The MOSIX Multicomputer Operating System for High Performance Cluster Computing*

Amnon Barak and Oren La'adan†

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel

Abstract

The scalable computing cluster at Hebrew University consists of 64 Pentium and Pentium-Pro servers that are connected by fast Ethernet and the Myrinet LANs. It is running the MOSIX operating system, an enhancement of BSD/OS with algorithms for adaptive resource sharing, that are geared for performance scalability in a scalable computing cluster. These algorithms use a preemptive process migration for load-balancing and memory ushering, in order to create a convenient multi-user time-sharing execution environment for HPC, particularly for applications that are written in PVM or MPI. This paper begins with a brief overview of MOSIX and its resource sharing algorithms. Then the paper presents the performance of these algorithms as well as the performance of several large-scale, parallel applications.

Keywords: Cluster computing, load-balancing, preemptive process migration, PVM

1 Introduction

Scalable Computing Clusters (CC), ranging from a cluster of (homogeneous) servers to a Network of (heterogeneous) Workstations (NOW), are rapidly becoming the standard platforms for executing demanding applications, high performance and interactive computing. The main attractiveness of such systems is that they are made of affordable, low-cost, commodity hardware, e.g. Pentium based Personal Computers (PC's), and fast LANs e.g. Myrinet [6]. Other advantages are that these systems are scalable, i.e., can be tuned to available budget and needs, and that they use standard software components, i.e., UNIX, PVM [8] and the MPI [10] parallel programming environments. Two examples of such systems are the UC Berkeley NOW project [1], which uses a cluster of SPARC based workstations connected by Myrinet, and NASA's Beowulf "pile-of-PCs" [14], that are connected by multiple Ethernet interface boards.

*This research was supported in part by the Ministry of Defense and the Ministry of Science.

†E-mail:{amnon,orenl}@cs.huji.ac.il WWW: http://www.cs.huji.ac.il/mosix

Unlike MPP's, which allow a single user per partition, CC's are geared for multi-user, time-sharing environments. In order to make CC systems as easy to program, manage, and use as an SMP, it is necessary to develop means for global (cluster-wide) resource allocation and sharing that can respond to resource availability, distribute the workload dynamically and utilize the available, cluster-wide resources efficiently and transparently. Such mechanisms are necessary for performance scalability in clusters of servers and to support a flexible use of workstations, since the overall available resources in such systems are expected to be much larger than the available resources at any workstation or server. The development of such mechanisms is particularly important to support multi-user, time-sharing parallel execution environments, where it is necessary to share the resources and at the same time distribute the workload dynamically, to utilize the global resources efficiently. Throughout this paper we use the term Computing Cluster (CC) to refer to a collection of UNIX computers (nodes), ranging from a network of workstations to a cluster of servers that are connected by a local area network.

The main techniques for dynamic work distribution are load-balancing and load-sharing, that use some form of remote execution and process migration. These techniques can perform initial distribution of processes among the nodes and redistribution of the work load when the system becomes unbalanced.

One mechanism that can perform dynamic work distribution efficiently and transparently, is a preemptive process migration. This mechanism can move a process from one node to another dynamically, for load-balancing or memory ushering, to improve the performance and the overall utilization of the CC resources. We use the term "memory ushering" to describe the use of remote memory resources to avoid local paging and thus provide processes with the memory resources they require. In spite of the advantages of process migration, very few operating systems have implemented it for adaptive resource sharing. Instead, most existing systems provide explicit, user-controlled remote execution and migration, while other systems provide automated remote executions, but perform preemptive process migration at the explicit request of a user with a limited functionality.

This paper presents the CC MOSIX, an enhancement of BSD/OS [5, 12] that supports preemptive process migration for load-balancing and memory ushering in a cluster of PC's. These algorithms attempt to improve the overall performance, by dynamic distribution of the workload and the available resources among all the processes. The goal is that the application programs do not have to know the current state of the resource usage. Parallel applications can be executed by simply creating many processes, just like a single-machine environment. The paper presents the performance of these algorithms and the performance of several, large scale parallel applications.

The paper is organized as follows: the next section gives an overview of MOSIX and its main characteristics. Section 3 presents the performance of its communication protocol and the resource sharing algorithms. Section 4 presents the performance of several, large scale parallel applications. Our conclusions are given in Section 5.

2 Overview of MOSIX

MOSIX [4] is a set of enhancements of BSD/OS [5] with algorithms for adaptive resource sharing. These algorithms are geared for efficient resource utilization among the (homogeneous) nodes of a distributed-memory, shared-nothing scalable CC, including LAN connected networks of workstations and servers. This section describes possible hardware configurations for MOSIX and its main software characteristics.

2.1 Hardware configurations

MOSIX is designed to run on clusters of Pentium based workstations, PC's, file and CPU servers that are connected by standard LANs or fast interconnection networks. Depending on the type of applications and the budget, MOSIX configurations may range from a small cluster of PC's that are connected by Ethernet, to a high performance system, with a large number of high-end, Pentium based servers that are connected by a Gigabit/Sec. scalable LAN, e.g. Myrinet [6].

The main advantage of the above configurations is the use of standard, low-cost, off-the-shelf, commodity hardware components. For example, one multicomputer that we use has 32 Pentium-Pro 200MHz based servers that are connected by Fast Ethernet and Myrinet. At a cost of less than $100,000, this system delivers almost 2 GigaFLOPS, it has over 8GB of main memory and 50GB of disk space.

2.2 The MOSIX software enhancements

MOSIX is a set of enhancements of BSD/OS with adaptive resource sharing algorithms and a mechanism for preemptive process migration that are geared for efficient cluster computing. These algorithms are designed to respond dynamically to variations in resource usage among the nodes, by migrating processes from one node to another, preemptively and transparently, to improve the overall performance. The granularity of the work distribution in MOSIX is the UNIX process. Users can run parallel applications by initiating multiple processes in one node, then allow the system to assign these processes to the best available nodes at that time. If during the execution of the processes other resources become available, then the MOSIX algorithms are designed to utilize these new resources, by possible reassignment of the processes among the nodes. The ability to assign and then reassign processes is particularly important for "ease-of-use" and to provide an efficient multi-user, time-sharing execution environment.

In MOSIX, each user interacts with the multicomputer via the user's "home" node (workstation or server), which is similar to the "home node" of Sprite [7]. The system image model is a computing cluster, in which all the user's processes seem to run at the home node, and all the processes of each user have the execution environment of the user's home node. Processes that migrate to other (remote) nodes use local (in the remote node) resources whenever possible, but interact with the user's environment through the user's home node. As long as the requirement for resources such as the CPU or main memory are below certain threshold levels, all the user's processes are confined to the user's home node. When these requirements exceed the threshold levels, e.g., the load created by one CPU bound process or the size of the local memory, then some processes are migrated to other nodes. The overall goal is to maximize the performance by efficient utilization of the network-wide resources.

The MOSIX enhancements are implemented in the BSD/OS kernel, without changing its interface, and they are completely transparent to the application level. Its main characteristics are:

- *Probabilistic information dissemination algorithms* - that provide each node with sufficient knowledge about available resources in other nodes, without polling or further reliance on remote information. Each node sends, at regular intervals, information about its available resources to a randomly chosen subset of nodes. At the same time it maintains a small buffer (window) with the most recently arrived information. The use of randomness supports scaling, even information dissemination and dynamic configurations.

- *Preemptive process migration* - that can migrate any user's process, any time, to any available node, transparently. The cost of the process migration includes a fixed cost, to establish a new process frame in the remote site, and an additional cost, proportional to the number of pages copied. In practice, only the page table and the dirty pages of the process are copied.

- *Dynamic load-balancing* - that continuously attempts to reduce the load differences between pairs of nodes by migrating processes from over-loaded nodes to less loaded nodes. This scheme is decentralized – all of the nodes execute the same algorithms, and the reduction of the load differences is performed independently by pairs of nodes. The load-balancing algorithm responds to changes in the loads of the nodes, the runtime characteristics of the processes, and the number of nodes. This algorithm prevails as long as there is no extreme shortage of other resources, e.g., free memory.

- *Memory ushering* - by a memory depletion prevention algorithm that is geared to place the maximal number of processes in the "network RAM" across all the nodes, to avoid as much as possible processes thrashing or the swapping out of processes. The algorithm is triggered when a node starts excessive paging due to insufficient free memory. In this case the algorithm overrides the load-balancing algorithm and it attempts to migrate a process to a node which has sufficient free memory, even if this migration results in an uneven load distribution.

- *Efficient kernel communication* - that was specifically designed to reduce the overhead of the system's kernel communications, e.g. between the process and its home node, when it is executing in a remote site, and for process migration. The new protocol is geared for a locally distributed system, e.g.,it does not support general inter-networking such as routing, and it assumes a relatively reliable media. The result is a fast, reliable datagram protocol with low startup latency and high throughput.

- *Decentralized control and autonomy* - each node is capable of operating as an independent system, i.e., it makes all its own control decisions independently, and there is no master-slave relationships between the nodes. This organization allows a dynamic configuration, where nodes may join or leave the network with minimal disruptions.

- *Scaling considerations* - that ensure that the system runs as well on large configurations as it does on small configurations. The main considerations include symmetry, and the use of randomness in the system control algorithms. Each node bases its decisions on partial knowledge about the state of the other nodes and it does not even attempt to determine the overall state of the cluster or any specific node.

The most noticeable properties of executing parallel applications on MOSIX are its adaptive resource distribution policy, and the symmetry and flexibility of its configuration. The combined effect of these properties is that application programs do not need to know the current state of the system configuration, nor be concerned about the use of the resources in the various nodes. Parallel applications can be executed by creating many processes, just like in a single-machine system.

3 Performance of the MOSIX algorithms

This section presents the performance of the communication protocol, the process migration mechanism, the load-balancing and the memory ushering algorithms of MOSIX. The execution platforms were two computing clusters, with 16 identical Pentiums and 32 Pentium-Pro servers that were connected by fast Ethernet and the Myrinet LANs.

3.1 Performance of the TCP/IP protocol for the Myrinet LAN

MOSIX uses an optimized TCP/IP protocol for Inter Process Communication (IPC) as well as for process migration. The performance of the IPC between two Pentium-Pro 200MHz servers connected by the Myrinet LAN are shown in Figure 1.

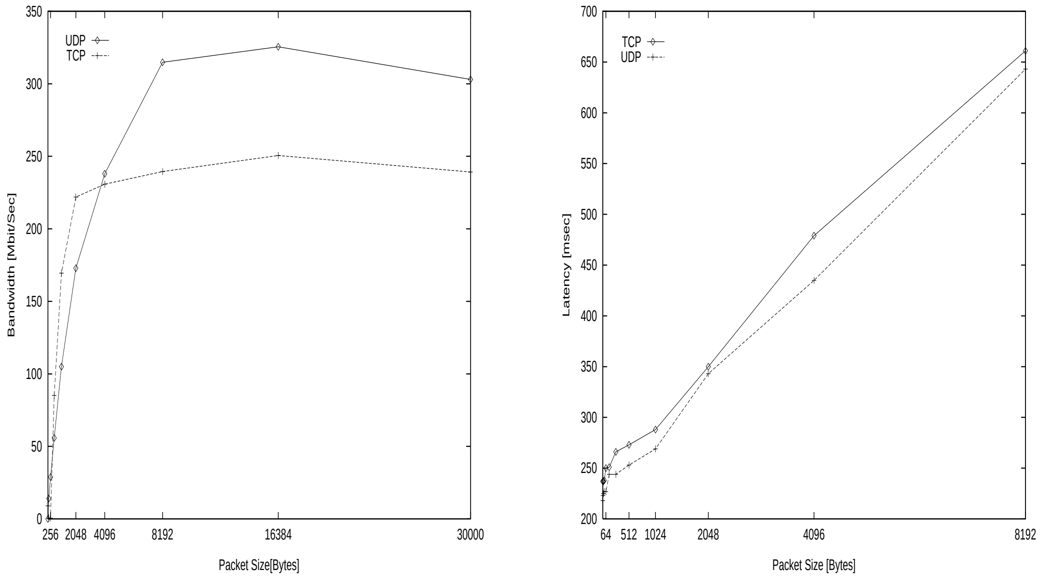

Figure 1: *Performance of the IPC over the Myrinet LAN*

Figure 1(a) presents the maximal IPC bandwidth of the Socket interface (API) via UDP and TCP for different packet sizes [9]. From the figure it follows that the maximal bandwidth of UDP and TCP are obtained for packet size of 16 KB. They are 325.2 Mbit/Sec. and 250.5 Mbit/Sec. respectively. The corresponding latencies are presented in Figure 1(b). Note that since the UDP protocol is much simpler than TCP, its bandwidth is better for packet sizes greater than 4K Bytes, and its latency is consistently better for all packet sizes.

3.2 Performance of the process migration mechanism

Process migration is carried out in two stages. The first stage involves establishing a connection and negotiations between two nodes and creating a process frame in the remote node. In the second stage the active memory of the process is transferred. The cost of the negotiations stage is fixed, while the data transfer cost is proportional to the amount of data transfered.

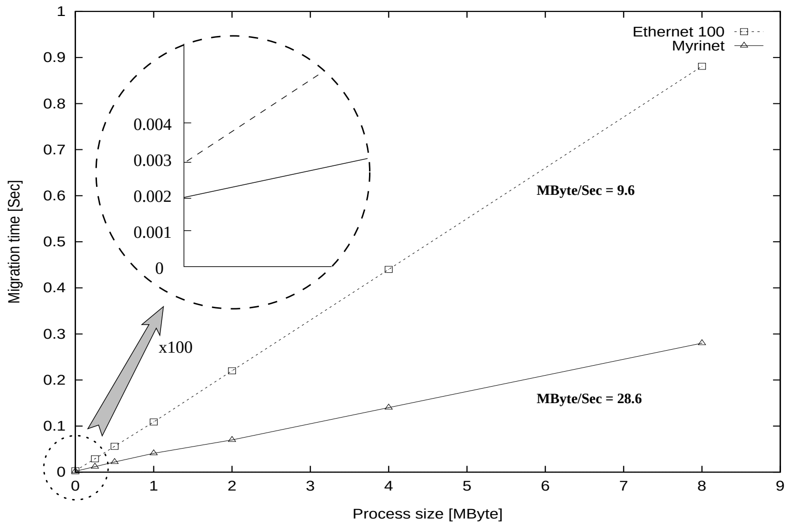

Figure 2: *Migration times vs. process size*

The performance of the process migration mechanism between two Pentium-Pro 200MHz servers that were connected by Ethernet-100 and the Myrinet LANs using TCP/IP, are shown in Figure 2. From the figure it can be seen that the migration time is a linear function of the process size. It amounts to 76.8 Mbit/Sec. for Ethernet-100 and 228.8 Mbit/Sec. for the Myrinet. The corresponding migration latencies (magnified) are 3ms and 2ms respectively.

3.3 Performance of the load-balancing algorithms

We conducted two tests to highlight the advantages of load-balancing by preemptive process migration. We measured the total execution times of a set of identical CPU-bound processes under PVM, with and without the MOSIX load-balancing algorithms [3], using a 16 node Pentium CC. Note that the process migration mechanism can migrate any user process, including processes that were initially been assigned to nodes by PVM.

The first benchmark was executed on a system with a background load. This background load reflects processes of other users, in a typical time-sharing, multi-user environment. It was generated by a set of additional CPU-bound processes that were executed in cycles of a random computing period followed by an idle (suspend) period. The results of this benchmark are shown in Figure 3(a). These results show that the average slow-down of PVM vs. MOSIX is over 35%, with as much as 62% slow-down, in the measured range, for 20 processes.

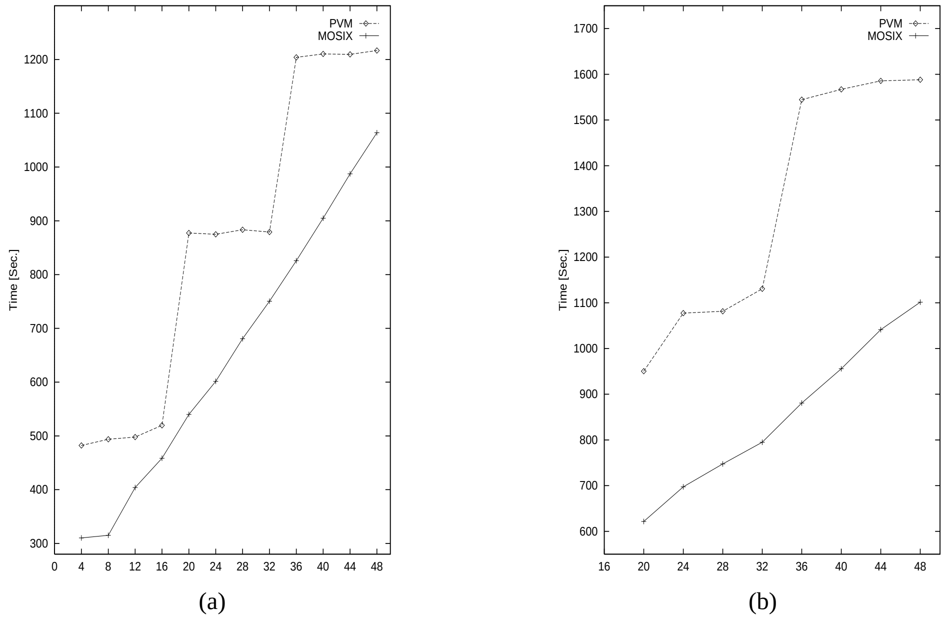

Figure 3: MOSIX vs. PVM: (a) with background load, (b) random execution times

A second benchmark, shown in Figure 3(b), presents the execution of parallel programs with unpredictable execution times, e.g. due to recursion, different amount of processing, etc., which are difficult to pre-schedule. We ran a set of CPU-bound processes that were executed for random durations, in the range 0 – 600 seconds. From the measurements it follows that the average slow-down of PVM vs. MOSIX is over 52%, with as much as 75% slow-down for 36 processes. Note that in a system with processors of different speeds, this slow-down can reach hundreds of percents.

3.4 Performance of the memory ushering algorithm

The memory ushering algorithm allows a node that has exhausted its main memory to use available free memory in other nodes, by migrating processes instead of paging or swapping to local disks [2]. The memory ushering algorithm is most useful in cases when memory is not being used uniformly or when nodes have different amounts of memory. Another scenario where this algorithm would be useful is when a large process is created, so that it can not fit into the free memory of any node. However, this process could fit into some of the nodes providing that the currently running processes are migrated away from these nodes.

The specific benchmark represents cases where the load of the nodes is balanced, but memory is not being used uniformly. In each execution, a set of processes with size distribution e^{-x} and average size of 8MB, was created and assigned to the nodes using PVM, such that some nodes run out of free memory. Each execution included a set of processes with a total size equal to the cluster-wide available memory (110MB per node). The benchmark was executed on clusters ranging from 8 to 32 nodes. For all cluster sizes we executed the same number of processes per node, thus ensuring the same load per node. In each case we executed the same set of processes twice using PVM with and without MOSIX.

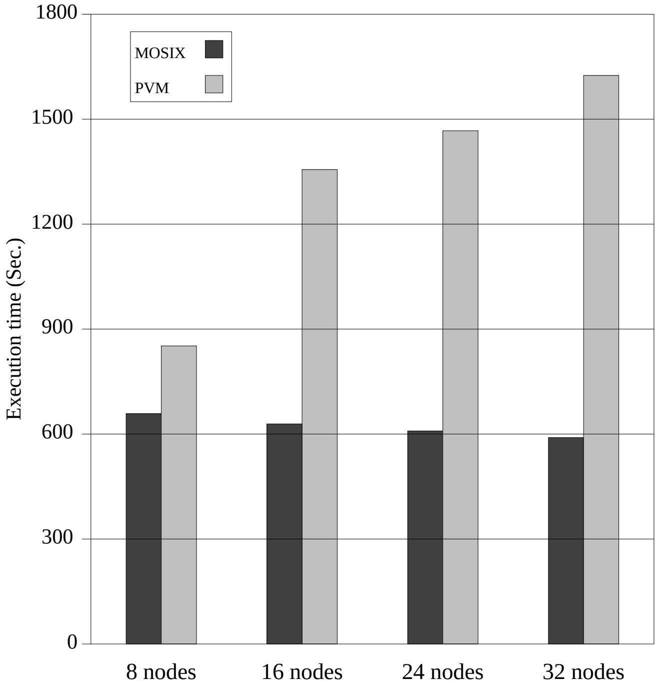

Figure 4: *Performance of the memory ushering algorithm*

The measured times of the executions are shown in Figure 4, where each bar represents the average measurement of five executions. From the measurements it follows that PVM without the process migration is 29% to 175% slower than PVM with the MOSIX process migration for 8 to 32 nodes respectively. These results prove the usefulness of the memory ushering and load-balancing algorithms.

Another interesting observation from Figure 4 is the effects of scaling. Although, for each cluster size we executed the same number of processes per node, it can be seen that the execution times of PVM monotonically increase by an average of 3.8% per node, relatively to the results obtained for 8 nodes. The corresponding times of MOSIX consistently decrease by 0.4% per node, when the cluster is scaled up from 8 to 32 nodes. These results show that the memory ushering and load-balancing algorithms of MOSIX are scalable, and indicate that MOSIX could perform well on much larger clusters.

4 Performance of parallel applications

This section presents our experience with the execution of several, large scale, parallel applications. The executions were carried on a MOSIX cluster with 32 Pentiums that were connected by fast Ethernet and 32 Pentium-Pro 200MHz that were connected by fast Ethernet and the Myrinet.

4.1 Global self organization of all known protein sequences

This project is a study of sequences of proteins. Unlike other projects which use algorithms for pairwise sequence comparisons and "nearest neighbors" methods for a small number of sequences, this project deals with the universe of all protein sequences, by translating the space of these sequences into an Euclidean space. Then a statistical, hierarchical clustering model is constructed. It offers additional insight into the large-scale organization and representation of the space of all protein sequences, and also reveals significant biological signatures of protein sequences [11].

The clustering algorithm associates each data point with the nearest centroid, where the centroids are re-estimated to minimize the distortion within each cluster. This process is repeated until convergence to a (local) minimum of the distortion. At each iteration, the cluster of highest aspect ratio is split. The algorithm is performed on two randomly chosen subsets of the data, aborting every split on which the two processes "disagree". The process terminates when all attempted splits get aborted.

The clustering algorithm was executed on a 32-node MOSIX configuration. The input data consisted of about 540,000 points, in a 200-dimension Euclidean space, that were initially split among several processes. In the execution, each process made the association of its (local) data points, and then calculated the first two moments (mean and covariance). For synchronization, a "master" process collected the results from all the "slaves", re-estimated the centroids and redistributed them. Since the algorithm is computational intensive, with only a small amount of communication, the speedup obtained was proportional to the number of nodes. The total amount of memory was about 1GByte. The longest execution time was 20 days, i.e., over 15,000 hours, during which we executed a case with 150 clusters.

Observations: this is a straightforward example of a parallel application that can benefit from the multiplicity of resources, e.g. CPU cycles and memory of a CC. The main MOSIX advantage is its ability to execute a memory intensive application for 20 days, without swapping, while at the same time allowing free access to the CC by other users.

4.2 Quantum simulations of large molecules

This project involves the development of novel parallel algorithms for quantitative simulations of large biological molecules, by computing the multidimensional, quantum wave-functions of these systems and studying their spectroscopic and dynamical properties. The algorithms use classical quantum mechanics methods to solve the time-independent and the time-dependent Schrödinger equations for systems of high dimensionality. The method uses as a first approximation a separation of variables, so that different degrees of freedom are handled by different processes. This is an intensive compute process, which requires a large amount of CPU time. The method was recently applied to calculate the ground-state wave-function and excited states of the protein BPTI.

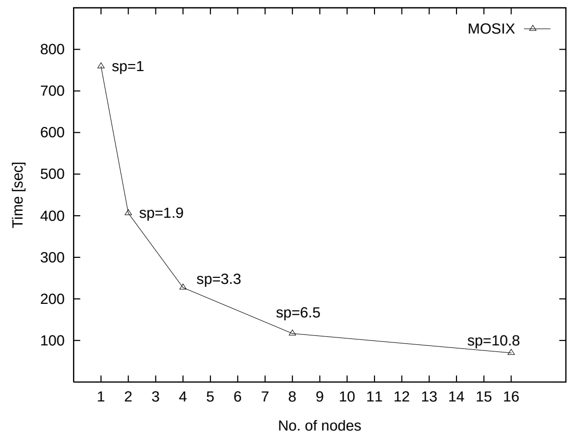

Figure 5: *Speedup (sp) obtained in a classical molecular dynamics simulation*

The specific application that was executed was parallel molecular dynamics on chemical systems with over 10,000 degrees of freedom, that simulate molecules of the order of complexity of proteins. The algorithm, which includes correlation effects (between the different degrees of freedom) was applied on Bacteriorhodopsin. Figure 5 shows the performance of this algorithm and the speedup obtained, using a 16 node Pentium-Pro 200MHz CC. The parallel algorithm uses a straight-forward spatial decomposition, with an internal mechanism for load-balancing between the different processes by modifications to the initial data decomposition. As the number of partitions increases, the communications overhead becomes very significant, since the CPU work of each partition decreases.

Observations: an intensive compute and communication bound parallel application can obtain a good speedup, with the increase in the number of nodes. In the current case, this speedup is impaired for 16 nodes due to the (relatively small) size of the problem.

4.3 Molecular dynamics simulation

Molecular Dynamics (MD) simulation has been used extensively as a tool to study irradiation damage. We describe two parallelization methods of large scale simulations.

4.3.1 High energy MD:

The first case is a physical system that consists of an energetic atom (in the range of 100 kev) impacting a surface, where a simulation for a large number of time steps and a large number $N > 10^6$ of atoms, is needed. The key issues are the different time scales involved and the number of particles.

The simulation applies a spatial decomposition into cubic cells. Most of the calculation is local, except the force calculation phase. In this phase each process needs data from all its 26 neighboring processes. The use of non-local potentials implied that this data communication became a major part of the execution. Hence, the decomposition attempted to minimize the surface to volume ratio. All communication was implemented using the PVM library.

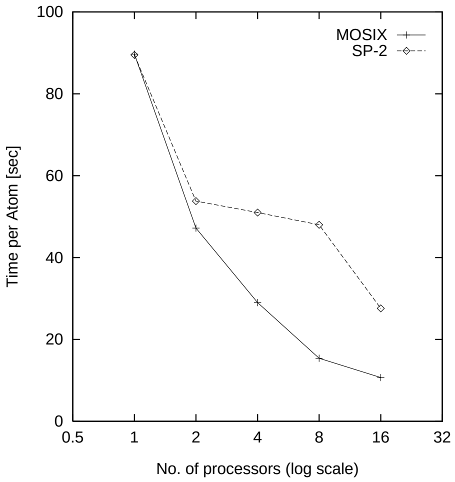

Figure 6: *MD performance of MOSIX vs. the IBM SP2*

The simulations were executed on a 16 node Pentium-Pro MOSIX CC, and the obtained results were compared with similar executions on the IBM SP2. The results of these executions are shown in Figure 6. From the figure it follows that MOSIX outperforms the SP2 by a factor of 2.6 for 16 nodes. The corresponding factors for 4 and 8 nodes are even higher. We suspect that in these last two cases, slower ("thin") nodes were used in the SP2. Note that in spite of the massive communication, MOSIX obtained a speedup of 8.4.

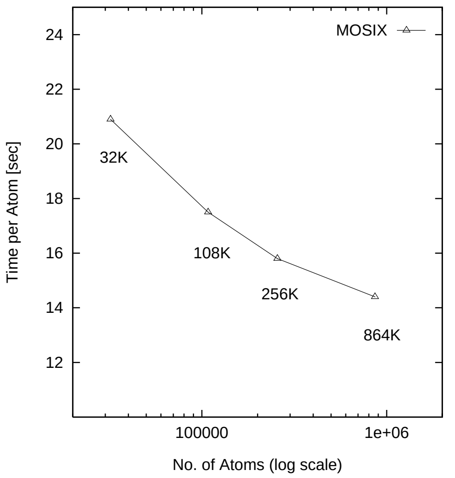

Figure 7: *MD scalability in problem size on MOSIX*

Figure 7 shows the scalability (in the problem size) on an 8 node MOSIX CC. Observe that the time per atom decreases with the increase in the problem size. This is due to a better initial load distribution and reduced communication overhead per atom. Since the simulation scales well in size, we expect that a large number (over 4 million) of atoms can be simulated on a MOSIX system with 32 nodes.

Observations: a cluster of Pentium-Pro 200MHz PC's connected by Myrinet can outperform the IBM SP2 for heavy applications such as MD simulations, which are both CPU and communication intensive.

4.3.2 Integration using MD:

This problem consists of the statistical analysis of the desorption probability of an atom from a surface due to a nuclear stimulated desorption. Prediction of the spatial distribution of a desorbing atom requires simulations over a wide range of physical parameters. The time needed for each simulation varies considerably with the initial parameters. A single case takes about 40 minutes (average) on an "SGI" workstation, resulting in up to a month execution time for a simulation with 10^3 different initial conditions.

The parallelization consists of concurrent execution of a large number of independent simulations, using different initial conditions. When some processes end, others are created, as needed, until a sufficiently small variance in the observed variable was achieved. The main execution requirement was to keep the system load balanced, in spite of the unpredictable execution times and total number of the processes. The MOSIX system enabled to completely disregard this issue and relieved the program from all assignment related decisions.

Observations: The MOSIX load-balancing scheme simplifies the execution of jobs with unpredictable execution parameters, e.g., time and number of processes, especially in the presence of other users' processes.

4.4 Shell model study of heavy nuclei

This project uses an algorithm which is based on permutational group concepts for doing calculations in the nuclear shell model. The Drexel University Shell Model (DUSM) code is an implementation of this new approach for performing shell model calculations in multishell configurations with good isospin. This code is CPU intensive and it requires minimal usage of disk space and I/O, compared to other traditional Shell-Model codes. The advantages of the current algorithm is particularly apparent for large nuclei in the fp shell.

We developed a parallel version of DUSM (DUPSM) [13]. This version has two computational phases: building the Hamiltonian matrix, and the Lanczos diagonalization procedure. The use of the permutational symmetry group introduces extra (unconserved) labels with which to label the basis; this splits the Hamiltonian matrix into independent submatrices. These submatrices are assigned to different nodes, effectively giving a straight-forward domain decomposition in Hilbert space. The building of the Hamiltonian matrix uses the bulk of the CPU time in large calculations. This part of the code scales almost perfectly with the number of nodes. At the end of this phase each node has different number of matrix elements. These elements are then redistributed among the nodes so that each node has the same number of matrix elements. This redistribution is performed by an algorithm that imitates the MOSIX load-balancing algorithm. The Lanczos procedure, in the second phase, has also been parallelized using the same data decomposition. This phase requires massive IPC of Lanczos vectors and therefore it does not scale linearly.

The computations were performed for the nuclei ^{51}Sc and ^{52}Sc, with 11-12 nucleons in the fp shell. The actual executions were performed on a 32 node Pentium-Pro 200MHz MOSIX system. These calculations used sparse Hamiltonian matrices of order 50,000, requiring about 170 MBytes in each node. The execution time was about 1 hour for each matrix, where one calculation of the full spectra has about 20 such matrices. This calculation was repeated about 30-40 times for each nuclei.

Observations: This is an intensive CPU and Memory bound application. The MOSIX load-balancing scheme simplifies the redistribution of the work load, which results in an almost linear speedup. With additional nodes we can use DUPSM to execute heavier nuclei.

5 Conclusions

This paper presented the MOSIX multicomputer system for a scalable cluster of PC's, and its performance for the execution of several large parallel applications. The use of adaptive resource sharing algorithms implies that users do not have to know the current state of the resource usage of the various PC's, or even the number of PC's. Parallel applications can be executed by allowing MOSIX to assign and reassign the processes to the best possible nodes, almost like an SMP. The performance of the CC MOSIX shows a good utilization of the resources, relatively good speedups in a scalable configuration and competitive results vs. the IBM SP2.

The main outcome of the MOSIX project is that it is possible to build a low-cost, scalable CC from commodity components, such as PC's, UNIX and PVM, as an alternative to traditional mainframes and MPP's. The main advantage of MOSIX over other CC environments is its ability to respond at run-time to unpredictable and irregular resource requirements by many users, e.g., execution times, memory usage or the number of processes - MOSIX adapts well to all such cases. As such MOSIX offers a convenient, general-purpose environment for executing large scale, demanding sequential and parallel applications.

The MOSIX project is expanding in several directions. First, we are developing new competitive algorithms for adaptive resource management that can deal with different kind of resources, e.g., load, memory, IPC and I/O. We are also researching algorithms for network RAM, in which a large process can utilize available memory in several nodes. We are also developing extensions to JAVA for supporting adaptive object migration, similarly to the MOSIX algorithms. Beyond that we are considering to port MOSIX for LINUX.

The BSD/OS version of MOSIX has been operational for over 4 years. It is used for student course work, research of operating systems for scalable CC, and the development of parallel applications. A 6 processors version, called MO6, is available for experimentation at URL `http://www.cs.huji.ac.il/mosix`.

Acknowledgments

We are grateful to A. Shiloh for his valuable contributions, and to Y. Ashkenazy, A. Braverman, E. Fredj, B. Gerber, I. Gilderman, I. Kelson , A. Novoselsky, G. Yona and M. Vallières for their contributions.

References

[1] T.E. Anderson, D.E. Culler and D.A. Patterson, A case for NOW (Networks of Workstations), *IEEE Micro* **15** (1) (1995) 54–64.

[2] A. Barak and A. Braverman, Memory ushering in a scalable computing cluster, *Proc. IEEE Third Int. Conf. on Algorithms and Architecture for Parallel Processing* (Melbourne, 1997).

[3] A. Barak, A. Braverman, I. Gilderman and O. Laden, Performance of PVM with the MOSIX preemptive process migration, *Proc. 7-th Israeli Conf. on Computer Sys. & Software Eng.* (1996) 38–45.

[4] A. Barak, S. Guday and R.G. Wheeler, *The MOSIX distributed operating system, load balancing for UNIX*, Lecture Notes in Computer Science **672** (Springer-Verlag, Berlin, 1993).

[5] *BSDI internet server (BSD/OS 3.0) administrative notes*, BSDI (Colorado Springs, CO, 1997).

[6] N.J. Boden, D. Cohen, R.E. Felderman, A.K. Kulawik, C.L. Seitz, J.N.Seizovic, and W-K. Su, Myrinet: A gigabit-per-second local area network, *IEEE Micro* **15** (1) (1995) 29–36.

[7] F. Douglis and J. Ousterhout, Transparent process miration: design alternatives and the sprite implementation, *Software – Practice & Experience* **21** (8) (1991) 757–785.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM - Parallel Virtual Machine* (MIT Press, Cambridge, MA, 1994).

[9] I. Gilderman and A. Barak, Profiling the communication layers performance of the Myrinet gigabit LAN, Technical Report, The Hebrew University of Jerusalem (1997).

[10] W. Gropp, E. Lust and A.Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (MIT Press, Cambridge, MA, 1994).

[11] M. Linial, N. Linial, N. Tishby and G. Yona, Global self organization of all known protein sequences reveals inherent biological signatures, *J. Molecular Biology* **268** (1997) 539 - 556.

[12] M.K. McKusick, M.J. Karels, K. Bostic and J.S. Quarterman, *The design and implementation of the 4.4BSD operating system* (Addison-Wesley, Reading, MA, 1996).

[13] A. Novoselsky, M. Vallières and O. La'adan, Full fp shell calculation of ^{51}Ca and ^{51}Sc, *Physical Review Letters* (1997) to appear.

[14] D. Ridge, D. Becker, P. Merkey and T. Sterling, Beowulf: harnessing the power of parallelism in a pile-of-PCs, *Proc. IEEE Aerospace* (1997).