Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

PARALLEL IMPLEMENTATION OF A MOLECULAR DYNAMICS SIMULATION PROGRAM

Alan Mink
Christophe Bailly

Scalable Parallel Systems and Applications Group
Information Technology Laboratory
National Institute of Standards and Technology (NIST)
Gaithersburg, MD 20899, U.S.A.

ABSTRACT

We have taken a NIST molecular dynamics simulation program (*md3*), which was configured as a single sequential process running on a CRAY C90 vector supercomputer, and parallelized it to run in a distributed memory message passing environment. Since portability was a major concern during parallelization, we used the Message Passing Interface (MPI) standard. The features of MPI provide a basic set of interprocess communication primitives on many architectures. The parallel *md3* program has two basic algorithms resulting in a MPMD (Multiple Program Multiple Data) structure, versus the more common SPMD (Single Program Multiple Data) structure, and has the potential to exploit heterogeneous processing. For any given number of nodes we have devised an equation to determine the initial node allocation among these multiple programs which yields near optimal load balance. We also dynamically manage the load balance between processes to correct for run time variations and to achieve better performance. We compare the performance of this MPMD parallel code run on a range of distributed memory machines (an IBM SP2, a cluster of Pentium Pros, and a cluster of SGI Indigo2s with the R10000 processor) against the original code performance on the Cray. In addition to better performance, the code on distributed memory machines offers an ability to scale the problem size based upon the combined memory size of the host systems.

1 INTRODUCTION

The NIST Cray C90 is a six processor vector supercomputer, although operationally each processor normally runs independent processing streams. Because it is an expensive resource it must be shared among a large number of researchers, thus creating a critical resource with heavy demand. Distributed processing offers similar high performance computing through parallel processing; in addition it has a lower cost and the potential to scale the system to handle larger problem sizes. Problem size limited by memory size is increasingly a computational bottleneck. The current trend in parallel computing is towards clusters of computer nodes that are either Symmetric Multiprocessors (SMPs) or single processors. These clusters are capable of supporting both shared memory (SM) and distributed memory (DM) programming paradigms, although shared memory variants cost more or run more slowly. Current clusters implement parallelism by using commodity microprocessors interconnected with either commodity or proprietary networks. Clusters are more scalable than SMPs alone, because networks scale better than buses in terms of aggregate bandwidth and number of nodes.

Integrated clusters are commercially available (e.g., SGI/Cray Origin 2000 and HP/CONVEX Exemplar). These clusters have built a significant infrastructure between the microprocessor and the memory hierarchy, as well as a superior interconnect, to enhance performance. Clusters built using commodity PCs or workstations and commodity LANs are emerging as an attractive and possibly low cost alternative to traditional high performance distributed processing on integrated clusters. The distributed memory paradigm for parallel computing is widely used and standardized interfaces such as MPI, see Pacheco (1997), have enabled portability between clusters from different manufacturers. An advantage of the MPI standard is that different platforms can have their own optimized implementations.

Using MPI enables the parallel *md3* program to run, for example, on an SGI cluster, a Pentium cluster, the IBM SP2, SGI Origin 2000, etc. The implementation of MPI used on the clusters was Ohio Supercomputer Center's LAM 6.1, see Burns et al. (1994). The shared memory paradigm will similarly benefit from the newly proposed Open MP interface standard. Commodity clusters, as described by Becker et al. (1995), are commonly based on

the newer Intel Pentium processors which are capable of significant computing power, although floating point performance is not strong. However, a main factor limiting performance often is the overhead of the message passing communications network. Newer network technologies such as ATM and Fast Ethernet address this bottleneck.

NIST is investigating the benefits of both commercially integrated clusters and do-it-yourself commodity clusters. NIST has an IBM SP2 and an SGI Origin 2000. NIST has constructed a commodity cluster using an ATM (OC-3) LAN for the interconnect. This cluster consists of a number of heterogeneous machines, among which is a subcluster of 8 SGI workstations, each powered by the R10000 processor, and a subcluster of 16 identical 200 MHz Pentium Pro based machines; see Figure 1.

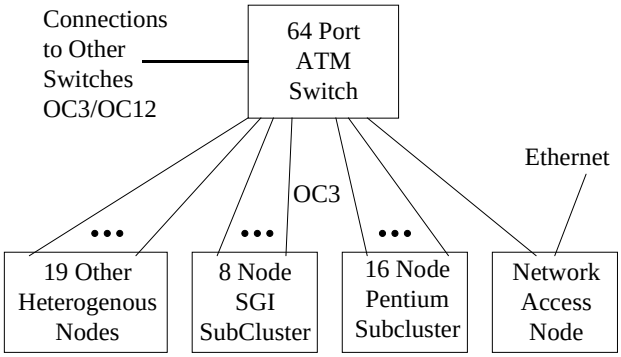


Figure 1: Block Diagram of the NIST ATM Cluster

As part of this evaluation we selected a sequential molecular dynamics simulation code, called *md3*, that was running on the Cray. We redesigned it as a parallel code using MPI to allow portability among our cluster architectures: the IBM SP2, the SGI subcluster and the Pentium subcluster. The purpose here is to describe the parallel redesign and to compare the performance of the parallel code on our clusters against the sequential code on the Cray.

2 SEQUENTIAL *md3*

Molecular dynamics simulation has become an important tool for the study of the liquid state at the molecular level. Possible applications include the study of molecular films and the study of chromatographic separation mechanisms. The *md3* program, written in Fortran 77, computes the motion of molecules from Sangster and Dixon (1976) in a cubic volume by calculating their interaction potential. This calculation consists of:

- short range forces, including repulsion and Van Der Waals terms, and
- long range coulombic forces.

The molecular state consists of the position, velocity and orientation of the particles. The *md3* program computes the interaction among all of the particles to obtain the resultant forces on each particle. Based on this force, each particle is moved a distance corresponding to a time duration of 10^{-15} seconds and given a new state. This cycle, called a step, is repeated until the simulated time is reached. For example, if the simulated time is 2 sec, then $2x10^{15}$ steps of the program must be executed.

Obtaining simulation results of relatively complex systems within a reasonable period of time requires (i) sufficient computer memory to hold the data describing the simulation and (ii) sufficient computation cycles to complete the task in the time allocated. Sequential programs running on monolithic supercomputers (e.g., the C90) are running out of both. Parallel computing can better accommodate these requirements.

The sequential *md3* program uses two input files:

- md35: which defines the features of the problem, such as the number of molecules, the geometry of the volume, the number of steps to compute, etc., and
- md38: which provides the initial position and state of each molecule.

The sequential program, whose flow chart is shown in figure 2, begins by reading the md35 file to obtain the problem description and the md38 file to obtain the initial particle positions and energies. Then the main loop executes two functions; *ewald* computes the long range forces and *vf2* computes the short range forces. These forces are combined and a new position and energy is computed for each particle over the duration of the time step. This main loop repeats until the simulated time is reached. When the main loop completes, the position and energy for each particle is written to an output file and the program terminates.

The *ewald* and *vf2* functions are both based on a double summation computation $\sum_{i=1}^{N}\sum_{j=1}^{N} p_{i,j}$, where N is the number of particles and $p_{i,j}$ is the interaction potential between particle i and j. By using the direct *ewald* summation method, the double sum of *ewald* can be rearranged into a single sum, yielding a linear computational complexity for *ewald*. The computational complexity for *vf2* remains of order N^2. We have verified these computational complexities during program execution via a profiler tool whose results are shown in figure 3.

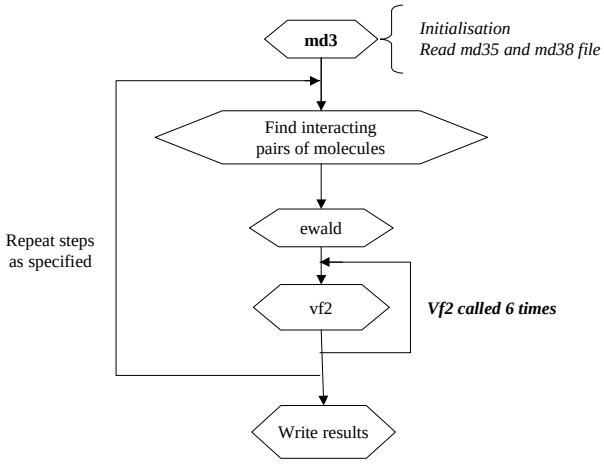


Figure 2: Flowchart of Sequential *md3* Program

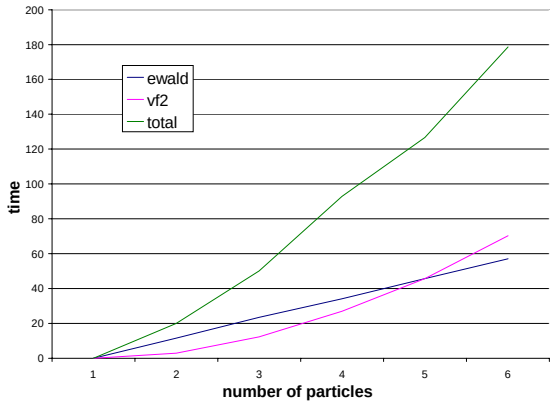


Figure 3: Measured Execution Times of *vf2* and *ewald*

Not only does *vf2* require the bulk of the computation, it also uses the most memory. The memory requirements for *vf2* are 120*N*(N-1)/2 bytes for N particles. Table 1 shows how quickly the memory requirement increases.

To determine if the sequential *md3* program is a true high performance vector code, we used the Cray C90 Hardware Performance Monitor (hpm) tool, see Morreale (1994), available with the UNICOS operating system. Its execution statistics are:

- 94.8% of floating point operations are vector operations,
- average vector length is 74 bytes, and
- average instruction rate is 67.5 MIPS.

Table 1: Memory requirements as a function of Particles.

Particles (N)	Memory (MBytes)
216	2.8
1000	59.9
2000	239.9
3000	539.8
4000	959.8
5000	1499.7

This shows that the code is mostly vectorized and is running at 170 MFLOPS. Nonetheless, *md3* is not taking significant advantage of vertorization because of its low vector length and low MIPS rating. A good vector code for the CRAY C90 can achieve 600 MFLOPS. In comparison the execution of sequential md3 on a 200 MHz Pentium PRO, on one processor of the IBM SP2, and on an SGI R10000 based workstation, is slower than on the CRAY C90. See figure 4, below.

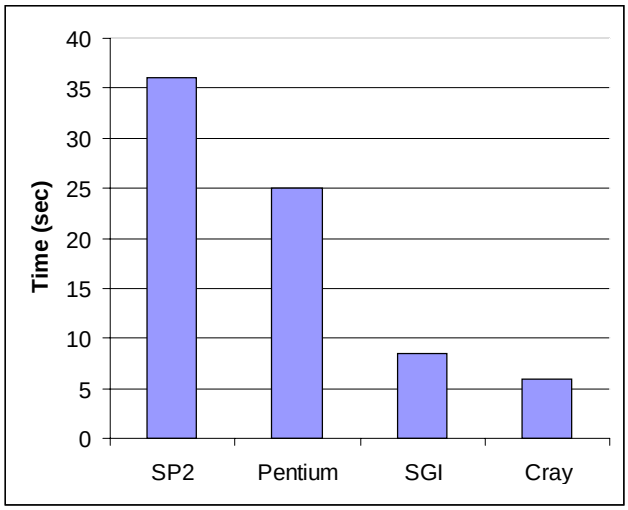


Figure 4: Comparison of Sequential *md3* Execution Times on Different Computers for 10 Time Steps

3 PARALLEL *md3*

The most common approach to parallelism is data parallelism, also referred to as Single Program Multiple Data (SPMD) parallelism, in which multiple copies of the same program cooperate in parallel on disjoint subsets of a common data set. A less common approach to parallelism is control parallelism, also referred to as Multiple Program Multiple Data (MPMD) parallelism, in which distinct programs cooperate in parallel.

The *md3* program can exploit both approaches to parallelism. Since the *ewald* and *vf2* functions calculate different independent components of the interaction forces they can be redesigned into separate, distinct cooperating MPMD parallel programs. In addition, each of these two

programs could further benefit from SPMD parallelism. The flow chart of the redesigned parallel version of *md3* is shown in figure 5. All communication and synchronization between processes are done using the MPI standard, which facilitates portability between different platforms. The IBM implementation of MPI was used on the SP2, while the LAM implementation was used on the SGI and Pentium clusters.

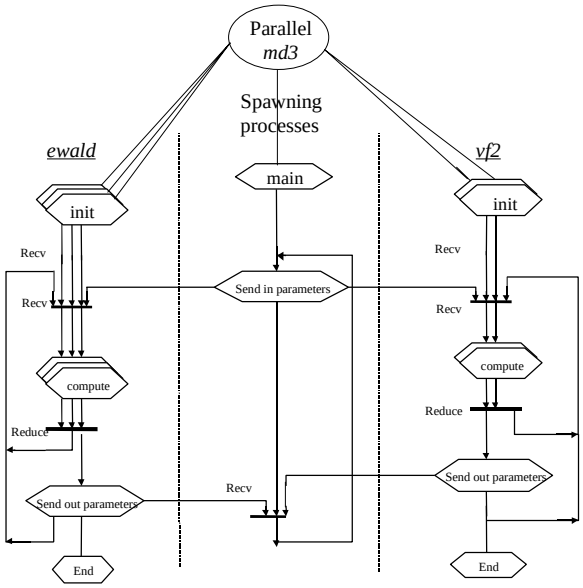


Figure 5: Flowchart of Parallel *md3* Program

The main process, written in Fortran 77 but converted automatically to C via the *f2c* utility, reads the md35 file to obtain the problem description and then spawns an appropriate number of processes for both *ewald* and *vf2*, both written in C. The main process then reads the md38 file to obtain the initial position and energies of all the particles. The main loop of the program now consists of broadcasting the current particle parameters to the processes of both *ewald* and *vf2*, waiting for the interaction results from all the processes, and calculating the new particle positions and energies. It then repeats the loop. The interaction results from each process can arrive and be processed in any order. Each MPI message is tagged to distinguish between *ewald* and *vf2* data.

The individual processes of both *ewald* and *vf2* wait for the new particle parameters to arrive, compute the interaction force on their subset of the data, send their results back to the main process, and then repeat the loop. Since the results of the *ewald* processes are not independent, they must first be integrated together before being sent back to the main process. We use the MPI collective function MPI_REDUCE to accomplish this. Similar dependencies exist for the *vf2* processes and the application of MPI_REDUCE is also employed. An optimal implementation for the MPI_REDUCE function is a binary tree which optimizes the overlapping summations. In this case, the minimum number of communication steps needed to gather the results is $\lfloor \log_2 P \rfloor$, where P is the number of processes.

Table 2 lists the input and output data requirements, in bytes, of the *ewald* and *vf2* processes.As the problem size, N, increases, the granularity of *md3* also increases, since communication time, O(N), grows slower than computation time, $O(N^2)$. This large grain characteristic makes *md3* an excellent candidate for parallel processing. An additional benefit to the parallel version of *md3* is that larger problem sizes can be run, since the distributed data can use the combined total memory of all the processing nodes.

Table 2: I/O Data Requirements.

	vf2	ewald
input parameters	48N	48N
output parameters	12052+80N	44+72N

4 LOAD BALANCING

To obtain optimum performance from a parallel code it is necessary to balance the execution time of the individual processes involved, since the slowest process dictates the performance achieved. For SPMD codes it is usually sufficient to distribute the data uniformly and spawn as many processes as there are nodes available. The nature of the problem or data can introduce further restrictions, such as only use an even number of nodes or a power of 2. Individually, for *ewald* and *vf2*, this is true on a set of homogeneous nodes, but because of the difference in computational complexity between the two it is necessary to determine an allocation of nodes that will achieve equal execution times for both. Because this is an MPMD code there is an opportunity to utilize heterogeneous processors, making the load balancing even more difficult.

4.1 Static Load Balancing

Our objective is to determine the allocation of P nodes among *ewald* and *vf2* processes so that the computation time of each process is approximately equal. For the general case of a heterogeneous environment, we can represent the computation times of the *ewald* and *vf2* processes, based on their computational complexities, as:

$$Te = Te(N')i \times \frac{CPUi}{\sum_{j=1}^{Pe} CPUj} \times \left(N / N' \right) \qquad (1)$$

and $$Tv = Tv(N')i \times \frac{CPUi}{\sum_{j=1}^{Pv} CPUj} \times \left(N / N' \right)^2 ,$$

respectively, such that P = Pe +Pv, where:

- N is the problem size in number of particles,
- N' is a problem size with known computation times,
- Pe is the number of *ewald* computational nodes,
- Pv is the number of *vf2* computational nodes,
- CPU_i is the relative processing rate of the ith CPU
- compared to CPU_1,
- Te(N')i the *ewald* computation time for N=N' on
- CPU_i,
- Tv(N')i the *vf2* computation time for N=N' on CPU_i.

For example, on the NIST ATM cluster we could allocate SGI nodes to *ewald* and Pentium nodes to *vf2*, or vice versa. The drawback here is that balancing the workload may result in a node allocation that doesn't use all the processors available.

For the case where all nodes are homogeneous, (1) reduces to:

$$\text{Te} = Te(N') \times \frac{(N/N')}{Pe}$$

$$\text{and} \quad \text{Tv} = Tv(N') \times \frac{(N/N')^2}{Pv} \quad .$$

Let k=Pe/P be the fraction of nodes allocated to *ewald* and (1-k)=Pv/P is the fraction of nodes allocated to *vf2*. Substituting k in the above equations yields:

$$Te = Te(N') \times \frac{(N/N')}{kP}$$

$$\text{and} \quad Tv = Tv(N') \times \frac{(N/N')^2}{(1-k)P} \quad .$$

Load balance is achieved when Tv=Te. Substituting the above equations in this equality and solving for k yields:

$$k_{opt} = \frac{1}{1 + \frac{N}{N'} \times \frac{Tv(N')}{Te(N')}} .$$

Its clear that as the problem size increases a larger proportion of nodes should be allocated to the *vf2* processes which have a higher computational complexity. Thus, as N increases, k decreases. As a first pass we can estimate the ratio of Tv(N') and Te(N') empirically from the measured execution times of the sequential version of *md3* from figure 3. Tv(N') and Te(N') are equal at approximately N'=400, resulting in:

$$k_{opt} = \frac{1}{1 + \frac{N}{400}} \quad .$$

For the parallel *md3* program, we can run all the combinations of possible node allocations on a 16 node Pentium cluster for a few problem sizes and measure the execution times. Thus for each problem size, N, we can determine the node allocation, k_{opt}, which yields the minimum execution time. Substituting the measured values of N and k_{opt} back into the equation above we can then solve for the ratio Tv(N')/(N'*Te(N')). For a given problem size, the measured value for k_{opt} results in a small range of values, shown in figure 6, rather than a single value because fractional nodes are not allocated and a minimum of 2 nodes are allocated to both ewald and vf2. Averaging these values yields:

$$k_{opt} = \frac{1}{1 + \frac{N}{360}} \quad .$$

The actual value of the constant is not critical, the general form of the equation is much more critical since it relates the computational complexity of the two processes. Any value in the neighborhood would suffice for the constant, since it is subject to round off due to integer node allocation as well as normal execution time variations between processes.

4.2 Dynamic Load Balancing

This initial static node allocation provides a good approximation to balancing the load, but variations occur in execution times between nodes even in homogeneous environments. These variations can be due to background workloads or even to a different configuration or version of the same machine. Incorporating a dynamic load balance adaptive mechanism in the code could compensate for these variations. Since each process exchanges data with the main control program at the end of each step, the execution time of this last step could be added to that data from each node. The main control program could then dynamically determine the actual load balance achieved on this last step and send out adjustment information for the next step along with the normal particle information. This adjustment would direct a process to increase or decrease its portion of the data it is processing, resulting in an uneven workload distribution where slower nodes could be directed to do less processing while faster nodes could be directed to do more.

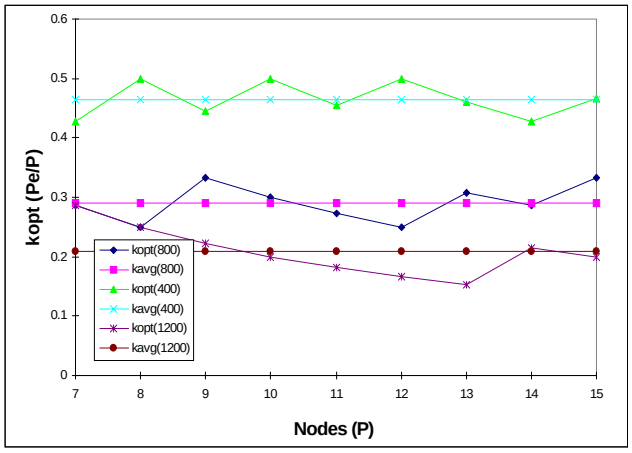


Figure 6: Kopt vs Allocatable Nodes for N=400 and 800

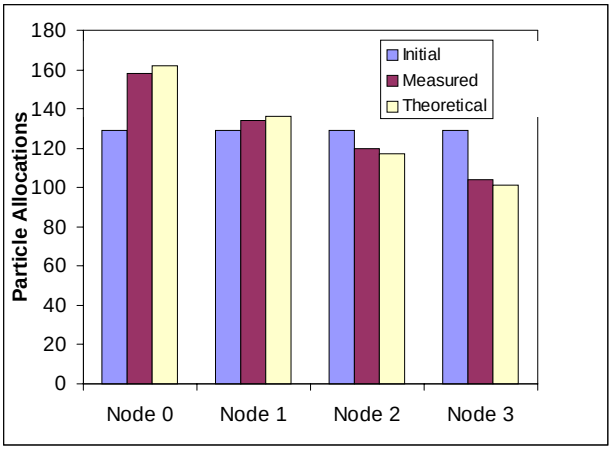


Figure 7: Kopt vs Allocatable Nodes for N=400 and 800

The adaptive algorithm we implemented is invoked at the end of each step and focuses on only one pair of nodes, the slowest and the fastest, for *ewald* and one pair for *vf2*. If the difference in execution time between these two nodes is below a threshold value, then no correction is applied. If the difference is above this threshold value then a correction in proportion to the difference between their execution times is applied. The faster node is directed to process n additional particles, while the slower node is directed to process n fewer particles. n is computed as follows:

$$n = Ns \times (Tf - Ts)/(2*Ts) \text{ for } ewald$$
$$\text{and} \quad n = Ns \times (Tf - Ts)/(4*Ts) \text{ for } vf2$$

where:

- Ns is the number of particles processed by the
- slow node during the previous step,
- Tf is the execution time of the fastest node,
- Ts is the execution time of the slowest node.

To test the effectiveness of this dynamic adaptive algorithm we simulated a simple case of uneven execution times for a few *ewald* processes. At the end of each step, we added different delays (via a sleep function) to each *ewald* process as follows:

- Node 0 is the fastest, no delay added,
- Node 1 is 10% slower than node 0,
- Node 2 is 20% slower than node 0,
- Node 3 is 30% slower than node 0.

Figure 7 shows the initial uniform data distribution and the measured and theoretical data distribution after ten steps. Figure 8 shows the load balance achieved during those ten steps. The load imbalance, initially 40%, is reduced to 10% in four steps and to less than 5% by ten steps.

5 PERFORMANCE

The performance of the parallel and the sequential (1 node) implementations of *md3* are compared in figure 9 for a number of different machines. It is clear that the Cray C90 is fastest for the sequential implementation, but the SGI is not far behind. With 8 nodes, using the k_{opt} allocation of nodes, all three machines, both the Pentium and SGI clusters and the IBM SP2, outperform the Cray; both clusters outperform the SP2 and the SGIs outperforms the Pentiums. Using 16 nodes the performance difference between the SP2 and the Pentium cluster narrows, but the Pentiums still outperforms the SP2.

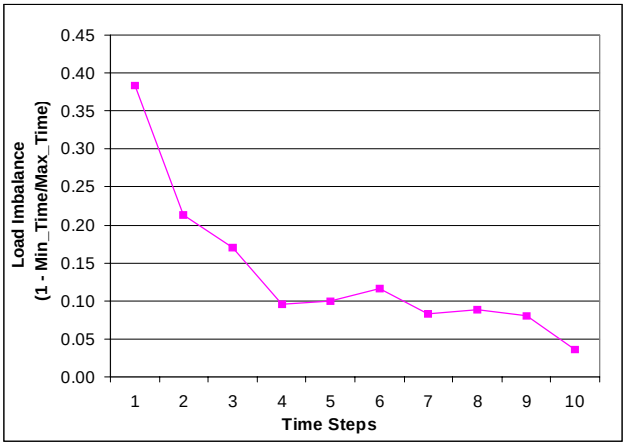


Figure 8: Dynamic Load Balance Algorithm Performance

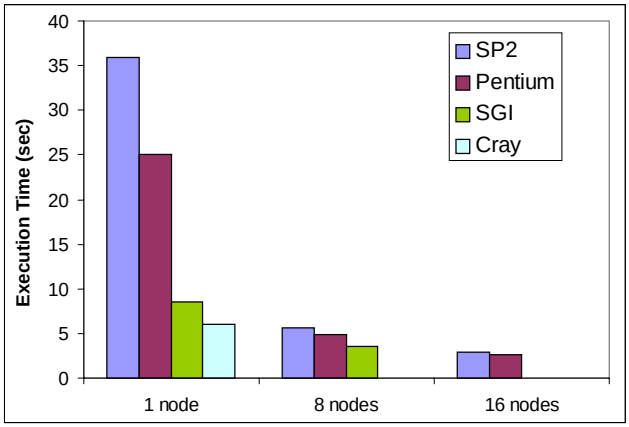


Figure 9: Comparison of Execution Time for 10 Processing Steps on Different Machines

The Gant charts shown in figure 10, produced using the Upshot performance visualization tool (available on the web from ftp://info.mcs.anl.gov), provides an indication of the load balance achieved through the use of the k_{opt} static allocation of nodes. In these charts Process 0 is the main control program, Processes 1-3 are the *ewald* processes, and Processes 4-7 are the *vf2* processes, see figure 5. The light areas for processes 1-7 indicate computation, while the dark areas indicate communications. The reverse is true for process 0. We can see that the load balance between *ewald* and *vf2* processes is not perfect, since *vf2* processes finish faster than *ewald* processes and must wait longer to complete the synchronization at the end of each step. Because of integer node allocation we can only approximate the allocation specified by k_{opt}, which results in round off errors for small numbers of nodes. As the number of nodes increase this approximation gets better.

These Gant charts verify that the computation is fastest on the SGI processors and slowest on the SP2 processors. They also show that the communication is superior on the SP2 compared to the commodity ATM network supporting the SGI and Pentium clusters. This is due to the higher bandwidth interconnect of the SP2 as well as the optimized implementation of MPI.

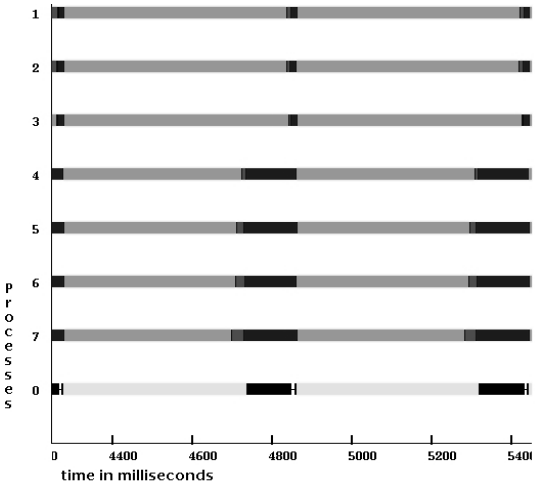
6 CONCLUSION

The *md3* molecular dynamics simulation program benefited from its parallel implementation in both performance and scalability. Execution time was diminished by 50% on a 16 node Pentium cluster compared to the original sequential code on a Cray C90. The parallel implementation included the ability to scale the problem size. This feature that could be added to the sequential code, but the collective memory of the Pentium cluster supports a larger problem size than can fit on the Cray. These results indicate that other molecular dynamics simulation applications of this class could also benefit from similar parallel implementations.

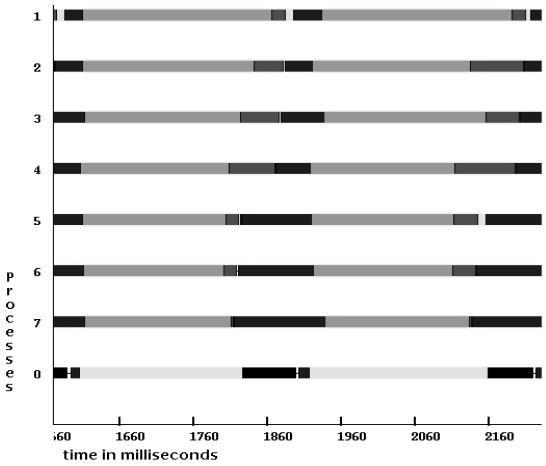
The use of MPI provides ease of portability between machines that support a message passing environment. It allows us to run and compare the performance of this parallel implementation on our Pentium cluster, a similar SGI cluster, and an IBM SP2. The results of this comparison show that inexpensive commodity PC clusters provide good performance when compared to more expensive machines like the Cray C90 and the IBM SP2 for this class of molecular dynamics simulation applications. The economy, however, applies only to the initial purchase cost of these commodity clusters, because they do require significant and expensive support.

ACKNOWLEDGMENTS

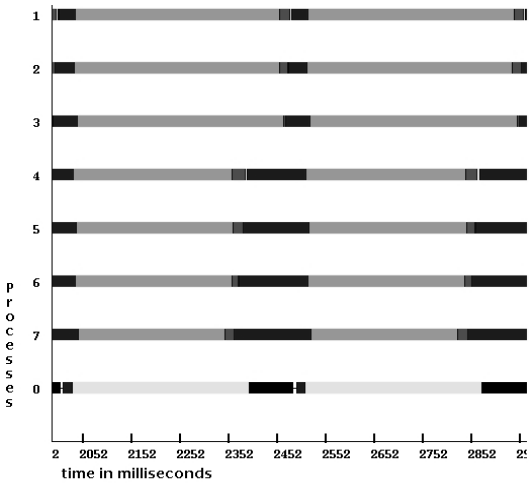
Thanks to Ray Mountain from the NIST Chemical Science and Technology Laboratory for providing the initial sequential *md3* code running on the Cray C90.



(a) IBM SP2 Cluster



(b) SGI Cluster



(c) Pentium Cluster

Figure 10: Snapshot of Execution and Communication Intervals of the *md3* Program on (a) an IBM SP2 Cluster, (b) an SGI Cluster and (c) a Pentium Cluster

REFERENCES

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

Burns, G. D., et al, *LAM: An Open Cluster Environment for MPI* , Proceedings of Supercomputing ’94 Symposium, Toronto, Canada (June 1994).

Morreale, P. Chapter 9 - *A UNICOS CPU Optimization Primer*, UNICOS guide, version 1.0 at http://www.scd.ucar.edu/archives/docs/UNICOS/unicos-guide.toc, Scientific Computing Division, National Center Atmospheric Research (June 1994).

Pacheco, P. Chapter 12 - *Wrapping up, Programming with MPI* (Morgan Kaufmann Publishers), University of San Francisco (1997)

Sangster, M.J.L and M. Dixon. *Interionic Potentials in Alkali Halides and Their Use in Simulations of The Molten Salts*, Advances in Physics Journal, Vol. 25, Physical Laboratory, University of Reading, England (1976)

AUTHOR BIOGRAPHIES

ALAN MINK is project engineer of the Distributed Systems Technology project within the NIST Information Technology Laboratory. He holds a B.S. in Electrical Engineering from Rutgers University and a M.S. and Ph.D. in Electrical Engineering from the University of Maryland. His research interests include computer architecture and performance measurement. He is a member of the IEEE and ACM.

CHRISTOPHE BAILLY was a guest researcher at NIST while completing his 2nd Masters degree from the Institut National des Télécommunications in Evry, France. Prior to this he completed a year of specialization in distributed systems also at the Institut National des Telecomunications in Evry and preceding that he earned a combined Masters and Bachelors degree in Electronics and Computer Science from the engineering school EERIE in Nimes, France. He is currently with the Intelligent Networks department of Alcatel CIT in Paris, France. His research interests include image processing, object oriented languages and distributed operating systems.