variables, this granularity is small in all our programs.
- **Parallel Performance:** Does the parallel performance of the application scale well with increasing numbers of processors for the fixed input size we use in each case? We assume an ideal memory system (PRAM), thus characterizing only algorithmic or computational speedup under a constant problem size. Real memory systems might reduce the speedups significantly.
- **Locality of Data Referencing:**
  - How regular and predictable are the access patterns to various data structures as the application executes? Is there a significant tradeoff between data locality-both within and across phases of

parallel computation—and load balancing? The scientific applications that operate on uniform problem domains (Ocean, Water) have very regular, predictable access patterns and make it very easy to statically partition for both load balancing and data locality. Nonuniform applications (such as Barnes-Hut and MP3D) do not afford good performance through static partitioning.

- What ***kinds*** of locality does ***the application*** afford, and which of ***these*** does ***the program*** exploit? Is the locality obtained mostly by temporal reuse of cached data (Water, Barnes-Hut), or is it also possible to obtain effective locality in main memory by data distribution (Ocean) or in the network topolgy by processor assignment? The machine model for which the programs were written is bus-based with a centralized shared memory, so that the only locality to exploit is in the per-processor caches. However, in some cases we discuss the ease and performance potential of exploiting locality in physically distributed main memory and in an interconnection network as well. Data distribution is easy in some applications (such as Ocean) in which locality in the data structure layout corresponds directly to the physical locality in the problem domain that most of the applications exploit. It is also very effective when most of a processor's references that are likely to miss in the cache are to data in its own partition (such as in Ocean again). In other applications (such as Barnes-Hut) data distribution is more difficult and not very worthwhile. Network locality can be exploited quite effectively in most of the scientific applications.
- How much communication is there relative to computation, and how does it grow with application parameters and the number of processors? Typically, we find that under the most realistic scaling models, the communication to computation ratio grows as larger problems are run on larger machines.
- Is there enough spatial locality in the data structures so that large cache lines can be used to enhance performance? This is true in applications such as Ocean, in which the layout of the data structures corresponds exactly to the physical locality in the computational domain. In many applications, the utility of long cache lines can be enhanced by organizing data structures to match the access patterns of the program (for example, keeping data that represent the same variable for different units of parallelism-molecules or grid locations, say-contiguous in memory, rather than data that represent different variables for the same unit of parallelism).
- What are the patterns of data communication in the application? For example, communication patterns are phase-structured in many scientific applications (i.e. data that are read-shared in one phase of computation are written only in another phase), a fact that can be exploited by cache coherence protocols and techniques to hide memory latency such as data prefetching.

- **Scaling to Larger Problems and Machines:** For some scientific applications, we also discuss issues related to to scaling the problems realistically to run on larger machines, and the impact this has on behavioral characteristics such as the communication to computation ratio and the granularity of computation between synchronization points. In general, we find that scaling under a constraint on execution time is most realistic for these applications, and that under realistic forms of time-constrained scaling (in which other application parameters are scaled along with the input data set size [4, 21]) the communication to computation ratio grows slowly and the granularity between synchronization points decreases as larger problems are run on larger machines. Also, the input data set size grows less than proportionally to the number of 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 the individual treatments of the applications in Sections 7-13.

The second column of Table 3 reveals a range of scheduling mechanisms, from fully static to 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 preferentially assigned to the same processor every time. The third column describes the dominant forms of synchronization used to preserve dependences in the program. Recall 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 7-13. A wide range of granularity is present, although none of the programs are extremely fine-grained. For the scientific programs whose granularity depends on input data size, it is listed in the table as being large. The computational speedups for a fixed problem size 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 an appropriate input data set. PTHOR speedups are also dependent on the input data set. However, realistic input circuits for this application generally provide limited parallelism.

Table 3: Summary of some behavioral characteristics.

Application	Scheduling	Synchronization	Granularity	Comp. Speedups
Ocean	static	barrier	large	good
Water	static	barrier	large	good
Barnes-Hut	dynamic: programmer	barrier, flag	large	good
MP3D	static	barrier	large	good
LocusRoute	semi-static	task queues, barrier	medium	good
PTHOR	semi-static	task queues, barrier	medium	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. There appear to be three types of data that might be termed shared: those that are declared in the program to be shared, those that may be shared in some run of the program with some input data set, and those that are actually referenced by more than one processor in a given run. The counting of shared references in Table 4 refers to the first category, which is perhaps the least useful to a hardware designer. We hope to rectify this situation by providing data for actual sharing as well. Note that the programs issue a substantial fraction of their references to private data as well, implying that these must be included in simulation studies if proper measurements of application or system behavior are to be made. This is particularly true when direct-mapped data caches are used, since mapping collisions with private data might affect the miss rates to shared data

Table 4: Reference statistics by application.

Application	Number of Processors	Input Data Set	Cycles (M)	Reads (M)	% Shared	Writes (M)	% Shared
Ocean	32	98-by-98 grid	2.46	11.0	88	3.17	68
Water	32	LW112	2.16	17.1	21	6.93	7.5
Barnes-Hut	32	8k bodies	33.6	204.2	25.5	118.1	1.3
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 [5]. 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 [7] 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 1.3 1) and `f 77` (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 or network architecture. 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 cold-start misses and those caused by invalidations due to interprocessor communication. Cache lines are 4 bytes long, to avoid false sharing problems. For some applications, we eliminate cold misses as well by gathering statistics only after the initialization phase is complete. 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 wait 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. If successive time-steps are nearly identical in application behavior, running for a small number of time-steps does not introduce much error into our measurements. For every application that we restrict to a small number of time-steps, we point out effects that this restriction potentially ignores. 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 provide details about the parallel program, including principal data structures, program structure and profiling information. Application characteristics along the lines of Section 4.2 are then detailed We also provide information about how the program is run, what inputs it requires and what output it produces. Finally, we present results obtained from running the application on the Encore Multimax and on the simulator, and discuss these results briefly 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 friction with the ocean walls and floor is included. 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.

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 [8]. The continuous functions are transformed into discrete counterparts by second-order finite-differencing, and the resulting difference equations set up and solved on two-dimensional tied-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 algorithm to solve the elliptic equations (see [8]); the parallel programs use an iterative method: Gauss-Seidel with Successive Over Relaxation (SOR) [8, 9]. This iterative solver works well for coarse grid resolutions such as the one we use. For finer resolutions, a more sophisticated solver may be necessary. We have developed a multigrid solver for this purpose. A new version of the Ocean application, with the SOR solver replaced by a multigrid one, will soon be added to the suite,

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 declared 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 external forces and solving a single elliptic equation. After this, the outermost loop of the program iterates over a fixed number of time-steps, each performing a number of computations on entire grids. Parallelism is afforded at a hierarchy of levels, both across and within grid computations (see [8]). 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 Ψ grids in Figure 1 represent the streamfunction values being solved for. Ψ_1 is the value at the middle of the upper layer of the ocean, and Ψ_3 at the middle of the lower layer. Ψ_{UM} and Ψ_{LM} are streamfunction running sums, while Ψ_a, Ψ_b and Φ are mathematical functions used in the solution process. The γ 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 (g-point and 5-point stencils, respectively) with different input and output arrays, while the equation solution is an m-place near-neighbor iteration to convergence with a 5-point stencil. An understanding of the equation system and solution method can be obtained from [8].

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

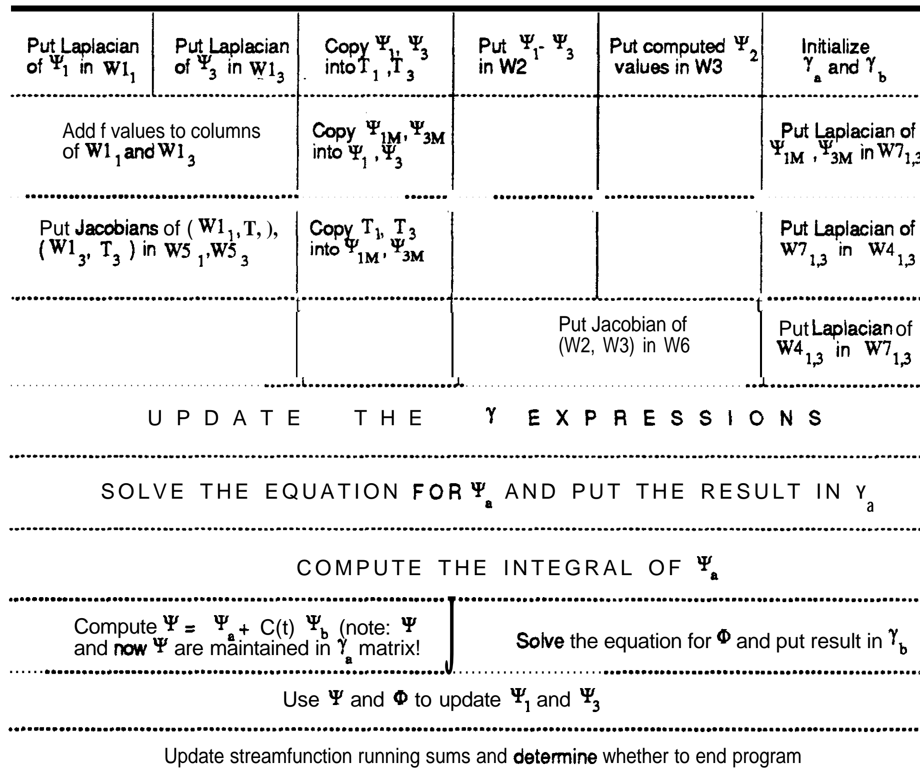


Figure 1: Parallel structure of Ocean.

7.2.1 Partitioning/Scheduling and Locality

The program proceeds from left to right within every horizontal phase of Figure 1, with all processors collaborating on every grid task. The referencing patterns of all tasks are regular and input-independent, which allows static scheduling for data locality both within and across grid tasks without the cost in load balancing that this might exact in less regular applications. A processor's data access patterns don't change across time-steps either, the only difference is that caused by variability in the number of SOR iterations to convergence.

A domain-decomposition based partitioning and scheduling method is both natural to the physical problem and well-suited to exploiting data locality. The non-boundary elements of the discretized ocean cross-section are split into equal-sized subdomains (chunks of adjacent columns in this case[3]). Each subdomain is permanently associated with a processor, and is assigned to that processor in every grid computation. The localized nature of the computations ensures that a task will access only grid points within its subdomain or their immediate neighbors, in all grids that it uses. Having every subdomain be contiguous yields intra-task locality in the near-neighbor computations. In fact, other than the few synchronizations and global accumulations every time-step, the only actual data sharing is at interpartition boundaries, and is among only a very small number (between two and four) of processors. If the grid size is N-by-N and the number of processors p, this communication grows as $O(N/p)$ in the grid dimension as opposed to computation which grows as $O(N^2/p)$. Communication can therefore be arbitrarily reduced relative to computation by increasing the number of grid points. A processor writes only grid points in its own subdomain, but may read boundary points from adjacent subdomains in near-neighbor tasks. In most cases, these boundary points are only read-shared in a given task, having been written by their own processors in a previous phase of computation. This sort of phase-structured communication (data being read-shared in one phase of computation in which they are not modified, and modified

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

in a different phase separated from the first by synchronization) is very common in scientific computation, and can be exploited by cache coherence protocols and latency hiding techniques such as prefetching. The one exception in this application is the SOR iterations, in which simultaneous read-write sharing and data races are permitted. Even in the latter case, (if partitions are large enough), the dynamic sharing patterns are conducive to effective data prefetching. In fact, as long as the grid data structures are traversed in the appropriate way (column-major in FORTRAN), the prefetching advantages of long cache lines overwhelm the disadvantages of false-sharing and fragmentation in this application, and provide substantial performance benefits [11]. This is primarily because the organization of data structures (arrays each representing the values of a variable over all grid points, rather than an array of records, each representing all the variables for a grid point) matches the access patterns of the program.

About thirty grid tasks are performed every time-step, each of these accessing a small number of the roughly twenty-five grids in the program. Every grid task sweeps through the entire ocean cross-section. If the caches in a machine are large enough relative to the data associated with a subdomain, they might yield data reuse across tasks that access the same grid. In this case, tasks can be temporally ordered (within dependence constraints) to enhance such reuse. If the caches are too small to fit even a couple of subgrids, the near-neighbor computations can be internally blocked for a small constant factor improvement in the number of cache misses. A limited amount of blocking is obtainable across tasks. Other than these and prefetching, the intra-task near-neighbor reuse is the only benefit that small caches provide in this application. Note that the same domain decomposition approach taken for cache locality here can be used to allocate data appropriately on a machine with physically distributed memory modules, just as would be done on a message-passing machine. This is easy in this case since spatial locality in the layout of the array data structures corresponds exactly to the locality in the physical domain that the application exploits [4]. Such data distribution can have a substantial impact on performance in this application if the machine's caches cannot hold their partitions of the entire data set, owing to either limited size or cache mapping collisions among the many distinct grids (which can substantially degrade performance unless cam is taken to minimize such collisions by adjusting the layout of grids relative to one another [11]). Since grid points within a processor's partition are never written by any other processor, misses to these points (which constitute most of the capacity misses) can always be satisfied locally. The only misses that have to go across the network are those at partition borders, to neighboring points in another processor's partition. These misses are usually due to true communication rather than capacity, an even their impact can also be reduced by prefetching. Finally, the near-neighbor communication patterns also allow convenient exploitation of geographic locality in an interconnection network with nonuniform distances between processors (for example, a mesh or hypercube). A more detailed discussion of data locality and memory system performance in this application can be found in [11].

7.2.2 Synchronization and Granularity

Mutual exclusion, enforced with locks, is required in obtaining a process identifier and in two other situations in this application: when every process accumulates its private sum into a shared sum in computing a matrix integral, and when processors communicate through a shared convergence flag in the iterative solver. The cost of the required locking is negligible. Synchronization is also needed to preserve dependences across grid computations, which is accomplished by inserting ***barriers*** at the end of every phase in Figure 1. Barrier synchronization is overly restrictive here: All that is required is the less expensive point-to-point synchronization between processors that handle adjacent subdomains. 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; that is, there is a barrier between successive SOR iterations. We should mention here that our parallel equation solver does not follow the sequential SOR ordering exactly within an iteration, violating it at inter-partition boundaries. The result is that runs with different numbers of processors may cause the same equation to take different numbers of iterations to converge. This effect, however, only becomes significant when each partition is very small.

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 typically be large, the fraction of references that access synchronization variables is very small and we call the program large-grained. Even

[4] If partitioning is done in square subgrids, data distribution in units of pages might not be quite so easy (see [11]).

when barriers are used, the cost of each is at worst proportional to the number of processors, not to the size of the grid The number of barriers per time-step scales with the problem size only to the extent that the number of iterations to convergence does. Just like data communication, then, the relative amount of synchronization can be arbitrarily reduced by increasing the number of grid points. 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. The uniprocessor execution times of all the grid tasks other than the elliptic equation solver scale as $O(N^2)$ in the grid dimension Each iteration of the solver scales in the same way; however, the number of iterations depends on the grid resolution and the tolerance used to determine convergence.

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-2)`—the matrix dimension excluding boundary elements-is a multiple of the number of processors used. The physical resolution of the grid in meters is specified in a variable called `RES` in a `BLOCK DATA` statement in the program.
- the tolerance for convergence of the iterative equation solver.
- the SOR parameter ω for the equation solver (suggested value for the current grid resolution = 1.15).

The program supplied 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 2 days (6 time-steps), this number being specified in a `BLOCK DATA` statement.

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. A tolerance of 10^{-7} is used in the SOR solver.

Normalized and self-relative speedups obtained on the Multimax are shown in the left graph of Figure 2. 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.

Measurements obtained with the simulator are shown in the right graph of Figure 2 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

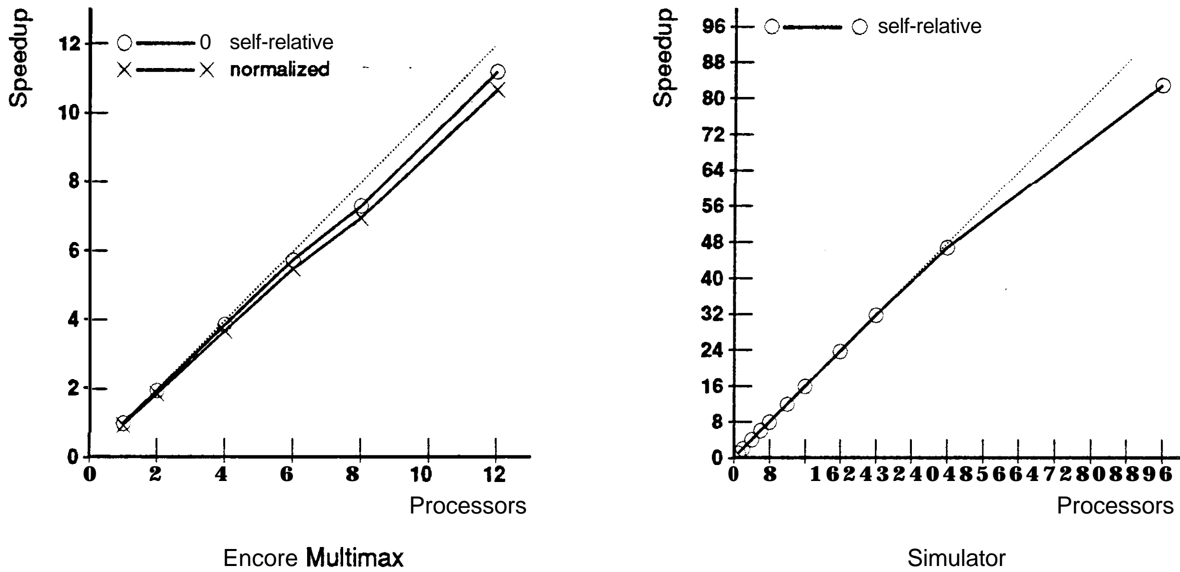


Figure 2: Ocean Speedups: Multimax and Simulator.

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 (with infinite caches) shown in the table arc 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 with infinite caches can be reduced by increasing the problem size or by using square sub-blocks-rather than column strips-as partitions. 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). 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.

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	12.35	3.74

7.6 Scaling to Larger Problems

Other than the amount of memory available, there are several limitations on the number of grid points that might be used on more powerful machines. Real oceans are of finite size, and the model used in this application is valid only for mesoscale (one to a few hundred kilometers) resolutions. As the number of grid points is increased and the spatial resolution therefore made finer, the solver takes more iterations to converge, particularly since

the error tolerance is made finer as well[5]. The error introduced by the finite differencing scheme is of the order of the square of the resolution. The error tolerance in the solver should therefore also scale with the square of the spatial resolution. Finally, as the error due to spatial discretization is reduced, the error due to temporal discretization should be reduced proportionally as well. Thus, the number of time-steps grows as the reciprocal of the square of the resolution. What this all means is that a k times more powerful machine cannot solve a problem with k times as many grid points in the same amount of time. The communication to computation ratio, or the miss rate with infinite caches, therefore grows under such time-constrained scaling. Also, since the number of time-steps simulated in a fixed execution time increases, the granularity of computation between synchronization points decreases under time-constrained scaling, thus increasing synchronization overhead and enhancing the performance potential of replacing barriers with point-to-point or finer-grained synchronization.

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 [10]. 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. The box length is computed by the program to be large enough to hold all the molecules. Double-precision accuracy is required for this simulation, 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 [12, 13], and details of the physical models in [14, 15, 16].

The sequential program, written in FORTRAN, is one of the Perfect Club set of supercomputing benchmarks [17]. 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 multiprocessor 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 that locate the molecule's center of mass. The data memory requirement of the program is 600 bytes per molecule, plus some per-process private storage. For our problem size of 288 molecules, the shared data set size of the application is about 200 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 3 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. As in the Ocean application, parallelism is available both across and within these tasks. Unlike Ocean, however, a

[5] **In fact, the SOR solver should be replaced by a multigrid solver for much finer spatial resolutions between grid points, and we shall provide a version of the application with a multigrid solver soon.**

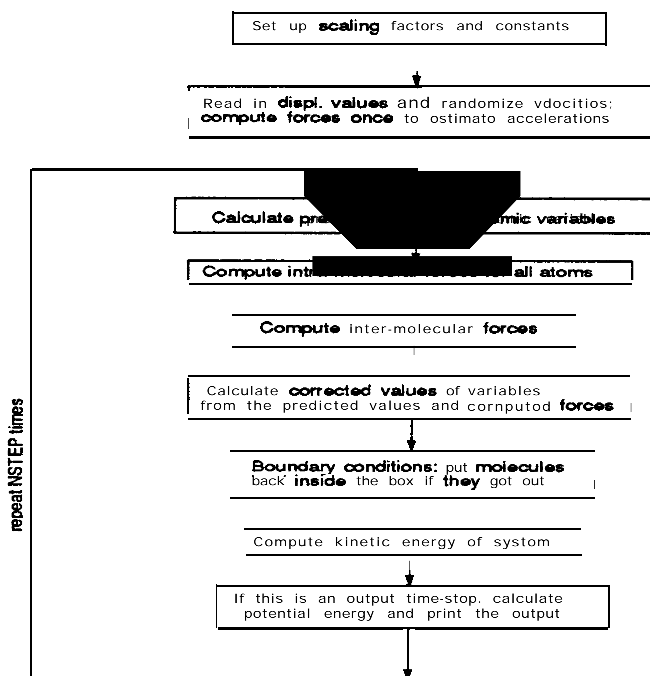


Figure 3: Structure of the Water program.

single task type (computing intermolecular interactions) takes up almost all the execution time, being $0(n^2)$ in the number of molecules as compared to the rest of the application which is $0(n)$.

8.2.1 Partitioning/Scheduling and Locality

Every box in the flowchart of Figure 3 is composed of one or more tasks (as defined above). Each of these tasks is partitioned among all processes. The parallelism available across tasks is used to reduce the amount of synchronization required between ***tasks.*** For the ***intramolecular*** computation, parallelism is mainly implemented by assigning every processor a set of molecules that 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 or set of molecules in every computation; that is, static scheduling is used. Data access patterns are completely predictable, both within and across time-steps, and there is no tradeoff with load balancing. The data for a processor's set of molecules might be allocated on the memory module closest to that processor on a machine with physically distributed main memory. This is not as easy to do in this case as in the Ocean program, since the granularity of a molecule's data is much smaller than that of a page which is the unit of data distribution. However, since molecules are statically assigned to processors in this version of the program, the distribution has to be done only once. There is no communication in the intramolecular computations, except in the small number of accumulations to a global sum every time-step. The impact of this communication can be arbitrarily reduced by increasing the problem size. The intramolecular computations do not afford any blocking for reuse of cached data within a task Some blocking is available across tasks, but is not **likely** to be very useful for realistic problems.

Ignoring the cutoff radius for the time being, every processor has simply to compute $\frac{n^2}{2p}$ interactions of the pool of $\frac{n^2}{2}$ for load balancing. The only locality issue is in access patterns to the VAR array, and is optimally treated by the following mechanism: A processor computes the interactions for every molecule in its partition with the $\frac{n}{2}$ molecules following it in the VAR array (with wraparound at the end). Thus, a processor communicates with exactly half the other processors in the machine. Each interaction involves reading the particle positions of the interacting molecules, computing the forces, and writing the force locations of both molecules. This communication can be as bad as $0(\frac{n^2}{p})$ per processor, i.e. of the same order as the computation, if communicated data are replaced in the caches before reuse. However, the displacements that processors read

are not modified during the intermolecular computation (they are written by their owning processor in other phases). The communication is therefore $O(\frac{n}{2})$ per processor if caches arc large enough, and can be reduced to this even if they aren't by blocking the computation with respect to remote references: performing all the interactions for a molecule fetched from another partition before fetching the next molecule. Blocking is useful on a uniprocessor in reducing cache misses; on a multiprocessor, blocking with respect to remote references rather than local ones has the additional advantage of reducing communication. As in the Ocean application, the fact that the data-sharing patterns are phase-structured-displacement data that are read in the intermolecular interactions phase are not modified in that phase, but only later in the update phase-can be exploited by selective update-based cache coherence protocols or by techniques such as prefetching. These communication patterns, and the fact that comunicated data are reused a lot in this most time-consuming phase of the application, make the caching and reuse of communicated data critical to effective parallel performance. Data distribution in main memory does not help very much in this phase, since most of a processor's shared references are to data from other processors' molecules, and data distribution will not help these references be satisfied locally. Long cache lines are useful to the extent that reading some kinds of data (say, displacement data) of a molecule involves reading several bytes (a double precision number per atom and per spatial dimension). Even longer lines could be exploited more effectively if the data structures were reorganized to keep the displacement (or velocity etc.) data for all molecules contiguous, rather than all the data for a given molecule. Finally, the active write-sharing on updates can be reduced to $O(\frac{n}{2})$ as well by giving every processor a private, temporary force array which it accumulates into the shared force locations when all interactions are completed.

Although all the data associated with a molecule are declared to be shared, only the positions, accelerations and forces are actually shared, each datum by no more than half the processors. Despite the communication being with half the machine, even the intermolecular interactions display good locality properties in their data referencing, since almost all the references for an interaction are to data that are private to the processor and are reused in every interaction it computes. Since almost all the execution time is spent in the interactions, the fraction of references to data that are even declared to be shared (let alone actually shared), is quite small (see Table 4).

The cutoff radius complicates both load balancing and data locality by introducing another form of locality: locality in the virtual space defined by the box and replicated versions of it about its eight surfaces (owing to the periodic boundary conditions). Since molecules have greater interactions [6] with other molecules that are close to them in this virtual space than with those that are outside the cutoff radius, there is potential for load imbalances if the partitioning does not assign roughly an equal number of actually computed interactions to every processor. Data locality also asks that molecules assigned to the same processor be close to one another another (or to one another's replicated images) as far as possible. The mapping of partitions onto a network topology becomes an issue, and the movement of particles with time can complicate all these considerations. This program makes no explicit attempt to incorporate locality in space. No spatial data structure is maintained [7], so all potential interactions have to be examined. If the molecules are inappropriately ordered at the beginning or move into such an inappropriate configuration with time, both load-balancing and data locality can be compromised. The fairly uniform distribution and randomized motion of water molecules in liquid state, however, makes this very unlikely.

8.2.2 Synchronization and Granularity

Barriers are used to maintain dependences where static scheduling doesn't suffice; i.e. before and after computing intermolecular interactions. The barriers can be replaced by more specific processor-to-processor synchronization, but this is quite fine-grained and is not implemented in the current program. Only a small number of barriers is used every time-step, this number being independent of the problem size or the number of processors.

[6] Note that every molecular interaction is composed of fourteen sub-interactions, each of which may or may not be computed in a particular case.

[7] Owing to the periodic boundary conditions and the cutoff radius being half the box length, most of the interactions examined are actually computed. The replication of space and the movement of particles between time-steps also complicate the use of a spatial data structure. All these facts make its omission quite reasonable. If the number of molecules is increased significantly, the cutoff radius will no longer be calculated as half the box length, but will instead remain fixed at 11-12 Angstrom. In this case, a spatial data structure can greatly reduce the number of molecule pairs examined. The computation of intermolecular forces is then $O(n)$, and the communication per processor is constant A new version of the Water application, which uses a spatial data structure to efficiently solve large problems, will soon be added to the SPLASH suite.

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 once every time-step.
- Since different processes may simultaneously be computing intermolecular interactions involving the same molecule, updates to the force locations of all atoms must be mutually exclusive. This implementation uses locks at the granularity of individual molecules. In the current version of the program, the number of these locked updates is about $\frac{n^2}{2}$ per phase of intermolecular interactions. It can be reduced to $\frac{np}{2}$ at the cost of some storage by giving every processor a private force array. Also, a processor need not lock when accessing the force locations of its own molecules. Each lock is accessed by half the processors.

Granularity in Water is measured as the amount of computation per processor between successive barrier synchronizations. This granularity is typically quite large (0($\frac{n^2}{p}$) in the intermolecular interactions and 0($\frac{n}{p}$)), and the overhead of synchronization relative to computation can be arbitrarily reduced by increasing the problem size. All the critical sections in the program are quite small: no more than a few machine instructions. Other than accumulations into shared sums, there are no race conditions in the program.

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 4 shows the variation, with problem size, in the amount of time spent in intermolecular interactions and in the rest of the program, timing the program as in Section 12.5.

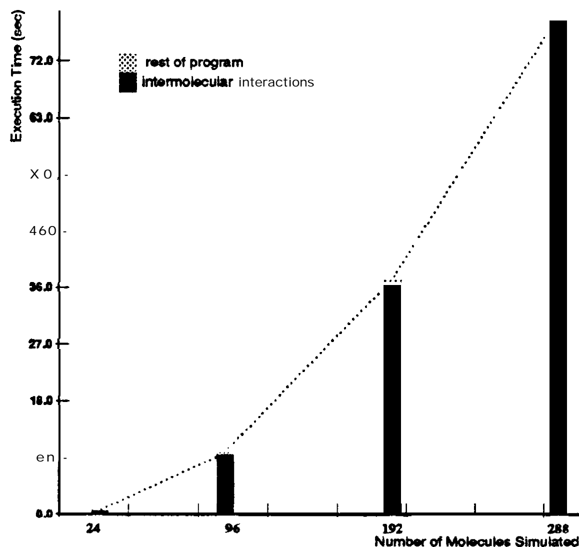


Figure 4: Water: uniprocessor execution time versus number of grid points

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 regular three-dimensional lattice. The file `random. in` contains pseudo-random numbers used to initialize particle velocities. If you use more than about 585 molecules, you will run out of random numbers in this file and will have to generate your own. 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 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. The use of a much larger number of molecules would retain the same absolute cutoff radius, and would therefore asymptotically transform the interactions from $O(n^2)$ to $O(n)$ (see previous footnote). 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.

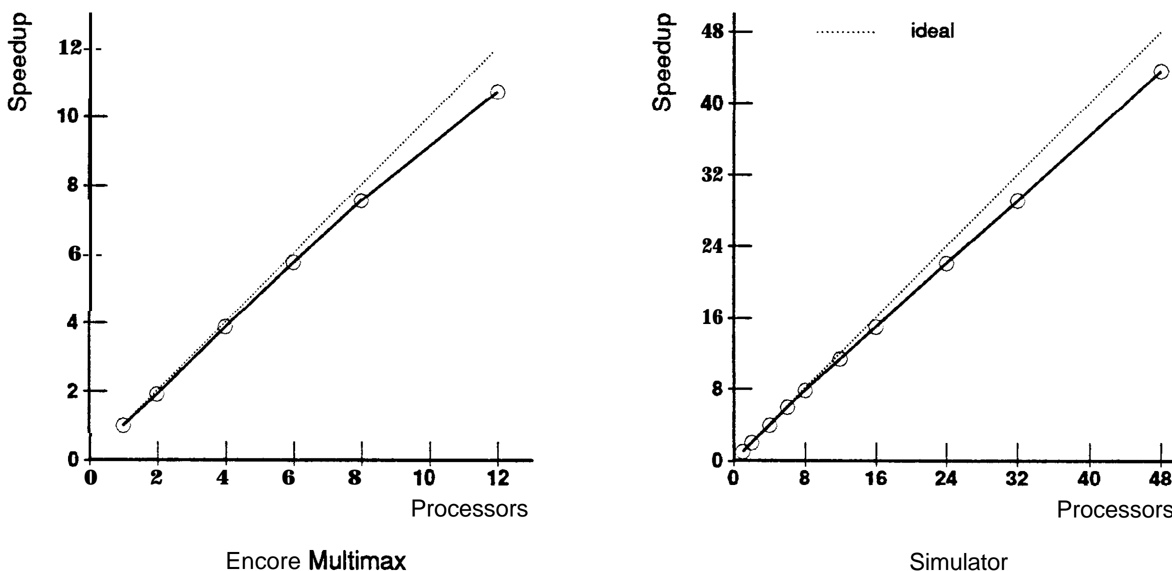


Figure 5: Water Speedups: Multimax and Simulator.

Speedups obtained on the Multimax are shown in the left graph of Figure 5. Measurements obtained

with the simulator are shown in the right graph of Figure 5 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. With 48 processors, the average processor spends only 0.2% of its time waiting at barriers, and negligible time at critical sections. Overall miss rates are very small with infinite caches, although miss rates to shared data are significantly higher. This is explained by the fact that only about 20% or the references are to shared data. Most of the misses are read misses to shared data (reading displacement data from other molecules when computing intermolecular interactions), and both the overall as well as the read miss rates are much higher than the corresponding write miss rates. Problem sixes for this application are likely to be limited by the physical usefulness of the simulation results for large numbers of molecules, rather than by the machine's memory.

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

8.6 Scaling to Larger Problems

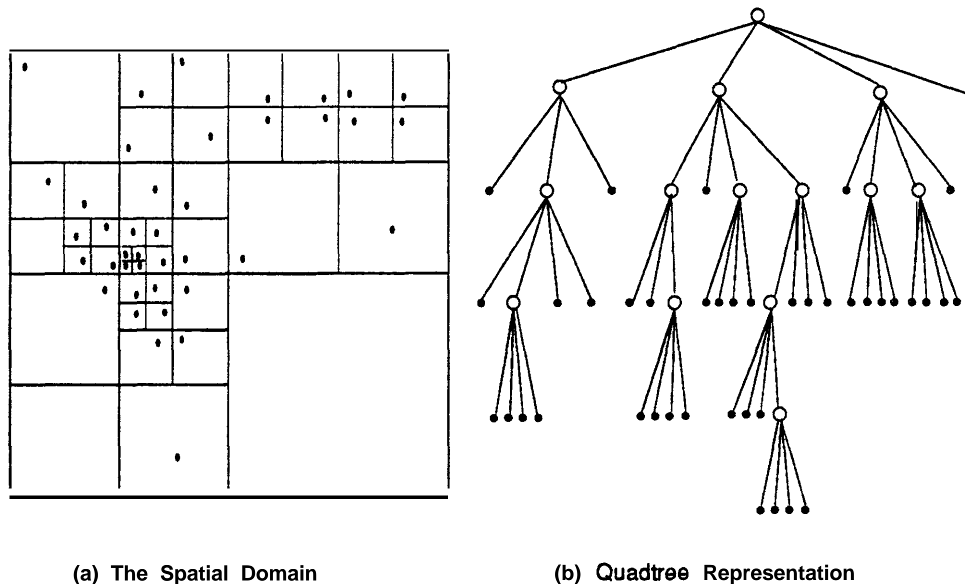
There are certain properties of liquid water to study which one needs to simulate substantially more than 343 molecules. For example, one property that this application can be used to study is the speed of sound waves in water. The maximum wavelength that can be studied in a system of water molecules is proportional to the Knudsen number (or K-number) of the system, which is itself inversely proportional to the system dimension. Studying this and other properties that require large h--numbers therefore requires increasing the system dimensions and hence the number of molecules. Neither the density of the system nor the absolute value of the cutoff radius change with the number of molecules, however. The physical time-step interval also need not change since it only needs to be sufficiently smaller than the vibration period of a molecule. The only parameter that may change is the number of time-steps that the application is run for, and how that changes depends rather unpredictably on how the properties being studied converge with larger systems. If only the number of molecules is scaled, the both the communication to computation ratio and the granularity of computation between synchronization events remains about the same as larger problems are run on larger machines. Of course, the 0(n^2) complexity of the algorithm dictates that the number of molecules cannot be scaled linearly with the number of processors when scaling under a fixed-execution-time constraint.

9 Barnes-Hut

This application simulates the evolution of a system of bodies under the influence of gravitational forces. It is a classical gravitational N-body simulation, in which every body is modeled as a point mass and exerts forces on all other bodies in the system. The simulation proceeds over time-steps, each step computing the net force on every body and thereby updating that body's position and other attributes. By far the greatest fraction of the sequential execution time is spent in the force computation phase. If all pairwise forces are computed directly, this has a time complexity that is 0(n^2) in the number of bodies. Since an 0(n^2) complexity makes simulating

large systems impractical, hierarchical tree-based methods have been developed that reduce the complexity to 0(n log n) [18] for general distributions, or even 0(n) for uniform distributions [22]. This application uses the 0(n log n) Barnes-Hut algorithm*.

The Barnes-Hut algorithm is based on a hierarchical octree representation of space in three dimensions (in two dimensions, a quadtree representation is used)[9]. The root of this tree represents a space cell containing all bodies in the system. The tree is built by adding particles into the initially empty root cell, and subdividing a cell into its eight children as soon as it contains more than a single body. The result is a tree whose internal nodes are cells and whose leaves are individual bodies. Empty cells resulting from a cell subdivision are ignored The tree (and the Barnes-Hut algorithm) is therefore adaptive in that it extends to more levels in regions that have high particle densities. Although we use a three-dimensional problem in this paper, Figure 6 shows a small two-dimensional example domain and the corresponding quadtree.



(a) The Spatial Domain (b) Quadtree Representation

Figure 6: A two-dimensional particle distribution and the corresponding quadtree.

The tree is traversed once per body to compute the net force acting on that body. The force-calculation algorithm for a body starts at the root of the tree and conducts the following test recursively for every cell it visits. If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single particle at the center of mass of the cell, and the force this center of mass exerts on the body computed If, however, the center of mass is not far enough away, the cell must be "opened" and each of its subcells visited A cell is determined to be far enough away if the following condition is satisfied:

$$\frac{l}{d} < \theta \qquad (1)$$

where l is the length of a side of the cell, d is the distance of the body from the center of mass of the cell, and θ is a user-defined accuracy parameter (θ is usually between 0.5 and 1.2). In this way, a body traverses deeper down those parts of the tree which represent space that is physically close to it, and groups distant bodies at a hierarchy of length scales.

9.1 Principal Data Structures

Conceptually, the main data structure in the application is the Barnes-Hut tree, Since the tree changes every time-step, it is implemented in the program with two arrays: an array of bodies that are leaves of the tree,

[8]We have also developed a parallel version of the application using the adaptive Fast Multipole Method. This application will soon be incorporated in the SPLASH suite.

[9]An octree is a tree in which every node has a maximum of eight children. In a quadtree, the maximum number of children is four.

and an array of nonterminal (internal) cells in the tree. Among other information, every cell has pointers to its children, and it is these pointers that maintain the current structure of the tree. The structure representing a body holds 94 bytes of data. If quadrupole moments are not used in the force calculation (see Section 9.3), a cell holds 72 bytes in a three-dimensional problem and 56 bytes in two dimensions. If quadrupole moments are used, these cell numbers are increased to 144 and 128, respectively. There is also a separate array of pointers to bodies and one of pointers to cells. These arrays are used by the processors to determine which bodies and cells they own: Every processor owns an equal contiguous chunk of pointers in these arrays, each chunk sized to be larger than the maximum number of bodies or cells a processor is expected to own. The total data space of the program is linearly proportional to the number of bodies for both uniform distributions (balanced tree) and nonuniform ones.

Partitioning using Orthogonal Recursive Bisection (see Section 9.2.1) introduces a separate, balanced ORB tree whose depth is the log of the number of processors used

9.2 Structure of the Parallel Program

The program first initializes the particle positions and velocities, and then iterates over a fixed number of time-steps. The structure of a time-step is shown in Figure 7. Over 90% of the sequential execution time is spent in computing the intexparticle forces, this percentage increasing with problem size.

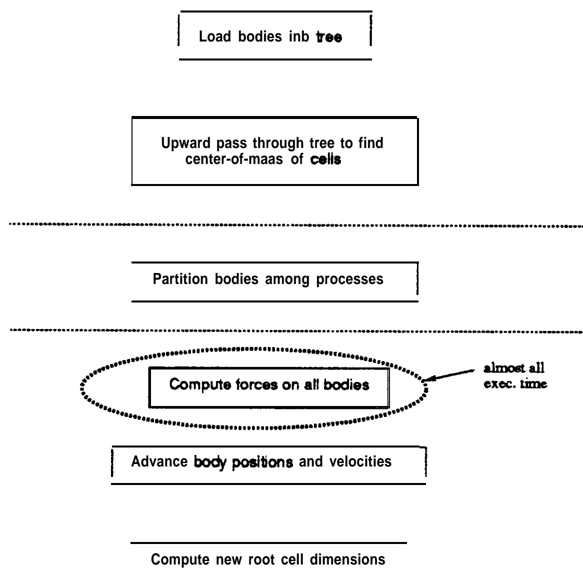


Figure 7: Structure of a time-step in the Barnes-Hut application.

Each of the phases within a time-step is executed in parallel. We do not exploit parallelism across phases or time-steps explicitly, except to eliminate synchronization between phases when unnecessary and thereby allow phases to overlap in a pipelined fashion. The parallelism exploited in all phases is across particles, except in computing the cell centers of mass, where it is across cells. The tree-building and center-of-mass computation phases require both interprocessor communication and synchronization, since different processors might try to simultaneously modify the same part of the tree. Although the force calculation for a particle requires the communication of position and mass information from other particles and cells, that information is not modified during the force calculation phase. Forces on different particles can therefore be computed in parallel without synchronization. Finally, the work for a particle in the update phase is entirely local to that particle and requires neither communication nor synchronization.

9.2.1 Partitioning/Scheduling and Locality

Obtaining effective parallel performance on the Barnes-Hut application is complicated by several characteristics of the problem and algorithm: the nonuniformity of the domain, which leads to highly nonuniform distributions of workload and communication among the units of parallelism (particles); the dynamically changing nature of the particle distribution; the need for unstructured, long-range communication; and the fact that different phases of computation (see Figure 7) have different preferred partitionings [23]. This program focuses its partitioning efforts on the force-computation phase, since it is the most time-consuming. The partitioning is not modified for other phases since the overhead of doing so (both in partitioning and in the loss of locality) outweighs the potential benefits.

Load balancing is obtained by counting the work done by a particle (i.e. counting the number of interactions it computes with other particles or cells) in a time-step, and using this work count as a measure of the work associated with that particle in the next time-step. This method works well because the system evolves slowly and the particle distribution does not change too much between successive time-steps. Work counting is also very cheap, since it only involves incrementing a counter when an interaction is performed, and is therefore done in parallel. Data locality is provided by exploiting ***physical locality*** in the problem domain: A partition should be spatially contiguous and, ideally, equally sized in all directions. Such partitions minimize interprocessor communication and maximize data reuse. The program uses one of the following two partitioning techniques (determined by a compile-time flag, see Section 9.3) to provide both load balancing and data locality [23].

Costzones The Barnes-Hut algorithm already has a representation of the spatial distribution encoded in its tree data structure. In the ***costzones*** partitioning scheme, the tree is conceptually laid out in a two-dimensional plane, with a cell's children laid out from left to right in increasing order of child number. The cost of (or work associated with) every body is profiled and stored with it. A cell stores the total work associated with all the bodies it contains. The total work in the system is divided among processors so that every processor has a contiguous, equal range or zone of work (for example, a total work of 1000 units would be split among 10 processors so that zone 1- 100 units is assigned to the first processor, zone 101-200 to the second, and so on). Which cost zone a body in the tree belongs to is determined by the total cost of an inorder traversal of the tree up to that body. Processors then traverse the tree in parallel, picking up the bodies that belong in their cost zone. The partitioning algorithm requires only a few lines of code, has negligible runtime overhead and yields a very good load balance. Its only problem is that it yields contiguity of partitions in the tree, which does not always match contiguity in space (see Figure 8). However, the number of processors is almost always very small compared to the number of particles, so that processes mostly pick up large cells from upper levels of the tree. Enough locality is obtained within each of these cells, and the number of transitions between them is negligible in comparison.

Orthogonal Recursive Bisection (ORB) ORB [19] is a more robust technique for preserving physical locality, because its actually partitions space rather than the Barnes-Hut tree. The tree is not used in the partitioning process at all. The idea here is to recursively divide space into two subspaces with equal cost, until there is one subspace per processor (see Figure 8). Initially, all processors are associated with the entire domain space. Every time a space is divided, half the processors associated with it are assigned to each of the subspaces that result. The Cartesian direction in which division takes place is usually alternated with successive divisions, and a parallel median finder is used to determine where to split the current subspace in the direction chosen for the split. Further details of implementing ORB are omitted for reasons of space; a more detailed description of its application to this problem is provided by Salmon [20]. ORB introduces several new data structures, including a separate binary ORB tree whose nodes are the recursively divided subspaces with their processor subsets, and whose leaves are the final spatial partitions. It also has a lot more runtime overhead, and is far more complex to implement than the tree-partitioning scheme we described above. Finally, it restricts the number of processors used to a power of two, although more complex forms of the technique can possibly be developed that do not have this restriction.

Spatial contiguity of partitions reduces the amount of communication in the tree-building and center-of-mass computation phases as well as in the force-computation phase. As in the Ocean and Water applications, the phase-structured data sharing patterns can be exploited by coherence protocols and latency hiding techniques. In

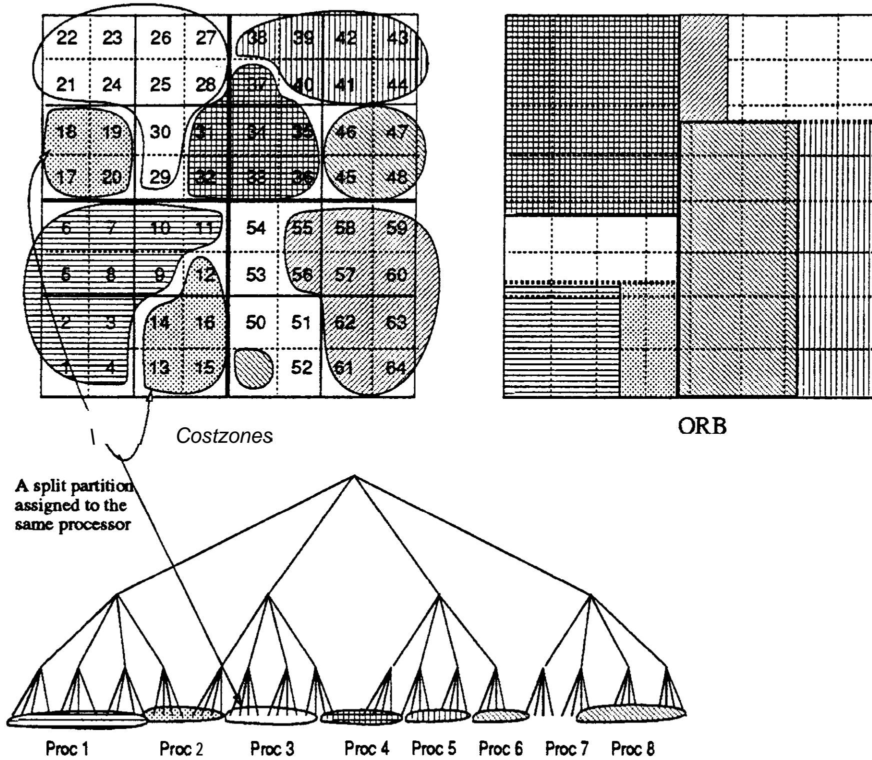


Figure 8: Tree Partitioning and Spatial Locality

this case, the information read from other particles in the force-computation phase is not modified in that phase, but rather later on in the update phase in which it is not shared. In fact, it is the temporal locality provided by caching that is the key form of locality in the application [21]. Distributing data to have a processor's partition be in its local main memory unit is not very worthwhile, for several reasons: (i) data have to be redistributed dynamically, which is expensive; (ii) such redistribution is difficult because the logical granularity of data (a particle/cell) is much smaller than the physical granularity of data migration (a page in physical memory), and spatial locality in the data structures does not match the locality in the physical domain that the application wants to exploit; and (iii) many of the shared misses in the application are to data from other processors' partitions, which have to go across the network even if data are redistributed-that is, data distribution is not likely to help very much anyway. As in the Water application, long cache lines help to the extent that reading a particle's displacement data (for example) involves reading several bytes of data. However, very long cache lines might cause more fragmentation than useful prefetch, since locality in the data structures does not match that in physical space.

9.2.2 Synchronization and Granularity

Barriers are used to maintain dependences across some phases (see Figure 7). These barriers cannot be replaced by synchronization between statically known pairs or subsets of processors, but only by dynamically determined synchronization at the level of individual data elements. The (small) number of barriers used in a time-step is independent of problem size or number of processors. ORB partitioning also uses some barriers internally, the number of these being proportional to the log of the number of processors. Another type of synchronization used is mutual exclusion through locks. Locks are used to protect global variables, and arrays of locks to protect cells when building the tree or computing centers of mass in parallel. The number of locks in the program can be chosen to be smaller than the number of cells, in which case multiple cells will share a lock. Finally, Event synchronization using regular variables as flags is used, for example, when a cell has to wait for its childrens' centers of mass to be computed before it can compute its own center of mass. The granularity of computation between synchronization points is large, particularly in the force computation and update phases, where it is

$O(\frac{n \log n}{p})$ and $O(\frac{n}{p})$, respectively. There is no synchronization within these phases. The need for locking cells in the tree-building and center-of-mass phases causes the granularity in those phases to be substantially smaller. However, synchronization-is not expected to be a dominant source of overhead for practical problems, particularly with moderate numbers of processors.

9.3 Running the Program

93.1 Compile-time Options

Several compile-time flags can be used to determine the kind of initial distribution to be generated, whether a three-dimensional or two-dimensional problem is to be simulated, whether to generate snapshots of the execution as program `output`, etc. These flags are described in the `makef ile` provided with the source code. The flags we use in our results are `TWO-CLUSTER`, `THREEDIM`, and `OWNEDCELLS`.

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

9.3.2 Input and Output

A single input file is read. It contains input parameters, one per line, in the following order:

1. `inf ile`: a string specifying the input file from which the intial particle displacement is to be read. If this line is left blank in the input file, the program generates its own input distribution in accordance with the compile-time flags used. We do not provide an input distribution file, so a user would either have to provide it or use the distribution generated by the program. Default is null.
2. `nbody`: an integer specifying the number of bodies to be simulated Default is 128, a very small number.
3. `seed`: an integer seed for the random number generator used to determine initial conditions. Default is 123.
4. `out f ile`: a string specifying the output file in which to put particle displacement data. If this line is left blank, or if the program is compiled without the OUTPUT flag, such output is not generated. Default is null.
5. `dtime`: a double-precision number specifying the physical time between steps. Default is 0.025.
6. `eps`: the softening parameter used to avoid singularities when particles get too close to each other. Double precision, default is 0.05.
7. `tol`: a double-precision number specifying the value of the accuracy parameter θ. Default is 0.6.
8. `f cell s`: a double-precision number specifying the amount of memory that should be allocated for cells. The maximum number of cells is assumed to be `fcells * nbody`. Default is 0.8.
9. `t st op`: a double precision number specifying the physical time at which to stop the simulation. The number of time-steps run is therefore $\frac{tstop}{dtime} + 1$. Default is 0.25, i.e. 11 time-steps.
10. `dt out` : a double precision number specifying the physical time interval at which the current displacements are output to `out file`. Default is 0.25, i.e. write output only at end (output is only written if the OUTPUT compile-time flag is enabled).
11. `NPROC`: the number of processors to be used Default is 1.
12. `debug`: a number specifying the level of debugging information to be used Not used in this application so can be left blank. Default is 0.
13. `work-tolerance`: Only used if the program is compiled to use ORB partitioning. Specifies the tolerance for the median-finder used in ORB partitioning at the highest level as the specified work-tolerance divided by `nbody`. Subsequent lower levels in the ORB recursion halve this parameter every time. Default is 3.0.

A sample input file `(sample. in)` is provided. Some of the input parameters are echoed to `stdout`, and some timing values are written there as well upon completion.

9.4 Results

Our results are obtained with the following input parameters: infile = "", nbody = 8192, seed = "", outfile = "", dtime = 0.025, eps = 0.05, tol (θ) = 1.0, fcells = 2.0, tstop = 0.075 (4 steps), and dtout = 0.25 (no particle output). ***Costzones*** partitioning is used. Of the four time-steps, only the last two are included in the measurements. Only self-relative speedups are presented. Normalized speedups are not significantly different from these.

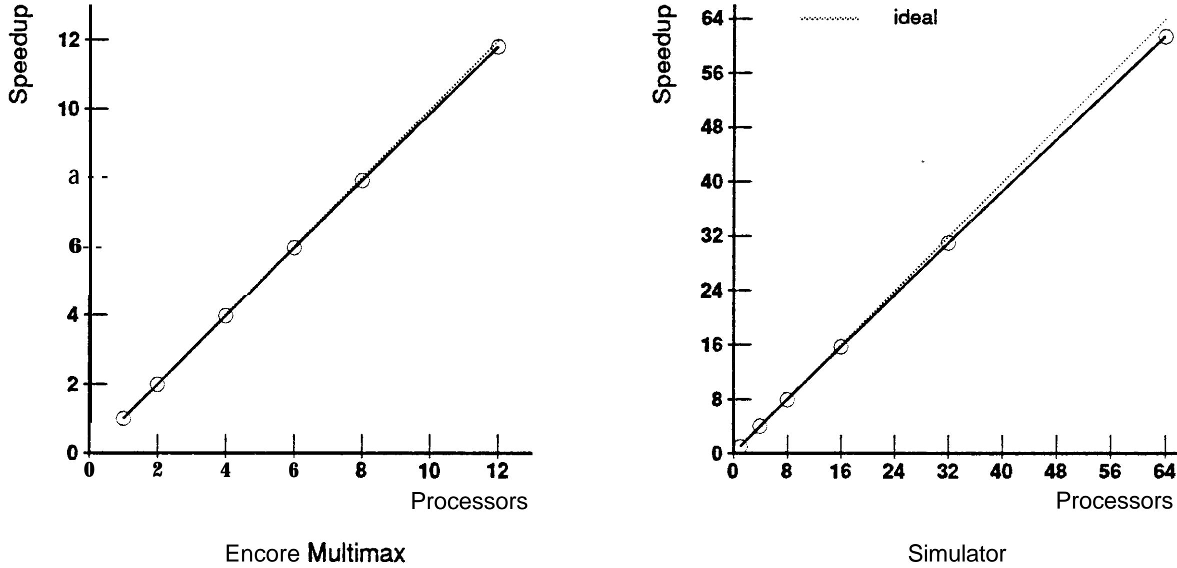


Figure 9: Barnes-Hut Speedups: Multimax and Simulator.

Speedups obtained on the Multimax are shown in the left graph of Figure 9. Measurements obtained with the simulator are shown in the right graph of Figure 9 and in Table 7. The synchronization wait time is small, and most of it is spent waiting at barriers between phases. The miss rates are also very small (particularly since over 80% of the references are to private data), indicating that the performance of the application is likely to scale very well. A much larger percentage of shared writes miss than do shared reads, but the fraction of writes that are to shared data is so small that the overall read miss rate is higher than the overall write miss rate.

Table 7: Barnes-Hut Information (Simulator).

Number of Processors	Miss Rate (%)	Synchronization Waiting Time (%)
1	0.00	0.00
4	0.07	0.48
8	0.11	1.32
16	0.17	2.23
32	0.24	3.32
64	0.36	4.03

9.5 Scaling to Larger Problems

Astrophysicists currently run about 64K- 128K particles with θ = 0.7-1 .O in realistic simulations. Three parameters are likely to be scaled to use more computing power: the number of bodies n, the force-calculation accuracy θ, and the physical time-step Δt. These parameters make the following contributions to the total simulation error [21]:

- n : The error from the increased relaxation rate due to Monte Carlo sampling scales as $\frac{1}{\sqrt{n}}$; thus, an increase in n by a factor of k leads to a decrease in simulation error by a factor of $\sqrt{k}$.
- Δt : The leap-frog method used to integrate the particle orbits has a global error of the order of Δt^2. Thus, reducing the error by a factor of $\sqrt{k}$ (to match that due to a k-fold increase in n) requires a decrease in Δt by a factor of $\sqrt[4]{k}$, and hence that many more time-steps to simulate a fixed amount of physical time (which is usually held constant).
- θ: force-calculation error scales proportional to θ^2 in the range of practical interest, for common distributions [10] Reducing the error by a factor of $\sqrt{k}$ thus requires a decrease in θ by a factor of $\sqrt[4]{k}$.

Scaling their error contributions equally implies the following scaling rule: If the number of bodies n is scaled by a factor of k, Δt and θ must each be scaled by a factor of $\frac{1}{\sqrt[4]{k}}$. The problems that astrophysicists currently run are still unbalanced in their error contributions, the error due to limited n being larger than the others. However, starting from a point where the errors are balanced, this scaling rule applies. Results for how communication and computation scale under different scaling models using this rule can be found in [21]. Essentially, under the most appropriate method of scaling (scaling all application parameters appropriately under a constraint on execution time), the communication to computation ratio increases, just as in the Ocean application. Also, the granularity of computation between synchronization points decreases in this case as well.

10 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.

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 [25].

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

[10]This is for the original Barnes-Hut algorithm that does not incorporate quadrupole corrections to the force approximation, which is the version we use; if quadrupole corrections are incorporated, the error scales as θ^4.

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 [24].

10.1 Principal Data Structures

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.

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.

10.2 Structure of the Parallel Application

A high-level flowchart of MP3D is shown in Figure 10. 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 $\vec{x} = \vec{x} + \vec{v}\,\Delta t$, the cell statistics are updated, and collisions with boundaries and other molecules are performed incrementally[11]. 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.

10.2.1 Partitioning/Scheduling and Locality

The work is partitioned by molecules, which are statically scheduled on processors. A given molecule, therefore, is always moved by the same processor. The partitioning of molecules, however, is in no way related to their positions in space, which change significantly with time. Access patterns to the space array therefore show much lower processor locality, which can severely degrade parallel performance. It is also quite possible for different processors to access a given space cell during the same time-step. Since a spatial data structure is explicitly maintained, an attractive alternative to provide locality in both data structures is to distribute perhaps irregularly shaped regions of space (together with the molecules they contain) among processors. This works well if regions can be chosen to follow the flow of the molecules, so that molecules don't migrate much between them, and gives rise to interesting challenges in scheduling simultaneously for data locality and load balancing. Note that load balancing depends on the number of collisions per particle, and therefore on the input geometry, even if all memory references are free.

Data sharing occurs during collisions and during accesses to the space array. Both of these depend on the number of molecules in the simulation, and on the input geometry. Scaling the problem size for MP3D is limited only by the memory available on the machine.

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

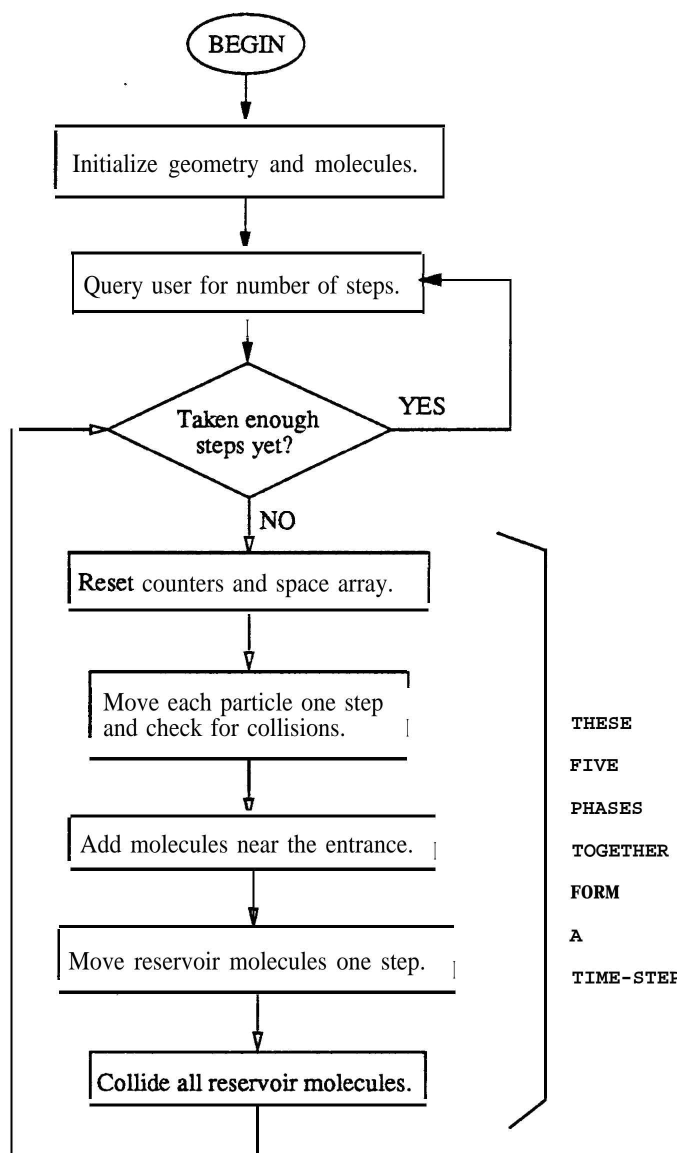


Figure 10: Flowchart of MP3D.

10.2.2 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. The number of barriers per time-step is small and is independent of the problem size and number of processors. 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. Once again, the relative overhead of synchronization can be arbitrarily reduced by increasing the problem size (unless the space array is locked).

10.3 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 8: 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.

10.4 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.

10.4.1 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 10.5. Other geometries may be substituted for this. After each period of simulation, the application prints the execution time.

10.5 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. Our runs were for 50 time-steps.

Speedups obtained on the Multimax are shown in the left graph of Figure 11. 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 11. For 64 processors, the speedup ratio is 52. Table 9 presents miss rates and synchronization waiting times. In the 64-processor run, only 0.76% of the time is spent waiting on locks,

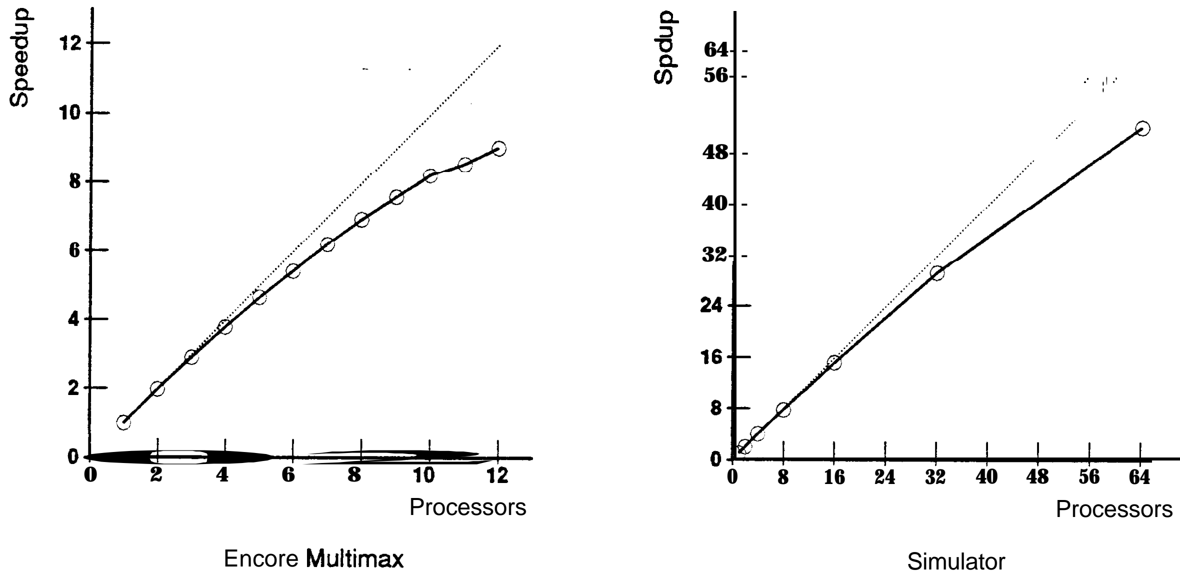


Figure 11: MP3D Speedups: Multimax and Simulator.

Table 9: 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

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 over-ah 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 [24], 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.

1 1 LocusRoute

LocusRoute [26, 27, 28] 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 [12]) 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.

[12] 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.

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.

11.1 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 12 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.

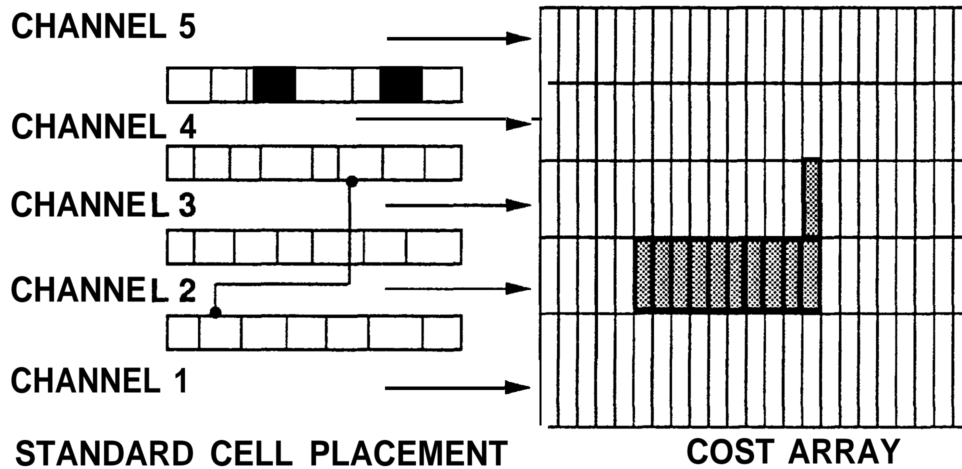


Figure 12: LocusRoute: Standard Cell Circuit and Corresponding Cost Array.

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[13] route has 3 straight segments. There are $n - 1$ two-bend routes for a wire with n pin-groups[14]. 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.

11.2 Structure of the Parallel Program

The program begins by reading the input files, initializing data structures, and spawning additional processes (see Figure 13). 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

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

[14] 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.

order of routing, two iterations are performed[15]. 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.

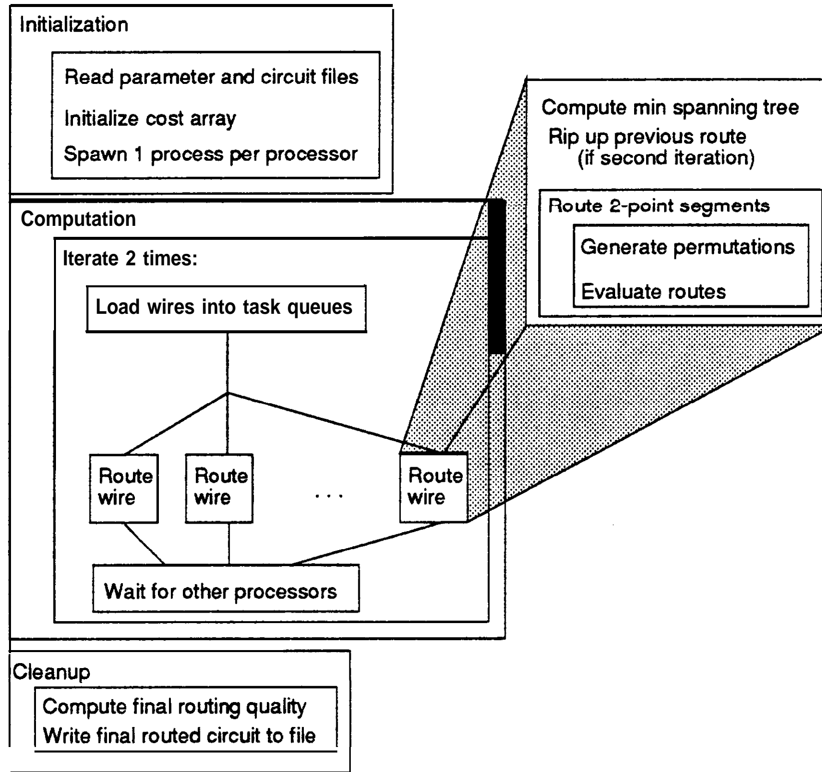


Figure 13: Execution Flowchart for LocusRoute.

11.2.1 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 11.5 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 11.5 use a "geographical" method of wire partitioning. In this method, the circuit is divided into spatial 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. Since references to the cost array are the dominant form of data sharing, geographic scheduling improves the data locality in the program[16]. It also improves the quality of the resulting routing (see section on

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

[16] Note that long wires spanning several regions may violate this locality, which may be a reason to split long wires into multiple segments

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.

11.2.2 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.

11.3 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 10 shows a breakdown of uniprocessor execution time spent in the major routines. The first input circuit (bnrE) has 5 11 wires in a 341 grid by 10 channel area The second circuit (Primary1) is larger, with 1266 wires in an 481 by 18 area

Table 10: 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

11.4 Running the Program

The program is run by typing the command:

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

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

11.4.1 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 wires and a 1290-by-20 cost array. The `ParameterFile` contains the parameter

---
for parallelization.

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 circuit into 15 regions. If more than 15 processors are used, several processors will be assigned to the same region.

LocusRoute generates an `Output F i le` 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.

11.5 Results

We present results for the **Primary2.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

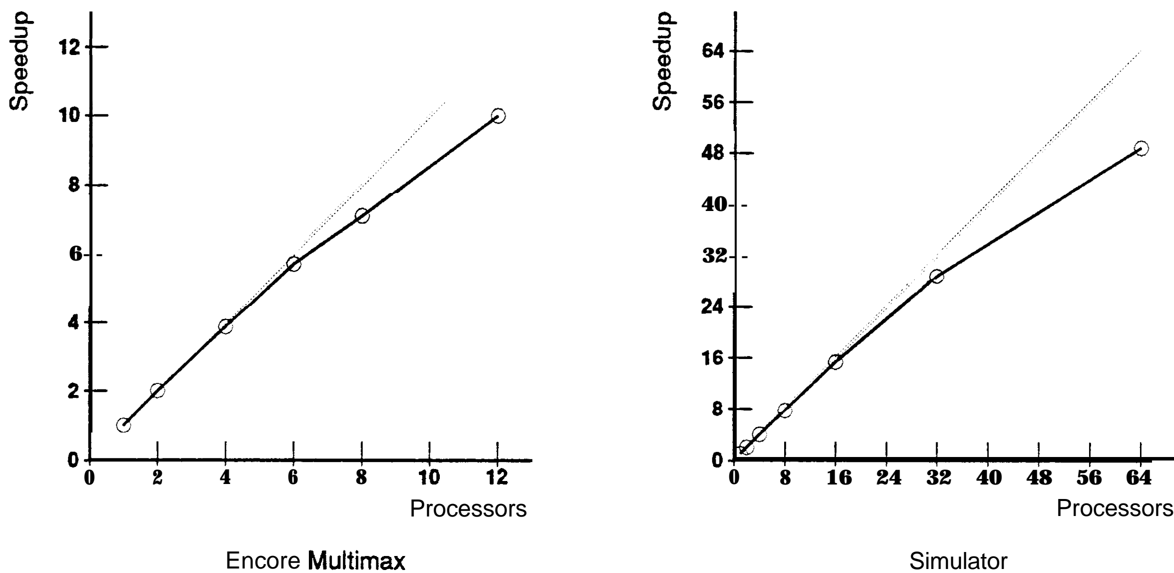


Figure 14: **LocusRoute** Speedups: **Multimax** and Simulator.

Speedups obtained on the **Multimax** are shown in the left graph of Figure 14. Measurements obtained with the simulator are shown in the right graph of the figure and in Table 11. 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. The limits on appropriate problem sixes for **LocusRoute** are determined by the sixes of realistic input circuits.

12 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.

Table 11: 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

PTHOR uses a variant of the Chandy-Misra [29] distributed-time algorithm (denoted CM). The CM algorithm will be described only briefly here; for an in-depth treatment see [30]. While the standard event-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.[17] When deadlocks do occur, they are resolved and the element evaluations continue as before.

12.1 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 outstanding on that node. There 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 48 bytes. Every event also takes up 12 bytes, and the number of events allocated depends on the circuit, input vectors, and length of the simulation. For the relatively small `ri s c` circuit used in our measurements, approximately 8,000 events are allocated. Every element structure takes up about 160 bytes plus some element dependent storage (e.g. pointers to input node arrays). For a typical element in the `ris c` circuit, the additional element-dependent storage averages about 70 bytes per element. Thus, the data set size for a simulation of this small circuit is about 1.5 Mbytes.

12.2 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.

12.2.1 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

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

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).

Besides having nodes pass from one processor to another to improve load balancing, there is also data sharing via the nets and in the global data structures. The communication requirements are basically linear with the number of element evaluations. Increasing the problem size will increase the number of element evaluations, but at a sublinear rate. These trends indicate that the scaling of PTHOR to larger problem sizes is limited only by the availability of these larger problems.

12.2.2 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).

12.3 Profile of a Uniprocessor Execution

For a one-processor simulation of the `risc` circuit with the run-time parameters described in Section 12.5, 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.

12.4 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.

12.4.1 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.min and stim before execution.

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.

12.5 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.

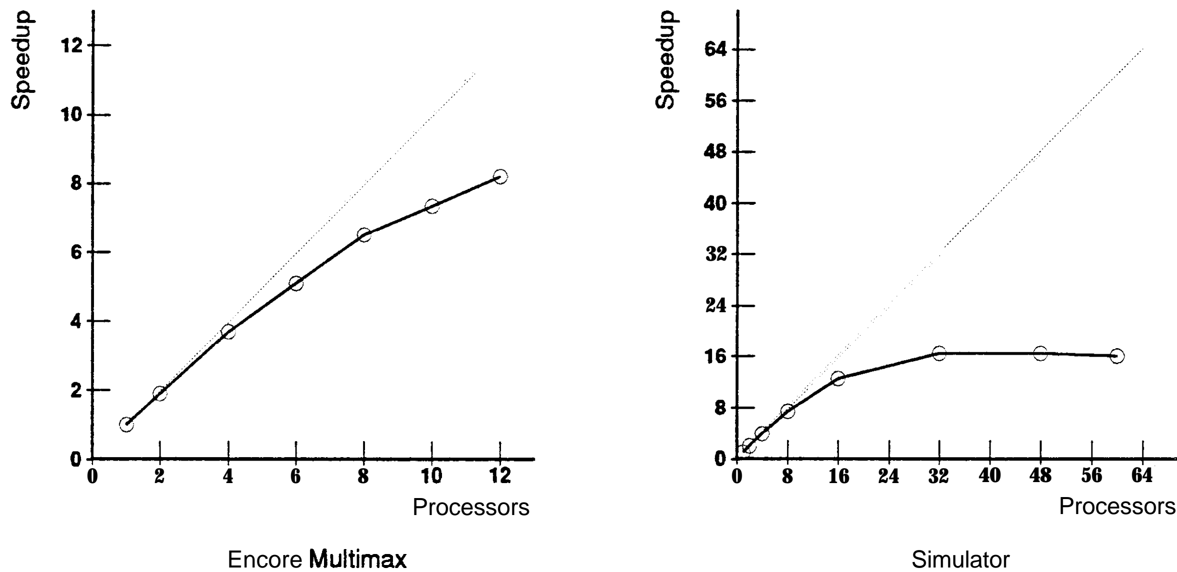


Figure 15: PTHOR Speedups: Multimax and Simulator.

Table 12: 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	329	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 15. Measurements obtained with the simulator are shown in the right graph of Figure 15 and in Table 12. 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. Like LocusRoute, problem sizes for PTHOR are limited by the sizes of realistic input circuits.

13 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*** [34], an enhancement of the fan-out method of [32]. 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.

13.1 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 [33]. 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.

13.2 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 distributing data to the processors handling each task would be too large. In supemodal 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. Data distribution overheads would still be too large even if a single supemodal modification were chosen as the grain size. Therefore, the task grain we use is the set of all supemodal modifications performed by a particular supemode. Although this might lead to worse load balancing, the gains in data locality are likely to make the tradeoff beneficial.

13.2.1 Partitioning/Scheduling and Locality

The parallel sparse factorization computation proceeds as follows. With each supernode, 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 supemodal modification, the incoming counts of the destination columns' supernodes are decremented to reflect the modifications done to them. If a supernode's count goes to zero, it is placed on the global task queue.

The data sharing patterns for each supemode are as follows. A supemode may be modified by several processors before it is placed on the task queue. Once this happens, it is read by a single processor and used to modify other supernodes. After it has completed all its modifications to other supernodes, it is no longer referenced by any processor.

13.2.2 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.

13.3 Profile of a Uniprocessor Execution

Almost all the sequential runtime is spent performing supemodal modifications.

13.4 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 bcsstkl4` 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.

13.4.1 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 [31]. `bcs tt k14` is a 1806-by-1806 matrix with 30,824 non-zeros in the matrix and 110,461 in the factor, it has 503 distinct supernodes, the largest of which contains 135 columns. The corresponding numbers for the larger matrix `bcsttkl5` are 3948-by-3948, 56934, 647274, 1295 and 211, respectively. Both matrices have been reordered using the minimum degree heuristic [33].

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

13.5 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.

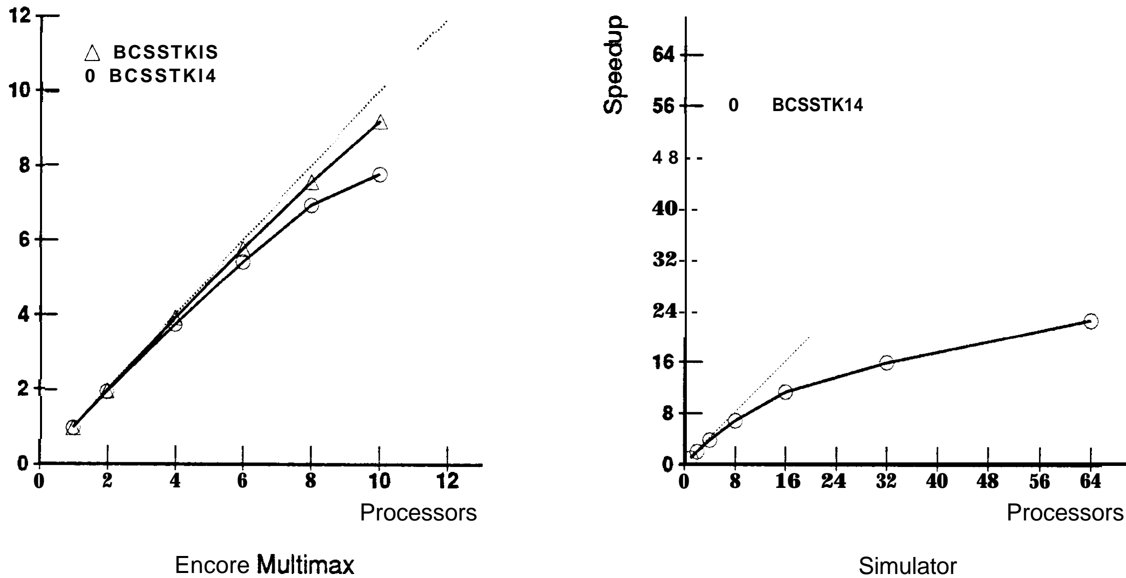


Figure 16: Cholesky Speedups: Multimax and Simulator.

Normalized speedups obtained on the Multimax are shown in the left graph of Figure 16. 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 16. 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.

Table 13: 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

Miss rates and synchronization waiting times observed on the simulator for the BCSSTK14 matrix are shown in Table 13. 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.

14 Concluding Remarks and Near-term Additions

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. In the near future, we will add the following applications or versions of applications:

- A galactic simulation using the Fast Multipole Method [22].
- A radiosity application from computer graphics that uses a hierarchical solution method.
- A version of the Ocean application with a multigrid solver instead of SOR.
- A version of the Water application with the spatial data structure to make the intermolecular force-calculation algorithm $O(n)$.

When an addition is made, electronic mail will be sent to our mailing list (see Section 2) and posted to the intemet electronic newsgroup `comp . parallel`. The new codes and updated versions of this report will appear in the anonymous ftp directory (Section 2).

15 Acknowledgements

We would like to thank the following people for running the programs and writing initial versions of the individual application reports: Steve Goldschmidt (MP3D), Margaret Martonosi (LocusRoute), Ed Rothberg (Cholesky) and Larry Soule (PTHOR). Okokon Okon helped with runs for the Ocean code. The work was supported by DARPA under Contract No. N00039-91-C-0138.

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-84-5 1, Rev. 1, Argonne National Laboratory, June 1987.

[4] J.P. Singh, J.L. Hennessy and A. Gupta, "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," submitted for publication.

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

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

[7] 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.

[8] J.P. Singh and J.L. Hennessy, "Finding and Exploiting Parallelism in an Ocean Simulation Program: Experience, Results and Implications," ***Joumal of Parallel and Distributed Computing,*** Vol. 15, No. 1, May 1992, pp. 27-48. Preliminary-version available as Tech. Report No. CSL-TR-89-388, Stanford University, Aug. 1989.

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

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

[11] J.P. Singh and J.L. Hennessy, "Data Locality and Memory System Performance in the Parallel Simulation of Ocean Eddy Currents," ***Proceedings of the Second Symposium on High Performance Computing,*** Montpelier, France, October 1991. Also Tech. Report No. CSL-TR-91-490, Stanford University, Aug. 1991. Available by anonymous `ftp` from `samay . s t anf ord . edu` (file papers/ocean-locality.ps).

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

[13] J.P. Singh and J.L. Hennessy, "An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelization," ***Proceedings of the International Symposium on Shared Memory Multiprocessing,*** Tokyo, April 1991. Also Tech. Report. No. CSLTR-91-462 Stanford University, 1991. Available by anonymous ftp from `samay. stanford.edu` (file papers/compilers.ps).

[14] 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.

[15] 0. Matsuoka, E.Clementi and M. Yoshimine, "CI Study of the Water Dimer Potential Surface," ***Journal of*** Chemical ***Physics,*** Vol. 64, No. 4, pp. 1351-61, Feb. 1976.

[16] R. Bartlett, I. Shavitt and G. Purvis, "The Quartic Force Field of H_2O Determined by Many-Body Methods that Include Quadruple Excitation Effects," ***Joumal of Chemical Physics,*** Vol. 7 1, No. 1, pp. 281-291, July 1979.

[17] 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.

[18] J.E. Barnes and ***I?*** Hut, "A ***Hierarchical O(N*** log ***N)*** Force Calculation ***Algorithm", Nature,*** Vol. 324, No. 4, pp. 446449, December 1986.

[19] G.C. Fox, "A Graphical Approach to Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube", in ***Numerical Algorithms for Modem Parallel Computer Architectures,*** ed. M. Schultz, Springer-Verlag, 1988, pp. 37-62.

[20] J.K. Salmon, "Parallel Hierarchical N-body Methods", Ph.D. Thesis, California Insitute of Technology, December 1990.

[21] J.P. Singh, J.L. Hennessy and A. Gupta, "Implications of Hierarchical N-Body Techniques for Multiprocessor Architecture", Technical Report CSL-TR-92-506, Stanford University, February 1992. Updated version available by anonymous `ftp` from `samay . Stanford. edu` (file nbody-arch.ps).

[22] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulation", ***Journal of Computational Physics,*** Vol. 73, No. 325, 1987.

[23] J.P. Singh, C. Holt, T. Totsuka, A. Gupta and J.L. Hennessy, "Load Balancing and Data Locality in Hierarchical N-body Methods", Technical Report CSL-TR-92-505, Stanford University, February 1992. Updated version available by anonymous ftp from `samay . st anf ord . edu` (file nbody-sched.ps).

[24] 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.

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

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

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

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

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