U.S. DEPARTMENT OF COMMERCE

National Institute of Standards and Technology

# Information Technology Laboratory

## High Performance Systems and Services Division

DSTP

DISTRIBUTED SYSTEMS
TECHNOLOGY PROJECT

NISTIR 6148

# Evaluation of Applications on a Loosely-Coupled Cluster

Wayne Salamon
Alan Mink
Mike Indovina
Michel Courson

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

April 1998

NISTIR 6148

# Evaluation of Applications on a Loosely-Coupled Cluster

**Wayne Salamon**
**Alan Mink**
**Mike Indovina**
**Michel Courson**

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Information Technology Laboratory
High Performance Systems and Services Division
Bldg. 220, Room B124
Gaithersburg, MD 20899-0001

April 1998

# Evaluation of Applications on a Loosely-Coupled Cluster [*]

Wayne Salamon, Alan Mink, Mike Indovina and Michel Courson

High Performance Systems and Services Division

National Institute of Standards and Technology [†]

{wsalamon,amink,mindovina,mcourson}@nist.gov

## Abstract

NIST is building a distributed testbed of heterogeneous workstations connected via an Asynchronous Transfer Mode (ATM) network. Currently, the ATM network cluster consists of Sun, Silicon Graphics, and Intel-based workstations. The purpose of the ATM cluster testbed is twofold, one is production and the other is research. The production focus is concerned with evaluating the benefit of bringing ATM to the desktop and determining the scalability and viability of such an environment for some of the NIST high performance computation workload. The research focus is concerned with integrating performance measurement for application tuning and developing light weight models that can be used to dynamically steer applications based on real-time measurements. Our initial efforts of porting and tuning parallel codes in this distributed environment are discussed.

**Key words:** Computers, Cluster Computing, Distributed Computing, Networks, ATM, Performance Measurement;

## 1 INTRODUCTION

As the availability of monolithic supercomputers decreases and the performance of relatively inexpensive commodity microprocessor chips for workstations and PCs increases, the high

[*] This work was partially sponsored by the Defense Advanced Research Projects Agency

[†] Contributions of the National Institute of Standards and Technology (NIST) are not subject to copyright. Certain commercial products are identified here in order to document our experiments. Identification of such products does not imply endorsement by NIST.

performance computing community has embraced parallelism via distributed memory in the form of cluster/networked computing. These clusters, running Parallel Virtual Machine (PVM) or Message Passing Interface (MPI) message passing environments, may be highly integrated, as in the IBM SP2, Convex Exemplar, or Cray T3 series, or more loosely coupled as in Network of Workstations (NOW) [1, 2] and Cluster of Workstations (COW). The highly integrated cluster machines incur a larger up-front capital expense, but their maintenance costs are usually 20%, or less, per year. Since they are sold as a system, they are supported by the manufacturer and have the "look-and-feel" of a unified system with a large number of the necessary system wide utilities. The loosely coupled clusters represent more of the do-it-yourself genre. They incur a lower up-front capital expense, but their maintenance costs are usually much higher in terms of knowledgeable internal staff. There is no system level support for such systems and they lack the "look-and-feel" of a unified system. Thus, based on life cycle costs, it is not clear which cluster is less expensive. Nevertheless, the low initial cost of NOWs, in trying to recover the unused cycles of existing networked machines, and the promise of significant performance [1, 3] is compelling.

In an effort to investigate the potential of NOWs and COWs, NIST is building a distributed testbed of heterogeneous workstations connected via an Asynchronous Transfer Mode (ATM) network[4]. Currently, the ATM network cluster consists of 20 Sun, Silicon Graphics, and Intel-based workstations, soon to grow to 60 and beyond.

The purpose of the NIST ATM cluster testbed is twofold, one is production and the other is research. The production focus is concerned with evaluating the benefit of bringing ATM to the desktop and determining the scalability and viability of such an environment for some of the NIST high performance computation workload. The research focus is concerned with integrating performance measurement for application tuning and developing light weight models that can be used to dynamically steer applications based on real-time measurements.

Performance measurement and tuning within this testbed will use a variety of tools, among them are the NIST developed S-Check[5] and MultiKron[6, 7, 8], the on-chip performance counters in newer microprocessors (UltraSparc, MIPS 10000, and Pentium-Pro), as well as Unix provided resource information.

We are currently porting a number of applications to the cluster. These parallel applications use a message passing environment, PVM or MPI, to communicate between processes. We will focus our measurement and analysis on two applications, a 3D Helmholtz solver application using PVM, and an Epitaxial Surface Growth application which uses MPI. The 3D Helmholtz solver application performs its communication in the form of a all-to-all pattern. The Epitaxial Surface Growth application communicates between neighboring processes. Scalability and comparative performance analysis will be presented and discussed for these applications.

## 2 CLUSTER DESCRIPTION

The NIST cluster computing testbed is built around existing scientific workstations and PCs using a Fore ASX-1000 ATM switch for the network interconnect. The switch can be configured with a maximum of 16 modules. Each module can be either a single OC-12 (622 Mbps) port or four OC-3 (155 Mbps) ports, yielding a total of between 16 to 64 ports. Our switch is configured as 60 OC-3 ports and one OC-12 port. The OC-12 port is for a future connection to an ATM backbone switch.

The workstations are connected to the ATM switch ports via an ATM interface card and multi-mode fiber optic cable, consisting of separate transmit and receive fibers. We are currently using ATM interface cards from both Fore Systems (200E) and Efficient Networks (ENI-155p). The software communicates using the Internet Protocol (IP), which in the ATM network is Classical IP. All of the ATM connections are Switched Virtual Circuits (SVC), set up via User-Network Interface (UNI) 3.0 and UNI 3.1 signaling.

The workstations include Silicon Graphics Indigo$^2$ and Indy, Sun Sparc-10 and UltraSparc, and Pentium-Pro based personal computers running Linux. In addition to a few IBM RS/6000s, DEC Alphas, and HP PA-RISCs, we have added 16 Pentium-Pro computers running Linux. Unlike the other machines which are the primary workstations of NIST scientists, these Pentium-Pro machines are not assigned to anyone, and thus are available at any time for cluster computing. All of the machines are dual-homed in that they are connected to both a switched Ethernet LAN, as well as the ATM LAN (switch). All of the network traffic normally associated with Unix networks (mail, NFS, etc.) occurs over the Ethernet network. The ATM network is dedicated to the communication necessary for the cluster computing work.

# 3 CASE STUDY 1 (3D Helmholtz Solver)

## 3.1 Application Description

Our first application is a Three-Dimensional Helmholtz Solver[9, 10], which implements a matrix decomposition algorithm to solve elliptic partial differential equations, specifically the Helmholtz equation. It was originally written in Fortran using the PVM communication library. We have converted the original program from Fortran to C, using the `f2c` utility. For the C version, we have also modified the communication portions to use the MPI communication interface standard. The MPI implementation used was Local Area Multicomputer (LAM) from The Ohio Supercomputer Center.

The three-dimensional matrix, of size $N^3$, is divided into $N$ two-dimensional slices, of size $N^2$ each. The application is restricted to matrix sizes that allow the slices to be evenly distributed among the $p$ processors, such that $N/p$ is an integer. Thus each processor operates on an $N^3/p$ portion of the matrix.

The Helmholtz Solver program is divided into three computational phases, each separated by a communication phase. The first computational phase performs a Fast Fourier Transform (FFT) on the matrix, followed by a communication phase to transpose the matrix. The second computational phase invokes a linear systems solver on the matrix. This is followed by another communication phase to again transpose the matrix. The final computational phase performs an inverse FFT to return the solution matrix to its original

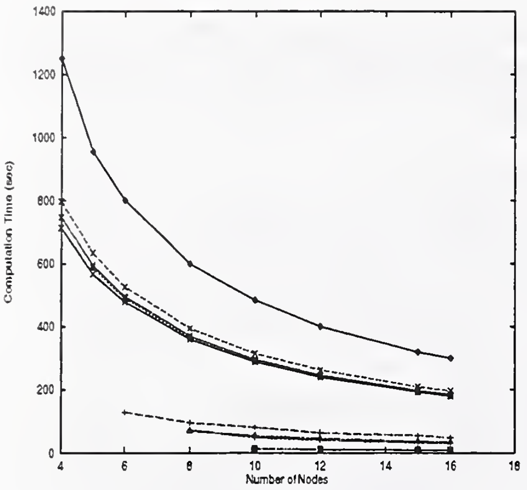

Figure 1: Computation time for 3D Helmholtz on heterogeneous ATM cluster

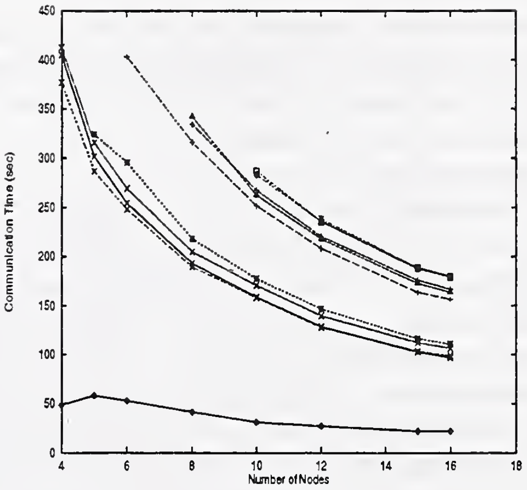

Figure 2: Communication time for 3D Helmholtz on heterogeneous ATM cluster

domain.

Each of the two FFT computational phases has a bimodal computational complexity, depending on the sub-interval size which is $N/2$ and constrained to be an integer. If the sub-interval can be factored into single digit prime numbers, then an optimized FFT can be invoked with a computational complexity of $O(N^3 \log_2 (N) + N^3)$. Otherwise a general FFT is invoked with a computational complexity of $O(N^4 + N^3)$. The linear systems solver phase has a computational complexity of $O(N^3)$.

Each of the two communication phases are identical, performing a transposition on the 3D matrix. Each processor manages an $N^3/p$ portion of the total data. In the communication phase each processor divides its portion of the data into $p$ chunks, of size $N^3/p^2$ each, and sends one chunk to each of the other $p-1$ processors and receives an equal size chunk from each of them. Thus a processor sends and then receives $\frac{N^3}{p^2} \times (p-1)$ data elements.

## 3.2 Performance Comparison

Our initial measurements of the Helmholtz solver program were on a 16-node heterogeneous subset of the cluster. These measurements were divided into overall run-time (how long to get my answer), computation time (time for all 3 computational phases), and communication time (time for each of the 2 communication phases, includes waiting time). These time values are the maximums, showing the worst-case performance. For any given data set we noticed a wide disparity in the communication time for the different processor types, as expected. From these measurements and the structure of the code it was easy to conclude that this code is indeed large-grained, applicable to loosely coupled cluster computing, and that it is best suited to run on homogeneous nodes vs. heterogeneous ones. Figures 1 and 2 show the individual computation and communication times for several machines in the heterogeneous ATM cluster. The measurements illustrate that the computation times are consistent with processor speed. Thus the faster machines have shorter computation times and longer communication times, indicating they are waiting for the slower machines to finish their computations. The faster machines are therefore under utilized. The two communication phases provide a synchronization point between all the processors, because each processor must wait to receive from all the others. When run on a homogeneous subset, the computation and communication times are more uniform, indicating that less waiting is occurring.

To verify the scalability of the Helmholtz solver program, we ran it on a 16 node subset of our IBM SP2, a highly integrated cluster with a node-to-node communication bandwidth of 400 Mbits/s, and an 8 node SGI Indigo$^2$ subset of our ATM cluster. Figures 3 , 4 and 5 show the run time, computation time and communication time, respectively, for a fixed-size problem. The long run time for the single node SGI is caused by paging due to insufficient real memory (128M on the SGIs vs. 512M per SP2 node). As additional nodes are added, the SGI computation times drop dramatically, to even better than the SP2 system.

The graphs shown in Figures 3, 4 and 5 provide a comparison of the relative speeds of our IBM SP2 and the SGI subset of our ATM cluster. For the computation phase, the SGI cluster scales well, and out-performs the SP2. However, the overall run time does not consistently reflect this higher computation performance, due to some erratic behavior of the communication phase on the SGI cluster for low levels of parallelism. As can be seen in Figure 5, the SP2 cluster has better communication behavior than the SGI cluster in the region of low parallelism. As the parallelism increases, the SGI cluster performs in a similar manner to the SP2, with slightly better run times. We are not sure what causes the peaks in the SGI communication.

A common approach in parallel computing is to scale the data along with the number of processors so that each processor continues to operate on approximately the same amount of data and incurs approximately the same amount of processing time. Figures 6, 7 and 8 show, for a scaled problem size, the run time, computation time, and the communication time, respectively. Although one would expect an approximately flat curve for computation time, there are two factors which contribute to the variations shown.

First, to hold the per processor data constant requires $N^3/p$ to be constant, but the program also requires $N/p$ to be an integer so that the slices are distributed evenly. Satisfying the second constraint causes $N^3$/p to vary as much as 10%.

The second factor was more difficult to determine. We observed peaks in the computation time that were not proportional to the data size variation. Our initial thoughts regarding causes were poor data alignment resulting in cache thrashing or even virtual memory paging. Our calculations indicated that the entire dataset should fit in primary memory, with no paging. We then used the SGI `perfex` tool[11] which utilizes the on-chip performance registers of the R10000 processor. This allowed us to acquire the cache misses incurred by our program. Unfortunately, this did not correlate with the computation time variations. Using `perfex` further, we investigated the total number of instructions and the number of floating point instructions executed. Here, we found a strong correlation with

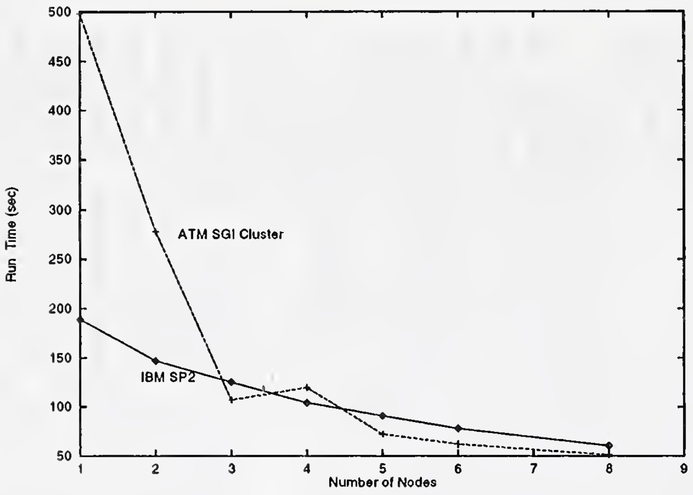

Figure 3: Run time for 3D Helmholtz on SP2 and SGI ATM cluster

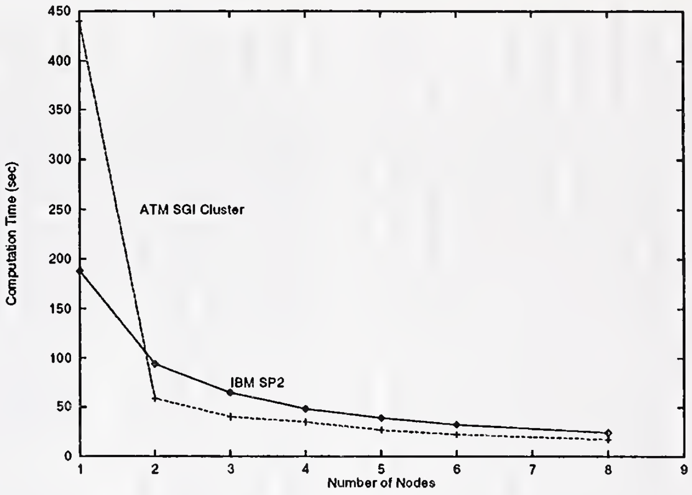

Figure 4: Computation Time for 3D Helmholtz on SP2 and SGI ATM cluster

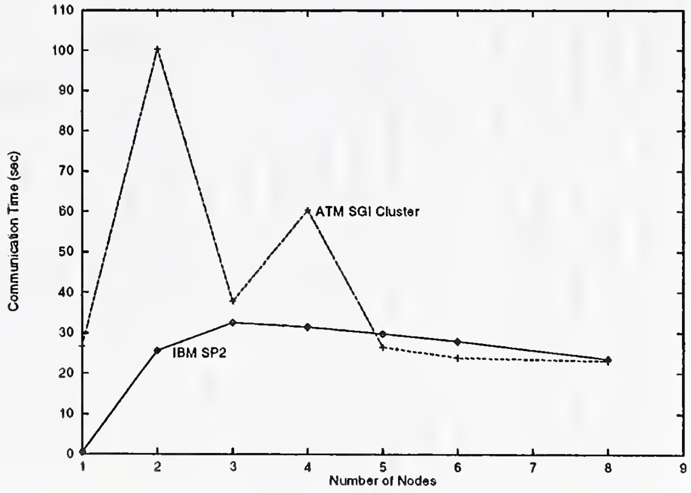

Figure 5: Communication time for 3D Helmholtz on SP2 and SGI ATM cluster

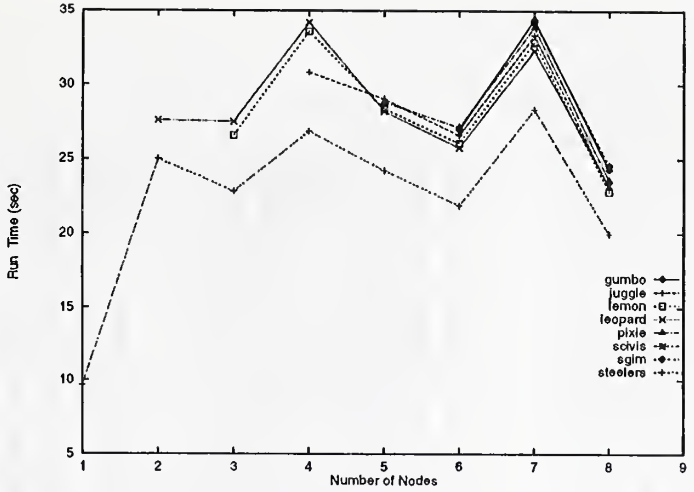

Figure 6: Run time for 3D Helmholtz with scaling problem size on SGI ATM cluster

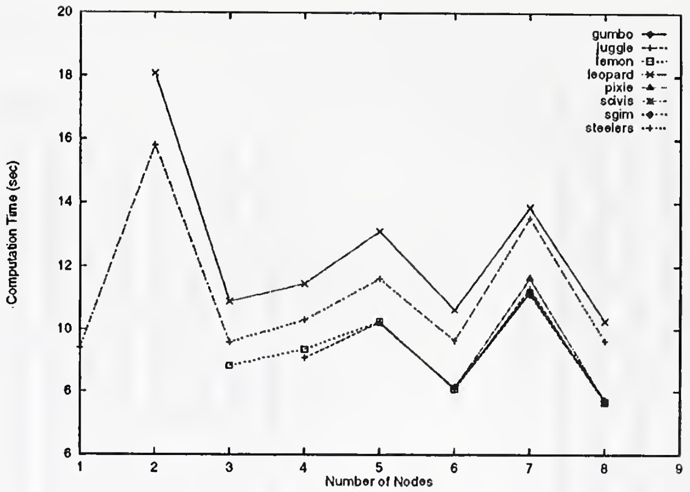

Figure 7: Computation time for 3D Helmholtz with scaling problem size on SGI ATM cluster

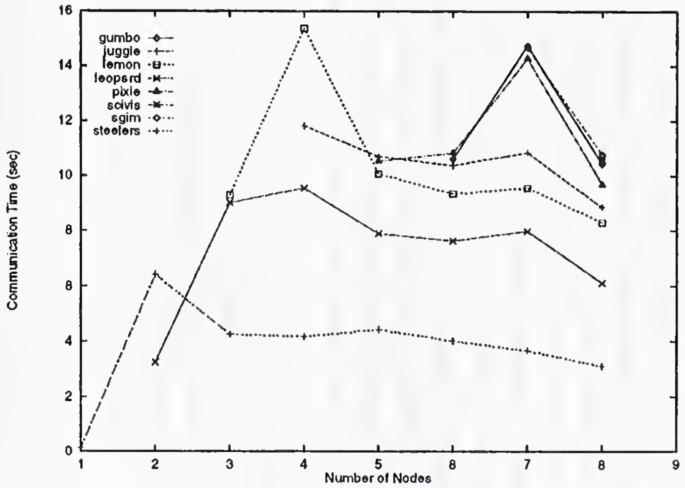

Figure 8: Communication time for 3D Helmholtz with scaling problem size on SGI ATM cluster

the computation time variations. This is where we were informed of the existence of the optimized FFT of $O(N^3\ log_2\ (N))$ vs. the general FFT of $O(N^4)$, and its dependency on the value $N$. Choosing a value $N$, such that it satisfies the criteria for the optimized FFT, versus the general FFT, can cause variations of 100% in the computation time.

As can be seen in Figure 8, there is a large variation in communication times for different processors in the SGI cluster. The pseudo code for the communication algorithm is:

```
for $i = 0$ to $p - 1$
    if ( $i$ != myself )
        send $N^3/p^2$ data elements to node $i$
        receive $N^3/p^2$ data elements from node $i$
```

We used the S-Check Barrier test[12, 13] to determine communication delays. This test provides a measure of the time variation between nodes completing their computation phase, and thus delaying the transmission of their data. Our analysis of this algorithm located the potential bottleneck in communication. When node $i$ sends to node 0, node $i$ then waits to receive from node 0, but node 0 does not send to node $i$ until it sends and receives from nodes $1 \ldots i - 1$. Node $i$ is forced to wait for its first receive, ignoring any other data that has arrived. This waiting occurs for most of the other nodes, but is more pronounced for the $(p-1)^{th}$ node, because node $i-1$ must wait until node $i$ has finished its communication. We revised the algorithm to first send everything, then to process receives in the order they arrive, as shown in the following pseudo code:

```
for $i = 0$ to $p - 1$
    if ( $i$ != myself )
        send $N^3/p^2$ data elements to node $i$
for $i = 0$ to $p - 2$
    receive $N^3/p^2$ data elements from any node
```

The results of this revised communication algorithm are shown in Figures 9, 10 and 11 along with the results of the original communication algorithm. As can be seen, the revised algorithm performs as much as 1.5 times faster. For the 2 node case both algorithms perform in a similar manner, in that both algorithms execute one send of half of their data to the other node and then execute one receive for the same amount of data. However, the revised algorithm uses twice the memory of the original algorithm for communication buffer space. The reason is that the outgoing message buffers cannot be reused until an acknowledgment is received from the target node, but this algorithm does not wait for acknowledgment for each message.

A potential problem with this revised algorithm is that all nodes first send to node 0, then to node 1, etc. If all nodes are homogeneous, then there is a significant probability that simultaneous transfers from multiple sources to a single destination could overwhelm the network/switch or flood the buffers of the PVM or LAM daemon handling the communications. Although we were expecting to see such behavior, it did not occur. No dropped cells were reported from the ATM switch or the ATM receiver card. Most likely there was a combination of things that prevented this from occurring. One is enough time variation between nodes that they all do not send to the same node at the same time. Another is sufficient buffering in the switch and receiving node to accommodate the burst of data. A solution for this would be to stagger the transmission destinations of each node as illustrated in the following pseudo code:

```
$i$ = myself
for $j = 0$ to $p - 1$
    $i = i + 1$
    if ( $i > p - 1$ )
        $i = 0$
    if ( $i$ != myself )
        send $N^3/p^2$ data elements to node $i$
for $i = 0$ to p−2
    receive $N^3/p^2$ data elements from any node
```

Figures 9, 10 and 11 show the LAM and PVM communication times for three different problem sets. The LAM version initially provides better results than the PVM version for up to a small number of processors. As the problem is scaled above five processors, the PVM version performs better.

Although PVM has no flow control, it does have error control, so any lost messages would be retransmitted causing additional traffic and longer communication times. While the ATM switch and buffers were not overwhelmed, we do expect to see throttling of the multiple byte streams as each node merges and sequentializes multiple streams. Both throttling and retransmissions, we believe, are the cause of the peaks in Figures 9, 10 and 11.

### 3.3 Evaluation

The 3D Helmholtz Solver is a large-grained application and was expected to scale well on our cluster. For a fixed problem size, the overall execution time can be reduced by distributing the problem over a number of processors. As the problem size grows, the overall execution time can be kept about the same by distributing the problem over a number of processors. If the problem size exceeds the main memory size, paging delays will degrade the execution time more so than the communication delays. By distributing the problem over a number of processors, the amount of memory required by each individual processor can be controlled and the benefits of our faster ATM communications will prevail. Thus problem sizes that are too large for a single machine, or a small number of processors, are now feasible over a larger virtual machine.

Running the Helmholtz Solver on our SGI cluster compares favorably to running on the IBM SP2 system. However, there are several caveats. First, the LAM and PVM

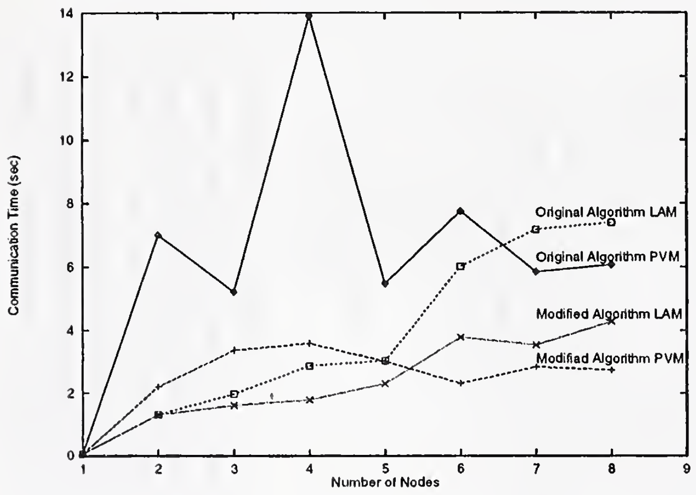

Figure 9: Communication time for 3D Helmholtz with 4-MByte problem size on SGI ATM cluster

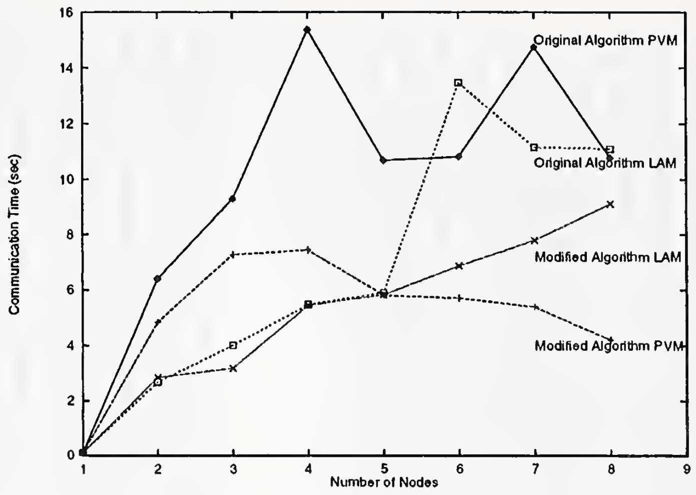

Figure 10: Communication time for 3D Helmholtz with 8-MByte problem size on SGI ATM cluster

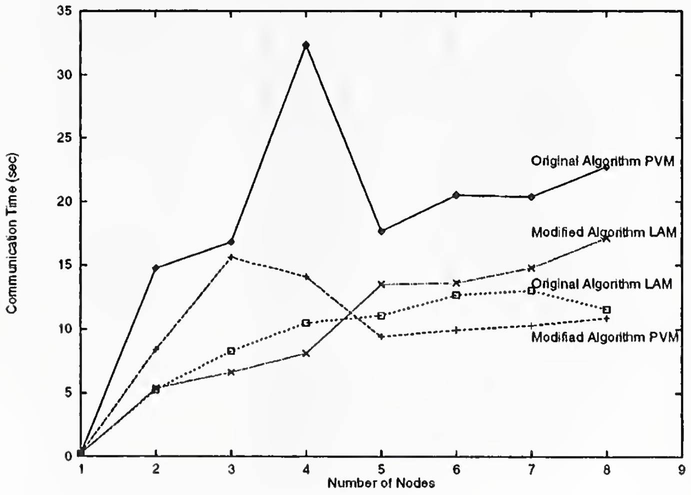

Figure 11: Communication time for 3D Helmholtz with 16-MByte problem size on SGI ATM cluster

communication libraries still have some erratic behavior, such as the much longer than expected communication times at certain dataset sizes. Second, on a heterogeneous cluster, adding nodes is not always beneficial, due to the synchronizing nature of the communication during the transposition phase. Adding a slower machine will force the other machines to wait at a communication phase, thereby slowing down the overall computation. A "slower" machine does not necessarily mean a slow architecture, but can mean "less responsive." So, for example, a machine that suddenly becomes busy with other work can slow down an entire computation. In such cases the capability for processes to migrate to other machines would be beneficial.

# 4 Case Study II (Epitaxial Growth Simulation)

## 4.1 Application Description

The molecular beam epitaxial growth (MBE) application uses Monte Carlo simulation techniques to model the growth of layers of atoms sprayed onto a surface[14]. The MBE code, written in C and using the MPI standard, iterates through a number of steps. Each step represents the introduction of and the operation on a single atom. The simulation models the number and distribution of atoms over the surface. Time is inferred from the probability of the event that occurred during each step. Thus each step represents a variable amount of time, the lower the probability the longer the time. During a step an atom is introduced with a deposition rate determined by the generation of a random number. This rate is used to traverse a rate tree. As the atom traverses the tree, its rate is reduced. When the rate of the atom is reduced below that of the current surface position the atom is deposited there. If the atom's diffusion rate exceeds the local diffusion threshold, the atom diffuses to one of the immediate neighboring cells. The step is completed when the deposition rate of the atom's final destination, and its immediate neighbors, are updated, and the path used in the deposition rate tree is recomputed.

The parallelized version subdivides the surface grid into uniform subgrids. Each processor executes a separate, but synchronized, step on its own subgrid, thereby generating one atom per subgrid at each step. The interaction between processors occurs when an atom lands on a boundary of its subgrid or diffuses across the boundary to a neighboring subgrid. This information must be communicated to that neighbor, so that the cell population and neighboring cell deposition rates are updated before the path through the subgrid rate tree is recomputed. In addition, when a diffused atom crosses to a neighboring subgrid and lands in the same grid cell as the atom from that subgrid, a conflict arises. Only one of these two events can occur. The earlier event, the one with the highest probability, would have changed the deposition rates so that the later event would have occurred differently. The later event, the one with the lower probability, is canceled and the earlier event is committed, and new rates are computed. The cancelation resulting from such a conflict must also be communicated back to the original processor. If a corner cell is involved, then two additional neighbors will have to be notified.

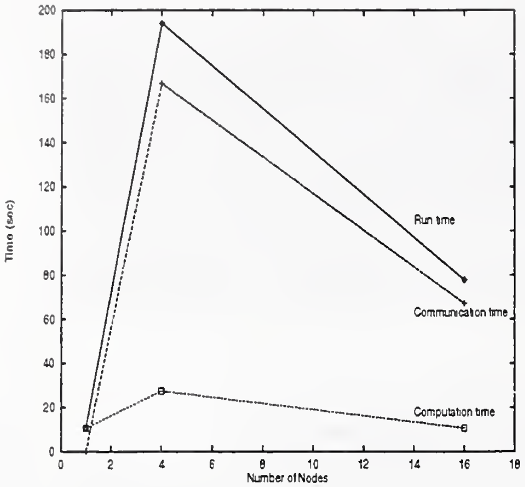

Figure 12: Run, communication, computation times for MBE application, fixed problem size

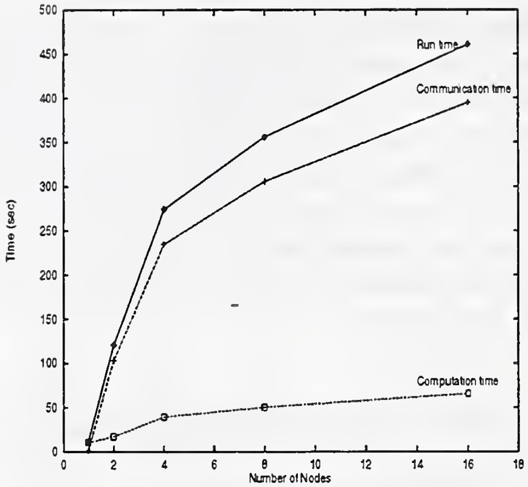

Figure 13: Run, communication, computation times for MBE application, scaling problem size

Thus the phases of a parallelized step are: (1) generate an atom and determine its destination, (2) synchronize and communicate those atoms that effect or cross a processor boundary and resolve conflicts, and (3) commit the surviving atoms and update the deposition rate tree. Phases (1) and (3) are of computational complexity $O(log_2(N^2))$, where $N^2$ is the size of the subgrid, because the deposition rates are organized as a tree whose leaves are the cells of the subgrid. Phase (2) is predominately synchronization and communication.

### 4.2 Performance Comparison

Our initial measurements were again divided into overall run time, computation time, and communication time. These measurements on a fixed sized problem are shown in Figure 12. Of importance to notice is that the communication dominates this parallel application. Therefore, the single processor version is considerably faster than the multiprocessor version since no communication is involved. For the multiprocessor version, acceleration does occur, relative to the baseline multiprocessor version (either 2 or 4 processors), as additional processors are added. This acceleration is due to a smaller subgrid and less steps per processor. An $N^2$ grid running 1600 steps on a single processor, uses an $N^2/4$ subgrid running 400 steps on each of 4 processors, and uses an $N^2/16$ subgrid running 100 steps on each of 16 processors, and so on.

Figure 13 shows similar graphs for a scaled problem size, in which the subgrid and number of steps are held constant for each processor. This yields a larger overall grid, $pN^2$ for $p$ processors, and $p$ times as many total steps, resulting in simulating more atoms over a longer simulated time.

Running S-Check on the MBE code yielded computational sensitivities in a few routines.

As shown in Figures 12 and 13, this code is communication bound, not computationally bound. Since computation time is such a small fraction of the overall execution time, it was not beneficial to invest the time to tune the computation portion for only a few percent of improvement.

The communication is localized to a single routine which first synchronizes all processors, then messages are only sent to affected neighbors informing them of atoms that are on the boundary or have crossed the boundary. The routine then enters a receive loop, calling the MPI "Allreduce" function, synchronizing via a root node, to determine if there are any incoming messages to process and if communications are completed for this step. Completion of communication is determined by tallying and comparing total sends and receives. The pseudo code for the original communication algorithm (**ORIGINAL**) is shown in Figure 14.

Although this algorithm is efficient in its explicit use of sends and receives to notify neighbors of boundary activity and their cancelation, it is inefficient in its synchronization and determining completion of communication. Since the number of boundary events can be anywhere from 0 to 4, a node does not know how many, if any, incoming messages to process. Thus, an abundance of synchronization messages are used to determine when to stop.

The current communications algorithm suffers from significant overhead in synchronization via the MPI "Allreduce" function. This is a "centralized" function that takes a parameter from each processor, sends it to a root node which performs a specified operation, such as add, on all copies and broadcasts the result back to all of the processors to complete the function. Instead of using three separate scalar variables (all_done, total_sends, and total_receives) to synchronize and determine completion, we propose to use two vector variables, each of length $p$, where $p$ is the number of processors.

The first vector would be initialized by the first set of sends. A send to processor $i$ would set the send_vector[$i$] = 1. After this vector is processed by the function, the returned rcv_vector[$i$] contains the total number of receives for processor $i$. Thus, each processor knows the number of receives to execute. If any of these receives cause a conflict that results in sending a cancelation message, a second set of receives is required. Thus, a second vector would tally these sends and specify the number of cancelation receives each processor must handle. This revised algorithm requires, at most, two "Allreduce" synchronizations, and if no atom crosses a boundary, which is most of the time, then only one synchronization is required. This boundary-crossing information can be incorporated into the first send vector. The pseudo code for this revised algorithm (**REV1**) is shown in Figure 15.

An alternative algorithm is a centralized algorithm. Each processor would send, in one message, its atom's destination to the root processor and receive back a single message with the committed destination or cancelation of its atom. The root processor would collect the information from all the other $p-1$ processors and decide the results of any conflicts and then send $p-1$ messages back to the other processors with the committed results for each processor. The pseudo code for this communication algorithm (**CENTRAL**) at all but the

**Algorithm ORIGINAL:**

```
send_cnt = num_sent = num_rcv = 0
barrier sync
for each affected neighbor i
    send atom destination to i
    increment send_cnt; increment num_sent
not_complete = 1
while ( not_complete )
    Test_how_many_of_my_sends_are_done (send_cnt,sends_done)
    if ( sends_done )
        Allreduce(sends_done,all_done,1)
        if ( all_done )
            Allreduce(num_sent,total_sent,1)
            Allreduce(num_rcv,total_rcv,1)
            if ( total_sent == total_rcv )
            not_complete = 0; break;
    if ( rcv_msg_avail )
        increment num_rcv
        if ( cancel_msg )
            cancel boundary crossing event
        else
            process event on or crossing my boundary
            if ( conflict )
                for each affected neighbor j
                    send cancel msg to node j
                    increment send_cnt; increment num_sent
```

Figure 14: Epitaxial Growth Simulation, Algorithm ORIGINAL

**Algorithm REV1:**

```
clear send_vector
if ( atom is on or crosses boundary )
    for each affected neighbor i
        send atom destination to node i
        if ( crosses boundary ) send_vector[i] = Cross_Flag
            increment send_vector[i]
Allreduce(send_vector,rcv_vector,p)
if ( SUM_of(rcv_vector) != 0 )
    clear send_cancel_vector
    for i=0 to rcv_vector[my_node]
        rcv_msg
        if ( crossed_and_conflict )
        for each affected neighbor j
            send cancel msg to node j
            increment send_cancel_vector[j]
        else
            accept event info
    if ( Any_Crossing_In(rcv_vector) )
        Allreduce(send_cancel_vector,rcv_cancel_vector,p)
        for i=0 to rcv_cancel_vector[my_node]
            rcv atom cancelation msg
```

Figure 15: Epitaxial Growth Simulation, Algorithm REV1

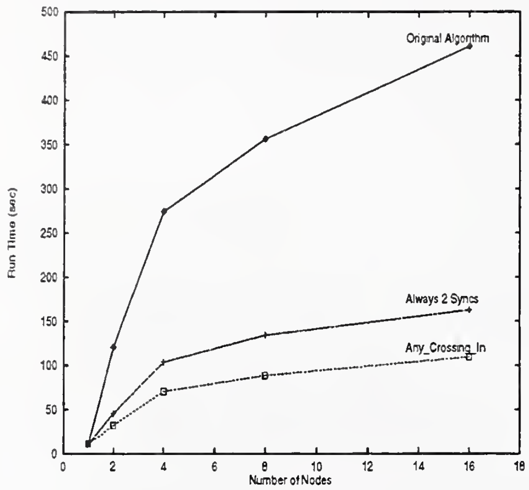

Figure 16: Run time for MBE application, algorithm changes, scaling problem size

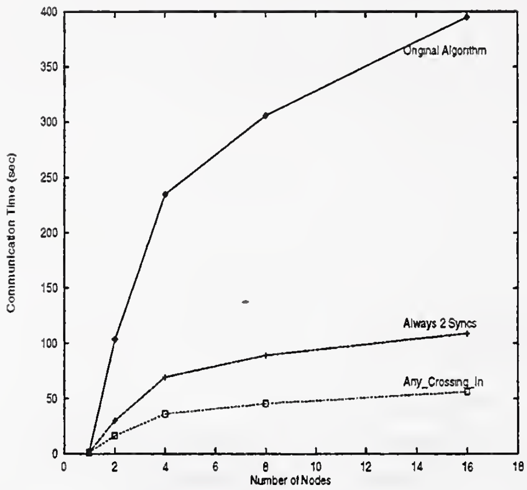

Figure 17: Communication time for MBE application, algorithm changes, scaling problem size

root processor is:

**Algorithm CENTRAL:**

```
    send atom destination to root processor /* 1 msg (send) */
    receive committed destination from root processor /* 1 msg (rcv) */
```

This centralized algorithm also results in a deterministic number of communications for each processor. Although the total amount of communication is less than the distributed algorithm, it implicitly imposes a sequentialization of the messages at the root processor for the $p - 1$ send messages. Instead of sending $p - 1$ individual response messages, the root node can obtain some efficiency, via concurrency, by sending a single, although larger, broadcast message containing all atoms and their committed positions.

As the number of processors increases, the communication time for the centralized algorithm should increase proportionally. The communication time for the distributed algorithm should stay nearly constant, independent of the number of processors. So, initially the centralized algorithm should perform better, but as the number of processors increases sufficiently, the distributed algorithm should perform better. We have not yet implemented the distributed communications algorithm since our current cluster size is still relatively small. We plan to implement and test it, once our cluster size significantly increases, in the very near future.

The results of the revised (**REV1**) and original communication algorithms are illustrated in Figures 16 and 17. These figures show that the complete revised algorithm,

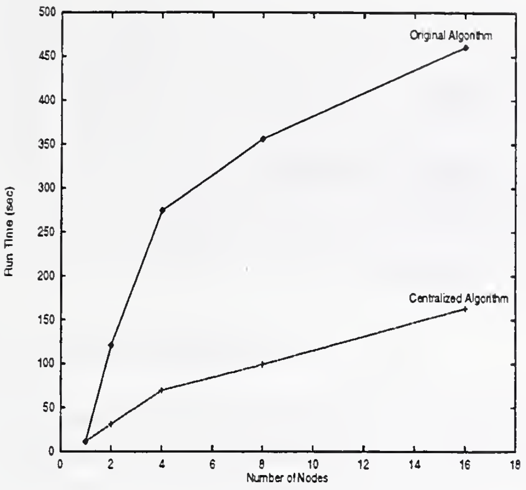

Figure 18: Run time for MBE application, centralized algorithm, scaling problem size

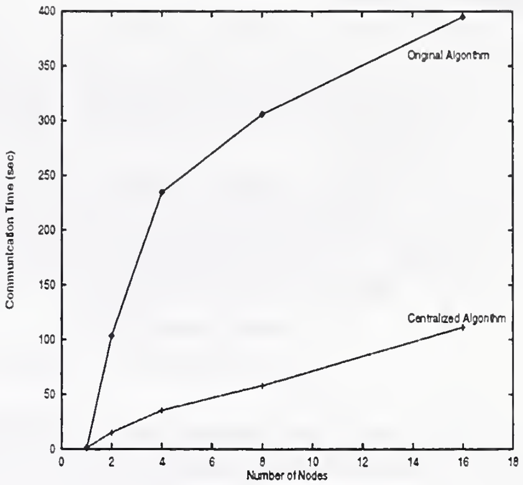

Figure 19: Communication time for MBE application, centralized algorithm, scaling problem size

shown as the "Any_Crossing_In" curve, can achieve a 75% to 80% communications acceleration over the original algorithm on our SGI cluster subset. Without the "SUM_of" and "Any_Crossing_In" tests in the **REV1** algorithm to avoid a second "Allreduce" synchronization, we only achieve about a 50% to 60% acceleration, shown as the "Always 2 Syncs" curve. The "Any_Crossing_In" test contributes only about 2%. The overall execution time shows an acceleration of from 60% to 70% for the **REV1** communications algorithm compared to the original. This **REV1** algorithm results in a more balanced execute cycle, where the communication represents about 82% of the original algorithm, it now represents about 67% for the **REV1** algorithm. But the **REV1** algorithm suffers the same problem as the centralized algorithm, since they both utilize centralized communications, they will both incur communication time increases proportional to the increase in the number of processors.

The results of the centralized (**CENTRAL**) communication algorithm are illustrated in Figures 18 and 19. These graphs show that the **CENTRAL** algorithm achieves a 75% to 85% communication acceleration over the **ORIGINAL** algorithm on our cluster. This acceleration is very similar to the **REV1** algorithm's performance, as expected, and so is the overall execution time acceleration of 60% to 70%.

An alternative distributed algorithm is to always send messages to your neighbors, resulting in 8 sends per processor ($\mathcal{N}, \mathcal{E}, \mathcal{W}, \mathcal{S}$ and 4 diagonal neighbors) and to require that a response be sent back for each atom crossing a boundary, whether it is canceled or committed instead of just if it is canceled. This change would result in a fixed number of sends and a deterministic number of responses, thus eliminating the need for additional synchronization messages. The pseudo code for this communication algorithm (**NEWS**) is shown in Figure 20. We didn't have time to investigate the run-time performance of this algorithm.

**Algorithm NEWS:**

```
send to all neighbors /* 8 msgs (sends)*/
for ( each neighbor rcvd a msg ) /* 8-12 msgs (rcvs)*/
    rcv_msg
    if ( neighbor atom in corner crossing another neighbor boundary )
        accept event info & increase expected rcv msg by 1
    else if ( atom crossing boundary )
        for each affected neighbor /* 1 or 3 msgs (sends) */
            if ( crossed_and_conflict )
                send cancel msg to neighbor
            else
                send commit msg to neighbor
    else if ( atom on boundary )
        accept event info
    else if ( no atom near boundary )
        NULL (continue)
```

Figure 20: Epitaxial Growth Simulation, Algorithm NEWS

### 4.3 Evaluation

The MBE parallel code is a fine-grained application; a small amount of computation is followed by a small amount of communication. This is generally a disqualifying property for candidacy as a parallel code. Our measurements clearly show that a single processor version is faster than the parallel versions. The driving force here is memory capacity instead of the more common execution cycle capacity. For example, a 32 processor parallel version, with each processor having 512 MBytes of memory, can achieve a 16 GByte effective memory for this simulation, far more than is available on any single processor.

## 5 Conclusion

Based on our analysis of two applications ported to our heterogeneous ATM cluster of workstations we are pleased with the performance of these NIST applications on the cluster. These applications, although vastly different, both ported well and for some subsets of our cluster performed better than in their original parallel environments.

The 3D Helmholtz Solver application is a coarse-grained code and scales well, as expected. Better performance and system utilization is achieved when a homogeneous subset of machines are selected, since computation and communication are implicitly synchronized and less waiting occurs. Because large bursts of communications occur, we are able to benefit from using the ATM network.

The MBE code is a fine-grained code and would generally not be considered for parallelization in a distributed environment. As such, the parallel version is slower than the single processor version, although the parallel version does scale as processors are added. The benefit from parallelizing this code is from the aggregate memory rather then total compute cycles. Although the parallel version is highly dependent on communication, it is not clear if any benefit is derived from our ATM network since the amount of each data transfer is small. The network architecture is more important to the efficiency of this application. The MBE code would map very efficiently onto a mesh architecture, accommodate a switched architecture reasonably, whereas a shared media architecture would be inefficient.

## Acknowledgments

We would like to thank Karin Remington and Isabel Beichl for their assistance with understanding the 3D-Helmholtz solver and Epitaxial growth applications. Both researchers have spent many hours working with us during the porting of the programs to our cluster.

## References

[1] C. Fischberg, C Rhie, R. Zacharias, P. Bradley, and T. DesSureault, "Using Hundreds of Workstations for Production Running of Parallel CFD Applications", *Proc. of Parallel CFD '95*, Pasadena, CA, June 1995.

[2] D. Becker, T. Sterling, D. Savarse, J. Dorband, U. Ranawake, and C. Packer, "BEOWULF: A Parallel Workstation for Scientific Computation", *Proceedings of the International Conference on Parallel Processing*, Urbana-Champaign, Ill, Vol. I Architecture, pp I11-I14, August 1995 (a version of this document is available at *http://cesdis.gsfc.nasa.gov/linux/beowulf/icpp95-.html*).

[3] *Pentium Pro Cluster Workshop,* Sponsored by Ames Laboratory and Iowa State University, Des Moines, Iowa , April 10-11, 1997 (Information available for a limited time at *http://www.scl.ameslab.gov/workshops/index.html*).

[4] "Issues and Challenges in ATM Networks" Special Issue *Communications of the ACM*, Vol 38, No. 2, Feb 1995.

[5] R. Snelick, "S-Check: a Tool for Tuning Parallel Programs," *Proceedings of the 11th International Parallel Processing Symposium (IPPS 97)*, Geneva, Switzerland, April 1-5, 1997, pp 107-112.

[6] A. Mink, *Operating Principles of MultiKron II Performance Instrumentation for MIMD Computers*, National Institute of Standards and Technology, NISTIR 5571, December 1994 (available at *http://www.multikron.nist.gov/multikron/multikron.html*).

[7] A. Mink, G. G. Nacht, and R. J. Carpenter, *Operating Principles of the SBus MultiKron Performance Interface Board*, National Institute

of Standards and Technology, NISTIR 5652, May 1995 (available at *http://www.multikron.nist.gov/multikron/multikron.html*).

[8] A. Mink and W. Salamon, *Operating Principles of the PCI Bus MultiKron Performance Interface Board*, National Institute of Standards and Technology, NISTIR 5993, March 1997 (available at *http://www.multikron.nist.gov/multikron/multikron.html*).

[9] K. Remington Bennett, "Fast Direct Solution of Three-Dimensional Poisson and Helmholtz Problems on Distributed Memory Machines," *Proceedings of Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, Virginia, March 22-24, 1993.

[10] R. Pozo and K. Remington, "Performance Characteristics of Fast Elliptic Solvers on Parallel Platforms," *Proceedings of the 1st Euro PVM User's Group Meeting*, Rome, Italy, 1994.

[11] Zagha, Larson, Turner and Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," *Supercomputing '96 Conference Proceedings*, Pittsburgh, PA, November 17-22, 1997.

[12] R. Snelick, J. Ja'Ja', R. Kacker, and G. Lyon, "Synthetic-perturbation techniques for screening shared memory programs," *Software - Practice and Experience*, 24(8)(1994) 679-701.

[13] R. Snelick, M. Indovina, M. Courson, and A. Kearsley, "Tuning Parallel and Networked Programs with S-Check," *Proceedings of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas, Nevada, June 30 - July 3, 1997.

[14] I. Beichl, Y. Teng, and J. Blue, "Parallel Monte Carlo Simulation of MBE Growth," *International Parallel Processing Symposium*, April, 1995.