Figure Captions

Figure 1: Speed-up of our MIMD parallel component labeling algorithm for the nCUBE/10.

Figure 2: Speed-ups of the parallel spin model algorithms as a function of system size for a network of 4 IBM RS/6000-340s.

Figure 3: The time per iteration for the N-body program on an IBM RS/6000-340 network for different numbers of particles and processors.

Figure 4: The speed-up for the N-body program on an IBM RS/6000-340 network for different numbers of particles and processors.

21. M. Bowick, P. Coddington, L. Han, G. Harris and E. Marinari, "The Phase Diagram of Fluid Random Surfaces with Extrinsic Curvature", Syracuse University preprint SCCS-357, submitted to *Nucl. Phys.* **B**.
22. B. Brügmann and E. Marinari, "4d Simplicial Quantum Gravity with a Non-Trivial Measure", Syracuse University preprint SCCS-361, SU-GP-92/9-2.
23. P. Coddington, L. Han and E. Marinari, work in progress.
24. J. Barnes and P. Hut, *Nature* **324**, 446 (1986).
25. J. Salmon, "Parallel Hierarchical N-body Methods", Center for Research on Parallel Computation Technical Report 90-14, Caltech, December 1990; J. Salmon and M. Warren, "Skeletons from the Treecode Closet", submitted to *J. Comp. Phys.*.
26. 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).
27. G. Cheng, K. Mills, and G. Fox, "An Interactive Visualization Environment for Financial Modeling on Heterogeneous Computing Systems", to be published in *The Sixth SIAM Conference in Parallel Processing for Scientific Computing* (1992).
28. S. Ranka *et al.*, in preparation.
29. P. Coddington, D. Edelsohn, L. Han, and E. Marinari, in preparation.

communications), systems software (making the network transparent and easily accessible to the user), and parallel software (portable higher 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.

Acknowledgments

This work is the result of a collaboration between the Northeast Parallel Architectures Center (NPAC), which provided hardware and software support and expertise, and the computational physics group in the Physics Department at Syracuse University. 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.

1. The High Performance Fortran Forum, "High Performance Fortran Language Specification", Center for Research in Parallel Computing Technical Report CRPC-TR92225.
2. 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); A. Choudhary, G.C. Fox and A.G. Mohamed, "Fortran D Compiler for MIMD Machines", NPAC Technical Report SCCS-275 (1992).
3. A.G. Mohamed *et al.*, "Application Benchmark Set for Fortran-D and High Performance Fortran", NPAC Technical Report SCCS-327 (1992).
4. "Express Reference Manual", ParaSoft Corporation, 2500 E. Foothill Blvd., Pasadena, CA 91107 (1988).
5. For a review of spin models and Monte Carlo algorithms, see *Monte Carlo Methods in Statistical Physics*, Ed. K. Binder (Springer-Verlag, Berlin, 1986).
6. N. Metropolis *et al.*, *J. Chem. Phys.* **21**, 1087 (1953).
7. R.H. Swendsen and J.-S. Wang, *Phys. Rev. Lett.* **58**, 86 (1987).
8. U. Wolff, *Phys. Rev. Lett.* **62**, 361 (1989).
9. J.-S. Wang and R.H. Swendsen, *Phys. Rev.* **B 38**, 4840 (1988).
10. G. Fox *et al.*, *Solving Problems on Concurrent Processors, Vol. 1*, (Prentice-Hall, Englewood Cliffs, 1988).
11. 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).
12. C.F. Baillie and P.D. Coddington, *Concurrency: Practice and Experience* **3**, 129 (1991).
13. J. Apostolakis, P. Coddington and E. Marinari, *Europhys. Lett.* **17**, 189 (1992).
14. J. Apostolakis, P. Coddington and E. Marinari, "New SIMD Algorithms for Cluster Labeling on Parallel Computers", Syracuse preprint SCCS-279, submitted to *Int. J. Mod. Phys. C.*
15. J. Apostolakis, M. Bernaschi, P. Coddington and E. Marinari, work in progress.
16. P.D. Coddington and C.F. Baillie, *Phys. Rev.* **B 43**, 10617 (1991).
17. P.D. Coddington and C.F. Baillie, *Phys. Rev. Lett.* **68**, 962 (1992).
18. F. David, in *Two Dimensional Quantum Gravity and Random Surfaces*, eds. D.J. Gross, T. Piran, and S. Weinberg, (World Scientific, Singapore, 1992).
19. V. A. Kazakov, *Phys. Lett.* **150B**, 282 (1985); F. David, *Nucl. Phys.* **B257**, 45 (1985); J. Ambjørn, B. Durhuus and J. Frölich, *Nucl. Phys.* **B257**, 433 (1985).
20. *Statistical Mechanics of Membranes and Surfaces*, eds. D. Nelson, T. Piran, and S. Weinberg, (World Scientific, Singapore, 1989).

integration systems will prove to be very important for future large scale simulations requiring high performance computing.

8. Discussion and Conclusions

The performances we obtained on the distributed network were, as expected, not as good as for tightly coupled parallel computers. 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 (4-16) numbers of nodes, where the amount of data per node is large. Note that a network of 16 IBM RS/6000-340 machines represents a 240 Mflop computing resource.

To exploit parallel computing systems based on networks of workstations, it is necessary to construct algorithms that take advantage of 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 tightly coupled parallel computers. For instance the simplest parallel implementation of the Metropolis algorithm, which works very well on fine grained machines like the CM-2, performs extremely poorly on the distributed network. However an improved parallel algorithm gives good speed-ups for this network. The cluster and N-body applications performed well on small numbers of processors, since a great deal of effort had already gone into optimizing the code for large numbers of processors on massively parallel MIMD computers such as the nCUBE/2 and Intel Delta. Other applications, such as the simple independent parallelism of the strings code, required no extra effort to gain maximum benefit from using multiple workstations across a network.

From our results it seems that many of our computational physics applications are unlikely to be scalable to large numbers of workstations using standard Ethernet connections. However much better network connections could change this conclusion. The important task is to provide a communication medium with low latency and high bandwidth. We have therefore been experimenting with connecting IBM RS/6000 workstations with a high performance prototype V7 switch provided by IBM [28, 29]. This is a copper wired switch based on 8x8 prototype crossbars, with an adapter which connects directly to microchannel of the RS/6000. Replacing a standard 10 Mbit/sec Ethernet network with a fast switch of this kind, or even with 100 Mbit/sec optical FDDI connections, will greatly improve the performance and usability of a large distributed network.

Developing parallel programs for many regular scientific programs is straightforward with current parallel tools. The introduction of Fortran 90, Fortran D, and HPF compilers will make this even simpler. We have seen that useful parallel algorithms can be developed even for complex, irregular problems requiring nonlocal data access and dynamic data structures, such as the cluster Monte Carlo and N-body programs. These problems are 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 distributed parallel computer can be an economical way of providing a high performance computing resource. However improvements in hardware (fast switches and high bandwidth

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. One would like to be able to just add a -O flag to a parallelizing compiler, but we are still very far from that ideal. 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 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.

7. Graphics and Visualization

*** PLOTIX??? ***

Graphics and visualization are increasingly becoming indispensable tools for analyzing and understanding large data sets and complex physical problems. Many of our physics applications involve complex and irregular data structures, and visualization tools are extremely useful for analyzing and understanding the physical system and the algorithms used to simulate the system. Graphical tools can also be a useful aid in debugging these algorithms, especially on parallel computers where debugging can be very difficult.

Dynamically triangulated random surfaces consisting of hundreds of two dimensional trianglar cells embedded in three dimensions are very difficult to picture without some sophisticated visualization tools. To help with our DTRS studies we used the Advanced Visualization System (AVS), since is particularly powerful at handling 3D images, especially for unstructured data. It also has a very simple and powerful interactive graphical user interface, which allows a complex visualization program to be constructed by connecting system modules for rendering and image processing with user modules for computation. AVS also allows the user to interactively adjust lighting, shading, coloring, and orientation of the image. Another beneficial feature is that it is available for many different systems, including Silicon Graphics, Sun, DEC and IBM workstations.

The current version of AVS provides a parallel module execution function. We are therefore able to visualize simulations running on different machines in the network in parallel. AVS also enables the user to incorporate input from modules running concurrently on many different machines, from networks of workstations to massively parallel supercomputers. This kind of dataflow model can be used for linking separate parts of a large simulation, with for example computational and database modules running on massively parallel supercomputers, data analysis and perhaps graphical rendering being done on a workstation network, with the results being passed to a fast graphics engine for display.

In the field of computational finance, NPAC is using an AVS framework to integrate visual and scientific computing in a heterogeneous computing environment with a set of stock option pricing models [27]. Computational modules running on massively parallel computers are coupled with workstation and network-based visualization and interactive control modules. AVS has also been used to support the linking of such modules in the NPSS project at NASA-Lewis, which is a multidisciplinary propulsion design system. We believe that AVS and similar software

compare the performance of this code using Fortran 90D to the Express Fortran code running on the same distributed MIMD machine.

One of the major difficulties in developing parallelizing compilers such as Fortran D lies in the fact that different algorithms are often required for parallel and sequential machines, and also for different parallel architectures, such as SIMD and MIMD, or coarse and fine grain machines. Many modern computational physics applications, such as cluster algorithms, hierarchical N-body simulations, and 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 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.

We believe that 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 the nCUBE, Intel, and Meiko message passing software libraries. These cannot be migrated between similar architectures, let alone between SIMD, MIMD, vector, distributed, and sequential machines.

The ability to run our Express codes *unchanged* on different MIMD parallel computers and on distributed workstation networks is a great asset to computational physicists who would rather spend their time time doing physics than porting codes between different parallel languages. 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.

We used the Express parallel programming tools for the majority of our work on distributed computing. The most appealing features of Express are that it can be used with standard C or Fortran, so sequential codes can be utilized, and it is portable. Codes which we had developed on nCUBE or Intel parallel computers ran unchanged on the network of IBM workstations. The only things which required adjustment were the Makefile and the directory path for finding the correct Express libraries.

Setting up the network of workstations to act as a distributed parallel computer was very simple under Express. The user need only to run an initialization program and specify the names of the machines to be used. Ideally this would not be necessary, and the systems software would automatically search the local network for unloaded machines to be utilized.

One of the problems with running under Express was that if one of the workstations in the network was loaded with other jobs, then this affected the performance of the whole network in running the parallel program. A solution to this problem would presumably require some kind dynamic load checking and load balancing by the Express 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

an interactive session was started on a workstation, the job would be 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.

6. 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) [1] and parallelizing compilers such as Fortran D [2]. 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.

A related project at NPAC is the development of a language benchmark suite which is being used to aid in the design, implementation, testing and validation of the Fortran D compilers and HPF language standard [3]. We have implemented a set of our physics application codes in a number of different languages including Fortran, C, Fortran and C with message passing, CMFortran (a variant of Fortran 90), and C*, in order to compare the efficiency, ease of use, and range of applicability of higher level languages such as Fortran 90 with lower level languages such as Fortran with message passing. These programs have also been added to the NPAC language benchmark suite.

During the course of this work it became clear that programming in Fortran 90 was much simpler than programming in Express Fortran (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. However in most cases this extra flexibility is not necessary, or of marginal benefit, and a simpler higher level language would be preferable.

Secondly, we found methods for parallelizing simple, regular programs such as the Metropolis algorithm for spin models to be straightforward, but very 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 these problems in CMFortran 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 in CM-Fortran. The problem with this algorithm was its inefficiency, which was due more to the CM-2 machine architecture than the compiler. It will be very interesting to

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 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 a method.

(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, since 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. If results are output frequently, this communication may cause unacceptable performance degradation. Synchronization overhead can be reduced by altering the code slightly to buffer data on each processor and thus output larger blocks of data less frequently. This was not required for the strings code, which performed at virtually 100% efficiency when distributed across the workstation network, even for the smallest system sizes used.

Synchronization during data output can also 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 jobs. 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.

We still have the problem of "load balancing", in the sense that some machines in the network may be loaded with other jobs, and thus take much longer to finish their work. It would be helpful if there were some kind of dynamic load balancing functionality, whereby if a machine became loaded with another job, then the process being run under Express would be migrated to another unloaded machine in the network, if one was available.

(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 submitted 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

UDP/IP protocol, much like ParaSoft's Cubix/NetExpress. This entire development system was ported to the RS/6000 running AIX with relatively little difficulty. SRV/NetCrOS provides better performance and less problematic interactions with the AIX operating system than ParaSoft's Cubix/NetExpress, but provides more limited facilities and can ignore certain compatibility issues. NetCrOS is basically a more highly optimized and specialized subset of NetExpress.

*** NetExpress or Express?? ***

The calculation itself is quite floating-point intensive which utilizes the IBM RISC architecture very effectively. Once the intra-timestep computation dominates the communication (both active message transit and idle waiting time due to external load imbalances such as other active processes on the system), the speed-ups closely approach the theoretical maximum, i.e. 100,000 particles on 16 processors achieves a speed-up of 15.2 or 95% of the ideal speed-up. Figs. 3 and 4 show the time per iteration and the speed-up for a variety of particle and processor configurations.

5. Distributed Computing Using Independent Parallelism

We have run both the DTRS strings program and the Wolff cluster algorithm program on MIMD parallel computers and on networks of workstations by using the trivial parallelism of running independent simulations on each processor, using different parameter values for each simulation. 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.

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.

Independent parallelism is especially applicable to Monte Carlo simulation, which can be parallelized in this way even without varying the parameter value. This is because Monte Carlo simulation 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 simple approach has been used very successfully for simulations of random surfaces, spin models, and many other statistical mechanics problems [16, 17, 21, 26].

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

state, which can be any configuration, to an uncorrelated state representative of the system being simulated, i.e. with a particular temperature or system parameter.

To use independent parallelism, each processor has to run for at least the equilibration time before it can start producing usable data, so this is effectively wasted time. For large systems, this time may be so great (due to critical slowing down) that in practice it is not feasible to equilibrate the system on a single processor. However a system being simulated in parallel over many processors may be equilibrated and produce useful data in a reasonable time period. For this reason, we are currently working on parallelizing our random surface code [23].

Since this application involves a dynamic irregular mesh, we expect that SIMD architectures would fare poorly on this problem, and efficient parallelization will be difficult even on MIMD machines, and especially on a workstation network. This is because neighbors of a node, which will originally reside on the same processor or a neighboring processor using standard domain decomposition, will change throughout the simulation so that they may end up on distant processors, thus increasing the communication cost. Some kind of dynamic load balancing will therefore be required to implement an efficient parallel DTRS simulation on a distributed parallel computer.

4. N-body Simulations

Large, inhomogeneous N-body problems, such as the gravitational formation of structure in the Universe, continue to pose a significant challenge for computational simulation. These simulations require evolving regions of various models of a realistic, hypothetical universe from the initial radiation-dominated era soon after the Big Bang to the present era. The degree to which a simulation agrees with observations of the Universe supports or rejects the model. The structures formed by the mass elements in the simulation must be well resolved in order to obtain meaningful, low-noise statistics, so that a conglomeration of particles assumed to represent a galaxy should have hundreds or thousands of particles, not tens. The simulations therefore need to evolve $10^5 - 10^6$ particles through a very large number of timesteps, representing the age of the Universe.

N-body simulations also play a crucial role in the study of the evolution of galaxies. Some of the observed internal structures of galaxies are due to collisions between galaxies. The fine detail which must be modeled in these simulations also requires of the order of 1 million particles in a galaxy.

Recently hierarchical methods such as the Barnes-Hut algorithm [24] have been introduced which reduce the time required for N-body simulations from $O(N^2)$ for standard algorithms to $O(N \log N)$ or even $O(N)$. While these new algorithms make million body simulations tractable, it still requires significant CPU time for the multiple, extended simulations necessary for statistically accurate results. Massively parallel MIMD distributed memory computers are an obvious choice for this supercomputing-class application.

We have ported John Salmon and Mike Warren's parallel implementation of the Barnes-Hut algorithm [25] to a cluster of IBM RS/6000 workstations using Salmon and Warren's optimized message passing communication library (SRV/NetCrOS). The communication uses the standard workstation network connection and the

- The ratio of computation to communication is high.
- The computation required for each replica is suited for a single scalar processor.
- The communication is local.
- The latency is low, since there are only two communication steps per iteration (passing the whole replica before and after each update).

We expected our parallel algorithm for the replica Monte Carlo method to perform well on a distributed network, however we found that we obtained speed-ups of less than 1 on a 4 processor network. It is unclear why this is the case, however we have spent very little time optimizing this code, and further optimization effort would surely be rewarded with greatly improved performance.

This result is in accordance with our belief that while it may be possible for a compiler to parallelize many computational physics algorithms (as would surely be the case here, since the parallelization was very simple), it is a much more difficult problem to parallelize the program *efficiently*. If the program runs slower on a multiprocessor machine that on a single processor, then the whole motivation for parallelism is lost.

*** NEED TO CHECK PARALLEL CODE AND GET BETTER RESULTS, OR FIGURE OUT WHY WE GET SUCH BAD RESULTS!!! ***

3. Dynamically Triangulated Random Surfaces

String theories and quantum gravity can be simulated using models based on dynamically triangulated random surfaces [18, 19]. For example, the worldsheet of a string can be represented as a triangular mesh. Calculations involve integrating over all possible worldsheets, which can be approximated by a sum over a large number of different meshes. These are obtained by making random changes to the positions and connections of nodes in the mesh throughout the calculation using standard Monte Carlo techniques, resulting in a dynamically triangulated random surface (DTRS).

DTRS models can also be used to simulate systems which have real fluctuating surfaces, such as biological and chemical membranes, lipid bilayers, microemulsions, and phase boundaries [18, 20]. We are currently doing Monte Carlo simulations to study the phase structure of DTRS models of string theories and quantum gravity [21, 22].

We have used the simple method of parallelism over independent simulations (see section) to run the random surface simulations on networks of workstations and on MIMD parallel computers such as the Intel Touchstone Delta and the nCUBE/2. This works very well for relatively small systems, however for large systems we strike a snag related to the problem of critical slowing down, which we have seen is a major problem for Monte Carlo simulations of spin models near phase transitions. Random surface simulations also suffer badly from critical slowing down. A consequence of this is that the *equilibration time* grows rapidly with the size of the system. This is the time required for the Monte Carlo algorithm to move the system from the initial

to communication ratio is quite high. Most of the communication occurs in the local propagation of labels of sub-clusters to neighboring processors, and for small numbers of processors very few local propagation steps are required. This algorithm has also been considerably optimized for coarse grain parallel computers, by minimizing both the amount of communication and the number of communication calls required, in order to reduce the latency, which is very large for networks of workstations.

The speed-up we obtained on the RS/6000 network is quite good, and similar to that achieved for the much simpler and more regular parallel Metropolis algorithm (see Fig. 2). However we expect the parallel Metropolis algorithm to scale much better to larger networks, as is the case for larger numbers of nodes on tightly coupled parallel computers.

*** COMPARE WITH RESULTS ON nCUBE/2 AND SUN NETWORK ****

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 Swendsen-Wang cluster algorithm [7] we can use parallel or distributed computers to study larger systems, using the parallel algorithms discussed above. However the cluster algorithm of Wolff [8], which grows a single cluster, cannot be efficiently parallelized or vectorized. A network of high performance RISC workstations is the most suitable system for simulations using this algorithm. Workstations have enough memory and power to handle quite large systems, and the IBM RS/6000-550 is almost as fast as a CRAY X-MP for this problem.

Although this problem 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, the temperature, model parameters (such as the number of allowed states of the system in the Potts model), or with different random numbers to increase the Monte Carlo statistics and thus decrease the statistical uncertainty in the results. We discuss this method of independent parallelism further in section . 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 Potts spin models [16, 17].

2.3. Replica Monte Carlo

The replica Monte Carlo algorithm [9] uses an approach similar to the Swendsen-Wang cluster algorithm, and has proven to be very effective for certain kinds of frustrated spin models such as spin glasses. The replica method uses a number of interacting copies (replicas) of the lattice, each at a different temperature. The spins for adjacent replicas (those closest in temperature) are combined, and a cluster update is done on this combined system. The individual replicas are then updated using the new values of the combined system.

This algorithm can be parallelized most effectively by putting each replica on a different processor. It therefore seems an ideal candidate for parallelism over a small network of powerful workstations, since

- The number of replicas required is generally fairly small (of order 10).

(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 [12]. 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 Swendsen-Wang cluster update algorithm, for both MIMD and SIMD machines [12, 13, 14].

The problem of identifying clusters in spin and percolation models provides a difficult test for parallel component labeling algorithms. Here the clusters to be labeled are very irregular in shape (possibly even fractal), come in all different sizes (generally including a very large cluster which covers a substantial part of the lattice), and have complicated topology (they can contain holes of all sizes). This irregular size of the clusters means that load balancing can be a problem, and a lot of inter-processor communication is required to label the largest cluster. The irregularity of the clusters also means that this problem is not well suited to a SIMD machine [14].

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, so different processors can be executing different instructions in parallel. 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.

Once each processor has labeled the clusters in its particular section of the lattice, we need a method for combining the local labels of these sub-clusters to create global labels for the full clusters. We found that this could be done easily and fairly efficiently by just using the naive technique of local label propagation, except that now only sites on the edge of processors pass labels to their connected neighbors on the edge of neighboring processors [12]. This causes a merging of sub-clusters across processors. As long as the ratio of sites to processors is relatively large, this will be fairly efficient (see Fig. 1).

For a parallel machine consisting of a modest number of very coarse-grained processors, such as a small network of workstations, local propagation of the edge labels as described above works very well. However for massively parallel MIMD machines with hundreds of nodes, this method scales quite poorly (see Fig. 1). Adding the non-local label propagation techniques which we developed for SIMD machines should greatly improve the performance of our MIMD component labeling algorithm for large numbers of processors. This is work in progress [15].

The MIMD parallel algorithm which we developed for the Swendsen-Wang cluster algorithm [12] works well on small networks of workstations, since the calculation

The only difference is that the function which we call to find the neighboring spin value handles periodic boundary conditions for the sequential code, while in the parallel code it will perform message passing if the neighboring site is on another processor. This regular local communication is particularly simple in CMFortran or Fortran 90, with just a call to a periodic SHIFT operation.

For small to medium grain massively parallel processors such as the CM-2 or nCUBE, with small latencies and fast local communication, this very simple parallel algorithm works very well. However this will not be efficient 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 values of a single site). In this case we need to optimize the algorithm more carefully, so that we pass a smaller number of messages each containing a larger amount of data. This can be done by using the checkerboard algorithm for the sub-lattice 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 black or white update. This greatly reduces the latency, and since we are on a coarse grain system we have a large calculation-to-communication ratio (basically the ratio of the volume of the sub-lattice to its perimeter), so that we again obtain good efficiencies.

As expected, the naive parallel Metropolis algorithm, where any message passing required is done for each site, was very inefficient on the distributed network due to the high latencies and the small amount of calculation required between each communication. This algorithm ran slower on four RS/6000's than the sequential algorithm running on a single machine (see Fig. 2). Note that the timings shown in Fig. 2 have quite large degree of uncertainty, since the workstations are time-shared with other processes and different runs will give different times. *** GET AVERAGES AND ERRORS *** 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 naive parallel algorithm on the RS/6000 network scaled roughly as the length of the lattice, and thus was was totally dominated by communication.

The optimized checkerboard method, where all edge spins are communicated at once, is much less impacted by the high latency of the network, and consequently gave substantially improved performance (see Fig. 2). We expect that this performance will be scalable to larger networks as long as the amount of data per processor remains constant.

*** COMPARE WITH RESULTS ON nCUBE/2 AND SUN NETWORK ****

2.2. Cluster Algorithms

*** TRIM THIS A BIT ***

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 [11]. 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

Metropolis algorithm [5, 6]. 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 [7, 8, 9]. 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.

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 should be well handled by any parallelizing compiler, and is easy to express in a language such as Fortran 90. The cluster algorithms, on the other hand, are very 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. They also test the architecture of parallel machines, since they require a lot of non-local data communication.

2.1. 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 [10]. 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. However 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 partitioning of the lattice, in which all the white sites can be updated in parallel, and then all the black sites. 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 is required of all Monte Carlo algorithms [5].

This checkerboard partitioning method is also used for solving differential equations, as a compromise between the completely parallel Jacobi algorithm, for which all the points are updated at once using the old values of the neighboring points, and the purely sequential Gauss-Seidel algorithm, in which the sites are updated in order one at a time, thus using a mixture of old and new values for neighboring points. The Gauss-Seidel algorithm has better convergence properties, but is sequential. The checkerboard approach is an efficient parallel algorithm like the Jacobi method, but with improved convergence properties like Gauss-Seidel [10].

Since the algorithm is regular we do not have to worry about load balancing, and writing a parallel Metropolis algorithm is trivial. 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 and the resulting code on each node will look almost exactly like the sequential code.

supercomputers, which currently represent the only plausible means of continuing the rapid increase in computational power available to scientists and engineers over the past 40 years.

The computational physics applications presented here provide an interesting spectrum of computationally challenging algorithms, including N-body simulation using hierarchical tree-codes, Monte Carlo simulation of spin models using both local and non-local algorithms, dynamical random surface simulation of string theories and quantum gravity, and the application of new Monte Carlo algorithms to general optimization problems. These algorithms span the range from regular and local (and therefore easy to parallelize) to adaptive, irregular, and multi-scale (and much more difficult to parallelize).

We have developed and studied a range of parallel algorithms for these problems on different parallel architectures. In particular we investigated the use of a network of workstations for distributed computing, and examined 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 our research consist of 5 IBM RS/6000 model 340 workstations. *** and 8? 16?? Sun SPARC1+ workstations. *** We can view these groups of workstations connected via a local area network as a powerful parallel computer. The communications network is standard Ethernet, which is much slower (10 Mbits/sec) than the internal communications of a tightly coupled parallel computer (*** Mbits/sec for the nCUBE/2). To use a workstation network as a distributed parallel computer, algorithms must be found to exploit the coarse grained parallelism and large node memory, in order to overcome the relatively high latency and low bandwidth inherent in a network based system. *** Numbers for latency of network communications? Of Express?? ***

The software we have used to make the network behave as a parallel computer is Express, from ParaSoft Inc. [4] Express is the result of research by Fox's group at Caltech, and is a powerful and portable message passing environment which has tools for domain decomposition, parallel I/O, graphics, debugging and profiling. It can be used on most MIMD parallel computers, including those from nCUBE, Meiko and Intel, as well as networked workstations.

We report below our experiences for each computational physics application. We briefly describe each application and the structure of the associated computational algorithms, and discuss parallel algorithms and the implementation of these algorithms on a distributed network of workstations. We then discuss general issues such as parallel languages and software environments, using graphics on workstation networks, and other computational issues relevant to the use of distributed workstations as parallel computers.

2. Monte Carlo Simulation of Spin Models

Phase transitions in magnets can generally be described very well by simple models of interacting spins on a lattice, such as the Ising model [5]. Numerical simulations of these models are mainly done using Monte Carlo methods such as the

USING NETWORKED WORKSTATIONS AS PARALLEL COMPUTERS FOR COMPUTATIONAL PHYSICS

P. CODDINGTON[1,2], D. EDELSOHN[1,3], L. HAN[1], and E. MARINARI[1,4]

[1] *Department of Physics, Syracuse University, Syracuse NY 13244 U.S.A.*

[2] *Northeast Parallel Architectures Center, 111 College Place, Syracuse University, Syracuse NY 13244 U.S.A.*

[3] *IBM Corporation, T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, U.S.A.*

[4] *Dipartimento di Fisica, Università di Roma* Tor Vergata, *Via E. Carnevale, 00173 Roma, Italy.*

December 4, 1992

ABSTRACT

We have implemented a set of computational physics codes on a network of workstations with standard ethernet connections. The algorithms range from simple, local, and regular to complex, non-local, dynamic, and irregular. We describe our experiences in using this network for distributed and parallel computing, and compare the performance of our programs on the distributed network and on a tightly coupled distributed memory MIMD parallel computer. We discuss ideas for improving the software and hardware to make distributed parallel computing on workstation networks more easily usable for computational physics.

. Inrodcion

Many companies, academic institutions, and industrial and government 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 and relatively small cost of such networks, there is a huge resource of unused cycles available for doing high performance computing at a relatively small cost.

The problems hindering the effective use of these workstation networks as powerful distributed computers are twofold – hardware and software. Due to the high latency and small bandwidth of Ethernet LANs, the type of applications which can be run effectively on these networks is relatively 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). 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. Of course the complexities of parallel programming also limit the use of tightly coupled massively parallel