An Analysis of Distributed Computing Software and Hardware for Applications in Computational Physics

P. D. Coddington
Northeast Parallel Architectures Center
Syracuse University
Syracuse NY 13244, U.S.A.

Abstract

We have implemented a set of computational physics codes on a network of IBM RS/6000 workstations used as a distributed parallel computer. We compare the performance of the codes on this network, using both standard Ethernet connections and a fast prototype switch, and also on the nCUBE/2, a MIMD parallel computer. The algorithms used range from simple, local, and regular, to complex, non-local, and irregular. We describe our experiences with the hardware, software and parallel languages used, and discuss ideas for making distributed parallel computing on workstation networks more easily usable for computational physicists.

1 Introduction

Many academic institutions and research laboratories own large numbers of high performance workstations. These workstations are typically connected by a local area network (LAN) such as Ethernet. Surveys have shown that in most instances the average CPU usage of these workstations is less than 10%. Due to the wide availability of such networks, there is a huge resource of unused cycles available for doing high performance computing. For example, a network of 50 IBM RS/6000-550 machines represents a 1.1 Gflop computing resource.

The problems hindering the effective use of these workstation networks as powerful distributed computers are threefold – hardware, systems software and programming languages. Due to the relatively high latency and small bandwidth of Ethernet LANs, the type of applications which can be run effectively on these networks is limited. Sophisticated systems software is also needed to allow a heterogeneous network of workstations to be used in this way, without degrading the performance of workstations which are being used interactively (users can be very possessive of their own CPUs), and with some kind of load balancing. Finally there is the difficult problem of coding efficient parallel programs so that these loosely coupled networks of workstations can be used as parallel computers.

Here we describe our experiences in distributed computing using some applications from the field of computational physics; in particular Monte Carlo simulation of spin models of magnetic crystals. The same kind of techniques and algorithms are also used for the simulation of quantum field theories such as QCD (the theory of the strong nuclear force), and in simulations of theories of quantum gravity using dynamically triangulated random surfaces. Similar techniques are also used to tackle general optimization problems using simulated annealing. We will consider two types of Monte Carlo methods: the standard Metropolis algorithm, which is regular and local and therefore easy to parallelize; and the cluster-style algorithms, which are irregular and non-local and thus much more difficult to parallelize.

We have developed and studied a range of parallel algorithms for these problems on different parallel architectures. Here we describe the use of a network of workstations for distributed computing, and examine the performance of these parallel algorithms on such a network. We are particularly interested in the suitability of distributed computing for our physics codes, and whether new algorithms are required for efficient use of a distributed workstation network as a parallel computer.

The computing facilities used for this work consisted of 4 IBM RS/6000 model 340 workstations connected with both Ethernet and a high performance prototype switch [1]. This is a copper wired switch based on 8×8 prototype crossbars, with an adapter which connects directly to the Micro Channel of the RS/6000. The same programs have also been run on an nCUBE/2, a MIMD parallel computer with a hypercube topology.

The software we have used to make the network behave as a parallel computer is Express, from ParaSoft Inc. [2] Express is the result of research by the Caltech Concurrent Computation Group, and is a powerful and portable message passing environment which has tools for inter-processor communication, domain decomposition, parallel I/O, graphics, debugging and profiling. It can be used on most Commercial MIMD parallel computers, including those from nCUBE, Meiko and Intel, as well as on workstation networks. Express can be used with C or Fortran, which simplifies the modification of sequential code. The programs described below were written in Express Fortran.

2 Monte Carlo Simulation of Spin Models

Magnetic materials usually undergo a transition from a magnetic to a non-magnetic phase as the temperature is increased. These phase transitions can generally be described very well by simple models of interacting spins on a lattice, such as the Ising model [3, 4]. Numerical simulations of these spin models are mainly done using Monte Carlo methods such as the Metropolis algorithm [3, 4, 5]. This algorithm works by making small, local changes to the configuration of spins. This leads to the problem of "critical slowing down", meaning that many iterations are required to generate a new uncorrelated configuration, especially near a phase transition or critical point of the system. Non-local Monte Carlo algorithms have been introduced for some spin models [6, 7]. These so-called "cluster algorithms" can dramatically reduce critical slowing down, and thus give a huge improvement in computational efficiency over standard Monte Carlo algorithms. Here we look at two different types of cluster algorithm – the Wolff algorithm [7] which finds a single cluster to update, and the Swendsen-Wang (SW) algorithm [6] which partitions the entire lattice into clusters which are all updated.

Parallel algorithms for spin models offer an interesting challenge for parallel machines, languages, and compilers, since they have widely varying levels of difficulty. The local Metropolis algorithm can be easily and efficiently parallelized using standard domain decomposition, since it is regular and needs only local edge communication. This is a simple case which is easy to express in a language such as Fortran 90, and should be well handled by any parallelizing compiler. The cluster algorithms, on the other hand, are difficult to parallelize, since they are non-local and irregular, and lack the structure of regular multi-scale algorithms such as multi-grid. Consequently they provide a challenging problem for any parallelizing compiler, or even a high level parallel language.

3. The Metropolis Algorithm

The Metropolis algorithm for simulating a spin model is very similar to the standard algorithm for solving simple differential equations such as Laplace's equation on a discretized grid of points [8]. The update of the spin depends only on the values of neighboring spins, which means that a parallel implementation of this algorithm is very simple and efficient, since only local communication is required. Since the algorithm is regular we do not have to worry about load balancing.

The dependence of the update on the values of neighboring spins means that not all spins can be updated simultaneously in parallel. We need to set up a checkerboard or red/black partitioning of the lattice, in which all the red sites can be updated in parallel, and then all the black sites [8]. In this way the sites updated simultaneously are completely independent of one another, and the parallel algorithm therefore obeys the technical constraint known as "detailed balance" which ensures that the Monte Carlo procedure is valid [3, 4].

As long as this constraint is taken into account, writing a parallel Metropolis algorithm is very simple. In fact if we insist that there be more than 2^d sites per processor for a d-dimensional lattice and that we update one site at a time on each processor, then we automatically satisfy detailed balance. The resulting code on each node looks almost exactly like the sequential code, the only difference being that the function which is called to find the neighboring spin value handles periodic boundary conditions for the sequential code, while in the parallel code it performs message passing if the neighboring site is on another processor. This regular local communication is particularly simple in Fortran 90, with just a call to a periodic SHIFT operation.

This is the type of algorithm we would use for a fine grained parallel computer such as the CM-2, which has relatively low latency and high bandwidth nearest-neighbor communication. In Fig. 1 we show the results for the efficiency (the speed-up compared to the sequential program running on one node divided by

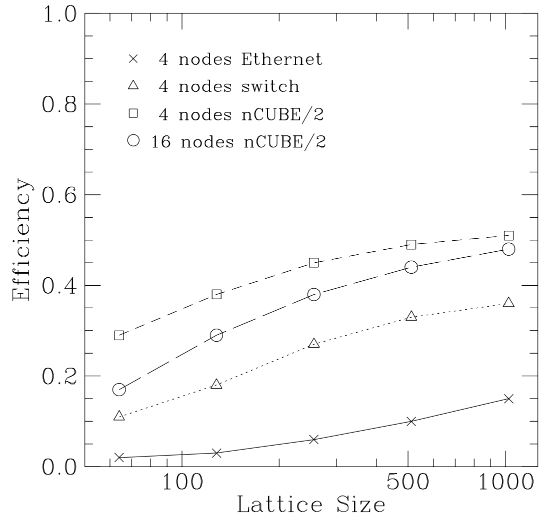

Figure 1: Efficiency of fine-grained Metropolis algorithm for different parallel architectures.

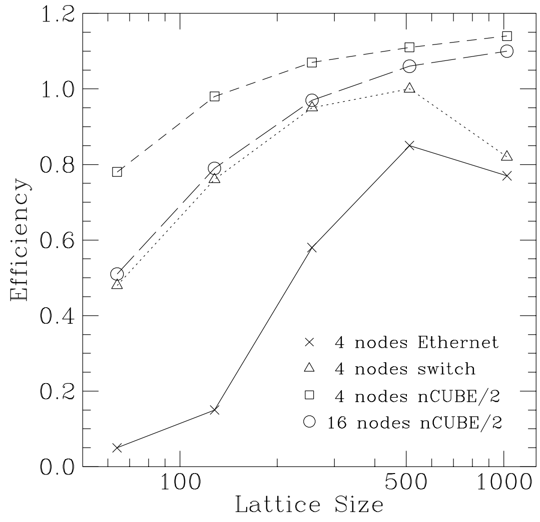

Figure 2: Efficiency of coarse-grained Metropolis algorithm for different parallel architectures.

the number of nodes) of the algorithm on different parallel architectures. Here the lattice size L refers to an L^2 lattice of sites. We can see that the fine grained algorithm also works reasonably well for coarser grained parallel machines such as the nCUBE/2. However this is clearly not an efficient algorithm for coarse grain parallel machines with high latencies, such as networks of workstations, since it involves passing a lot of very small packets of data (the spin value at a single site). In fact, using the 4 processor network is much slower than using a single processor. The calculation time for this problem scales linearly with the lattice volume, whereas the communication time scales as the length of the lattice. The time taken by this fine-grained parallel algorithm using the Ethernet LAN scaled roughly as the length of the lattice, and thus was was totally dominated by communication. Note that the low latency of the prototype switch gives dramatically improved performance over the standard Ethernet connections. The efficiencies are comparable with those for the nCUBE/2, even though the calculation time on the IBM workstations is very much smaller (for this problem the IBM RS/6000-340 processor is about 8 times faster than the custom nCUBE/2 processor).

For coarse grain machines this algorithm can be optimized by passing a smaller number of messages, each containing a larger amount of data. This can be done by using the red/black algorithm for the sublattice on each processor. In this case we can pack all the spin values along a processor boundary into a single message, which we only need to pass before every red or black update. This greatly reduces the latency, and since for a local coarse grain algorithm we have a large calculation-to-communication ratio (basically the ratio of the volume of the sub-lattice to its perimeter), we obtain very good efficiencies (see Fig. 2). Notice that the prototype switch still outperforms the Ethernet, and gives efficiencies close to that of the nCUBE/2. In some cases the efficiency is greater than 1, presumably due to more use of cache memory when the data is distributed over multiple processors. We believe that the decrease in efficiency at the largest lattice size for the workstation network is also an artifact of this kind. For the nCUBE/2, the performance of this algorithm is scalable to larger numbers of processors as long as the amount of data per processor remains constant, and this should also be true for a workstation network.

4. Cluster Algorithms

The main computational task in cluster algorithms is the identification and labeling of the clusters. This is an instance of the well-known problem of connected component labeling [9]. Connected component labeling is used in a number of other applications, most notably percolation theory and image processing. The problem is very simple: there are connections between sites of the lattice, and we want to give the same label to all sites which are part of a connected cluster, with each of these clusters having a different label. This means that local information (connectivity) must be processed and converted into global information (the cluster labels).

There are very efficient sequential algorithms for this problem which execute in a time of $O(V)$, where V is the number of sites [10]. However this is a very difficult problem to implement efficiently on a parallel machine, since sites in the same cluster may be on distant processors, and the path connecting two sites may be quite labyrinthine. Parallel component labeling algorithms are generally rather complicated, and can be very different to the sequential algorithms. There are many algorithms in the computer science literature for tackling this problem. We have studied a number of these methods, and have also developed and implemented our own algorithms for use in a parallel SW cluster update algorithm, for both MIMD and SIMD machines [10, 11, 12].

The extreme irregularity (in both size and shape) of the clusters in spin models means that this problem is not well suited to SIMD machines [12]. However MIMD parallel machines (which includes distributed workstation networks) can handle the irregular clusters, since they allow processors to follow branch points in the program completely independently. For our problem, this means that each processor can run the very efficient sequential labeling algorithms in parallel on their own section of the lattice. If we do a standard domain decomposition of the data so that each processor has the same number of lattice sites, the sequential algorithm will take approximately the same time for each processor, and we will also have good load balancing. A small amount of edge information then needs to be passed between processors in order to give the same labels to sub-clusters connected across processors. As long as the ratio of the number of sites per processor to the number of processors is relatively large, this method is fairly efficient on MIMD parallel computers [10].

This algorithm has been considerably optimized for coarse grain parallel computers, by minimizing both the amount of communication and the number of communication calls required. This optimization is vital for workstation networks where the latency is high. We can see from Fig. 3 that it allows us to obtain good performance, comparable to the much simpler and more regular parallel Metropolis algorithm. However we expect the parallel Metropolis algorithm to scale much better to larger networks, as is the case for larger numbers of nodes on parallel computers such as

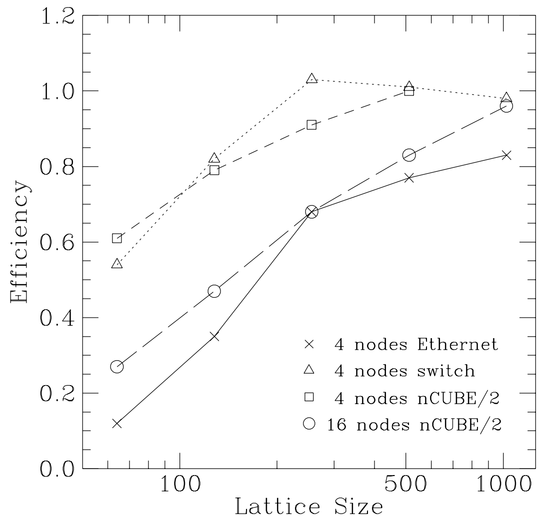

Figure 3: Efficiency of parallel cluster algorithm for different architectures.

the nCUBE/2 [10].

Note that the parallel labeling algorithm outlined above will only work efficiently for the SW algorithm, which identifies clusters on all the lattice sites. It is difficult to see how the Wolff algorithm, which only grows a single cluster, can be parallelized without having serious load imbalance.

3 Independent Parallelism

Quantitative studies of cluster Monte Carlo algorithms require extremely accurate data on large systems in order to make reliable calculations of quantities of interest, such as dynamic critical exponents. For the SW cluster algorithm we can use parallel or distributed computers to study larger systems, using the parallel algorithms discussed above. However the Wolff algorithm, which grows a single cluster, cannot be efficiently parallelized or vectorized. A network of high performance workstations is the most suitable architecture for simulations using this algorithm. Workstations have enough memory and power to handle large systems, and the IBM RS/6000 runs the Wolff code almost as fast as a CRAY X-MP.

Although the Wolff algorithm is not parallelizable over a single simulation, we can utilize the network by parallelizing over multiple simulations. The individual workstations in the network can be running simulations with different parameter values, such as the system size or the temperature. Independent parallelism is especially applicable to Monte Carlo simulation, which can be parallelized in this way even without varying the parameter values. This is because the Monte Carlo procedure is basically a sum over a large number of sample configurations of the system. This is a statistical process, with a corresponding statistical error which is proportional to the square root of the number of independent configurations sampled. Monte Carlo simulations are therefore "embarrassingly parallel", in that different processors can be generating different configurations completely independently, by using different starting configurations and different random number streams. These independent results can then be combined to reduce the statistical error in the simulation.

This kind of independent or "job level" parallelism is common to many kinds of simulations in science and engineering. Usually simulation involves studying changes in the system as parameters are varied. Sometimes the choice of new parameters will depend on the results of the simulation with the current parameters, so this is a sequential process. However in many cases one would like to know results for a large number of parameter values which can all be run independently.

As long as the system size is small enough so that each processor can hold all the data for a simulation, this procedure will give perfect speedup, since the sequential program is run independently on each processor. The data obtained using a network of workstations in this way allowed us to make accurate measurements of dynamic critical exponents for the Wolff algorithm for spin models [13, 14]. This method has also been used for Monte Carlo simulations of dynamically triangulated random surfaces, which are highly irregular and very difficult to parallelize [15, 16].

This approach is ideally suited to very coarse-grained parallel machines, and especially for distributed computing over networks of workstations. There are at least three methods of invoking job level parallelism on distributed networks, which we discuss below. We have done our physics production runs using the first two methods, neither of which we found to be satisfactory. This has led us to think of programming paradigms and systems software which would be ideal for these types of physics applications, and this is what we propose as the third method.

(1) The Brute Force Approach

The simplest but most tedious method is just to log on to any available machines in the network and submit a different job to each of these machines. This method is too time consuming to use on any more than a handful of workstations. It would be possible to set up a shell program to remotely run these jobs on different processors, however deciding which machines to use generally requires first checking that the machine is not already in use. Physicists and other scientists who might wish to use networked workstations in this way usually do not have (and should not need to have) the systems programming knowledge to implement such an approach automatically. Some systems software to deal with these problems would make the utilization of networked workstations very much easier.

(2) Portable Parallel Software

Job level parallelism can be used on distributed computing networks and MIMD parallel computers using parallel software. It is easy to set up parallel code to do this, since only data input and output need to be managed, as all computation is sequential and requires no communication. The problem with this method is that although the programs are run independently, they generally require synchronization every time data is input or output. This will lead to performance problems if one or more of the processors in the network is slower than the others, perhaps because it is also running other processes. This can be avoided by writing the data from each processor into separate files which can be accessed asynchronously. This can be done in Express, for example. This approach allows all the processors to run completely independently. However we are still left with the "load balancing" problem, of arranging for processes to be run on workstations which are idle or lightly loaded.

(3) A Transparent Network

It would be preferable to be able to run multiple jobs over the network without having to deal with parallel software (such as Express) at all. Ideally, one would like to be able to just set up a standard Unix runfile which submits multiple jobs, with different input and output files, to the *network*, rather than to a particular workstation. Some clever systems software would then distribute these jobs to whichever machines on the network had the smallest load and were not being used interactively. Again, some load balancing software would be required, so that if an interactive session was started on a workstation, the job would be suspended or migrated to another workstation. This would all be transparent to the user, who would just submit jobs to the network and not care where they were run.

Some shared memory parallel machines, such as the Sequent Symmetry and BBN Butterfly allow this kind of programming paradigm. No knowledge of parallel computing or parallel languages is required to run on these machines using job level parallelism. A number of independent Unix processes are submitted to the parallel computer, and each one runs on a separate node. Of course these machines can also be programmed as parallel computers. Ideally a distributed network of workstations could be programmed in the same fashion, using these two paradigms. The new IBM SP-1 machine has this functionality.

4 Parallel Languages

The main problem hindering the use of distributed and parallel computing for scientific applications has been the difficulty of parallelizing sequential codes, and the lack of simple, portable and efficient high level parallel languages. High level languages such as Fortran 90 are now becoming available, and a lot of effort is going into the development of future parallel languages such as High Performance Fortran (HPF) [17] and parallelizing compilers such as Fortran D [18]. The Northeast Parallel Architectures Center (NPAC) at Syracuse University is playing a major role in these efforts, including the development of the Fortran 90D compiler.

We used the Express parallel programming tools for all the work described here. The most appealing features of Express are that it can be used with standard C or Fortran, so sequential codes can be easily utilized, and it is portable. We have also coded our algorithms in CMFortran, a variant of Fortran 90 used on the Connection Machine. We found that programming in CMFortran (or Fortran 90) was much simpler than programming in Express Fortran (or Fortran with message passing). Explicit message passing allows the user the extra control over communications that one would expect from a lower level language, which can help in optimization of parallel algorithms. A simpler higher level language would be preferable, however we must be careful that the implementation of such a language is efficient enough to make using it worthwhile.

Methods for parallelizing simple, regular programs such as the Metropolis algorithm for spin models are straightforward, but tedious and error-prone, in Express Fortran. A parallelizing compiler such as Fortran D should therefore have no problem with algorithms of this kind, even when starting with sequential Fortran 77 code. The implementation of the Metropolis algorithm in Fortran 90 was very simple.

Implementing the irregular, non-local parallel algorithm for the cluster Monte Carlo program was quite difficult in Express Fortran, but surprisingly easy to code in Fortran 90. However this would be a tough problem for a parallelizing compiler starting from sequential Fortran 77 code. One of the difficulties in developing parallelizing compilers such as Fortran D lies in the fact that for many problems, including this one, very different algorithms are required for parallel and sequential machines, and even for different parallel architectures, such as SIMD and MIMD, or coarse and fine grain machines. Also, many modern computational physics applications, such as cluster algorithms, hierarchical N-body simulations, dynamically triangulated random surfaces, and adaptive grid finite element and computational fluid dynamics simulations, use complex, irregular, and dynamic data structures which may be difficult for parallelizing compilers (and even parallel languages) to handle. We expect that substantial user input will be required for efficient parallelization of these problems, especially over distributed computers with high latency and low communications bandwidth.

Portability of parallel software is crucial for the advancement of parallel and distributed computing. Software portability is one of the driving forces behind the High Performance Fortran initiative. Currently there are a number of different proprietary parallel languages, such as CMFortran and C* for the Connection Machine, MP Fortran and MPL for the Maspar, Fortran Plus for the AMT DAP, and different message passing software libraries for the nCUBE, Intel, and Meiko. These cannot be migrated between similar architectures, let alone between SIMD, MIMD, vector, distributed, and sequential machines. We were able to run our Express programs *unchanged* on different MIMD parallel computers and on distributed workstation networks. Such portability is a great asset to computational physicists who would rather spend their time doing physics than porting codes between different computers. Many portable parallel software packages now exist, such as Express, PVM, PICL, Linda, and Strand. The Fortran 90 standard and the development of Fortran D and HPF should further improve the situation and entice more computational scientists to move to parallel and distributed computing.

Setting up the network of workstations to act as a distributed parallel computer was very simple under Express. The user need only run an initialization program and specify the names of the machines to be used. One of the problems with running on the network was that if one of the workstations in the network is loaded with another process, then the efficiency of the distributed computer drops to under 50%, and other processors are under utilized. A solution to this problem would require some kind dynamic load checking and load balancing by the systems software, perhaps by migrating processes to other machines which are idle (or at least less loaded).

The main problem with Express is that it uses a fairly low-level implementation of parallelism, using explicit message passing. Writing distributed memory MIMD parallel programs can be extremely difficult, and debugging such programs can be even harder. We have seen that producing code which works efficiently, especially on a loosely coupled distributed network, requires an additional level of effort. Optimization usually requires carefully adjusting the communications strategy, and sometimes changing the algorithm. This tends to be very time consuming and error-prone. Clearly a higher level parallel language such as HPF is required, but it is also clear that developing and implementing such a language to work efficiently for a wide class of problems is a very difficult task.

5 Discussion and Conclusions

The performances we obtained on the workstation network using standard Ethernet were, as expected, not nearly as good as for distributed memory parallel computers such as the nCUBE. However we were able to use the large memory and processing power of each node to get reasonable performance for very large physical systems distributed over small numbers of nodes, where the amount of data per node is large.

To exploit parallel computing systems based on networks of workstations, it is necessary to construct algorithms that take advantage of very coarse grain parallelism and the large processor memory, minimize latency, and maximize the ratio of calculation to communication. This is much more crucial than for parallel computers. For instance the simplest parallel implementation of the Metropolis algorithm, which works very well on fine grained machines like the CM-2, and passably well on the nCUBE/2, performs so poorly on the Ethernet LAN that it runs slower using more than one workstation. However an improved coarse grained algorithm gives good speed-ups for this network. The cluster applications performed well on small numbers of processors, since a lot of effort had already gone into optimizing the code for large numbers of processors on massively parallel MIMD computers such as the nCUBE/2. Other applications, such as the simple independent parallelism of the Wolff code, required no extra effort to gain maximum benefit from using multiple workstations across a network.

Since we only have a very small number of workstations in our network, it is difficult to assess the scalability of the hardware. From results we have obtained for scaling to larger numbers of nodes on the nCUBE/2, it is unlikely that our applications will be scalable to large numbers of workstations using standard Ethernet connections, mainly due to high latency. However the prototype switch which we have used gave efficiencies comparable to the nCUBE/2, even though the nCUBE/2 has much slower processors. This implies that communications bandwidth, and more importantly the latency, of the switch is comparable to or better than a parallel MIMD computer. Such as switch should thus provide scalability on a par with MIMD computers, and thus allow the use of a large number of connected workstations to be used as a parallel computer on a single problem.

Developing parallel programs for many regular scientific applications is rather tedious, but straightforward, with current parallel software tools. The introduction of Fortran 90, Fortran D, and HPF compilers will greatly simplify this task. We have seen that useful parallel algorithms can be developed even for complex, irregular problems requiring non-local data access, such as the cluster Monte Carlo programs. These problems are much more difficult to implement with current languages and compilers, however higher level parallel languages should improve this situation.

Our results suggest that a cluster of workstations used as a parallel computer can be an economical way of providing a high performance computing resource. However improvements in hardware (fast switches and high bandwidth communications), systems software (making the network transparent and easily accessible without unduly impacting interactive users), and parallel software (portable high level languages such as HPF) are necessary for easy and efficient use of such a system. All of these improvements should be readily implementable in the near future.

Acknowledgements

I would like to thank John Apostolakis, Clive Baillie, Taitin Chen, Geoffrey Fox, Leping Han, Enzo Marinari and the NPAC systems staff for discussions and programming support.

This work was supported in part by the Center for Research on Parallel Computation with NSF cooperative agreement No. CCR-9120008, and a grant from the IBM Corporation.

This paper discusses a prototype which has not been announced as a product, and which may never be announced or made generally available as a product. Any performance data contained in this document was determined in a specific controlled environment, and therefore, the results which may be obtained in other operating environments may vary significantly.

References

[1] T. Chen *et al.*, "A Prototype Switch for Scalable High-Performance Distributed Computing", NPAC Technical Report SCCS-392 (1992).

[2] "Express Reference Manual", ParaSoft Corporation, 2500 E. Foothill Blvd., Pasadena, CA 91107 (1988).

[3] K. Binder ed., *Monte Carlo Methods in Statistical Physics*, (Springer-Verlag, Berlin, 1986).

[4] H. Gould and J. Tobochnik, *An Introduction to Computer Simulation Methods*, (Addison-Wesley, Reading, Mass., 1988).

[5] N. Metropolis *et al.*, *J. Chem. Phys.* **21**, 1087 (1953).

[6] R.H. Swendsen and J.-S. Wang, *Phys. Rev. Lett.* **58**, 86 (1987).

[7] U. Wolff, *Phys. Rev. Lett.* **62**, 361 (1989).

[8] G. Fox *et al.*, *Solving Problems on Concurrent Processors, Vol. 1*, (Prentice-Hall, Englewood Cliffs, 1988).

[9] E. M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice* (Prentice-Hall, Englewood Cliffs, N.J., 1977); E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, (Computer Science Press, Rockville, Maryland, 1978).

[10] C.F. Baillie and P.D. Coddington, *Concurrency: Practice and Experience* **3**, 129 (1991).

[11] J. Apostolakis, P. Coddington and E. Marinari, *Europhys. Lett.* **17**, 189 (1992).

[12] J. Apostolakis, P. Coddington and E. Marinari, "New SIMD Algorithms for Cluster Labeling on Parallel Computers", NPAC Technical Report SCCS-279 (1992).

[13] P.D. Coddington and C.F. Baillie, *Phys. Rev.* **B 43**, 10617 (1991).

[14] P.D. Coddington and C.F. Baillie, *Phys. Rev. Lett.* **68**, 962 (1992).

[15] C.F. Baillie and R.D. Williams, "Numerical Simulations of Dynamically Triangulated Random Surfaces on Parallel Computers with 100% Speedup", Proc. of the 5th Distributed Memory Computing Conference, Charleston (April 1990), eds. D.W. Walker and Q.F. Stout (IEEE Computer Society Press, Los Alamitos, California, 1990).

[16] M. Bowick, P. Coddington, L. Han, G. Harris and E. Marinari, "The Phase Diagram of Fluid Random Surfaces with Extrinsic Curvature", NPAC Technical Report SCCS-357 (1992), to be published in *Nucl. Phys.* **B**.

[17] The High Performance Fortran Forum, "High Performance Fortran Language Specification", Center for Research in Parallel Computing Technical Report CRPC-TR92225.

[18] G.C. Fox *et al.*, "Fortran D Language Specifications", NPAC Technical Report SCCS-42C (1990); A. Choudhary *et al.*, "Compiling Fortran 77D and 90D for MIMD Distributed Memory Machines", NPAC Technical Report SCCS-251 (1992).