Performance of PVM with the MOSIX Preemptive Process Migration Scheme*

Amnon Barak, Avner Braverman, Ilia Gilderman and Oren Laden
Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel§

Abstract

With the increased interest in network of workstations for parallel and high performance computing it is necessary to reexamine the use of process migration algorithms, to improve the overall utilization of the system, to achieve high performance and to allow flexible use of idle workstations. Currently, almost all programming environments for parallel systems do not use process migration for task assignments. Instead, a static process assignment is used, with sub optimal performance, especially when several users execute multiple processes simultaneously. This paper highlights the advantages of a process migration scheme for better utilizations of the computing resources as well as to gain substantial speedups in the execution of parallel and multi-tasking applications. We executed several CPU and communication bound benchmarks under PVM, a popular programming environment for parallel computing that uses static process assignment. These benchmarks were executed under the MOSIX multicomputer operating system, with and without its preemptive process migration scheme. The results of these benchmarks prove the advantages of using preemptive process migrations. The paper begins with an overview of MOSIX, a multicomputer enhancement of UNIX that supports transparent process migration for load-balancing, and PVM. We then present the performance of the executions of the benchmarks. Our results show that in some cases the improvements in the performance of PVM with the MOSIX process migration can reach tens or even hundreds of percents.

Key words: Distributed systems, dynamic load-balancing, high performance systems, preemptive process migration.

* Supported in part by grants from the Ministry of Defense and the Ministry of Science and Arts.
§ E-mail: amnon@cs.huji.ac.il, http://www.cs.huji.ac.il/mosix
This paper appeared in the Proc. 7th Israeli Conf. on Computer Systems and Software Engineering, June 1996.
Copyright © 1996 IEEE Computer Society Press.

1 Introduction

With the increased interest in Network of Workstations (NOW) as an alternative to Massive Parallel Processors (MPP) for high performance and general purpose computing [1], it is necessary to reexamine the use of dynamic process migration to improve the overall utilization of the NOW and to allow flexible use of idle workstations. In traditional MPPs, process migration mechanisms were not developed due to their complexity and because in many cases the whole machine was used to run one application at a time. The operating systems of many MPPs supports static, single process allocation to each node, a simple scheme that is easy to implement and use but may result in poor performance.

In a NOW system, where many users need to share the resources, the performance of executing multiple processes can significantly be improved by process migrations, for initial distribution of the processes, to redistribute the processes when the system becomes unbalanced or even to relieve a workstation when its owner wishes so. One mechanism that can perform all these tasks is a preemptive process migration, which combined with load balancing can maximize the overall performance, respond to resource availability and achieve high degree of overall utilization of the NOW resources.

In spite of the advantages of process migration and load balancing, there are only few systems that support these services [2, 7, 9]. The main reason is the fact that most parallel programming environments are implemented above the operating systems and are geared to support heterogeneous configurations. For example, p4 [5], is a library of macros and routines for programming a wide range of parallel machines, including shared-memory and message passing systems. In p4, process allocation is pre-scheduled, using a configuration file that specifies the pool of hosts, the name of an object file to be executed, and the number of instances to start, on a per-machine basis. Dynamic process creation is limited to process spawning in the local host by a preassigned parent process.

This paper presents the performance of executing several benchmarks using PVM, with its static process assignment vs. PVM with the MOSIX preemptive process migration [2]. PVM [8] is a popular programming environment which lets users exploit collections of networked computers and parallel computers. Its main advantages are the support of heterogeneous networks and machines, dynamic process and virtual machine management, and a simple and efficient user interface library. The main disadvantages of PVM are its static assignment of tasks to hosts, which results in its inability to respond to variations in the load of the hosts, and its assumption that all the workstations are of the same speed. While static assignment may be acceptable in MPPs, where the nodes have the same speed and each node executes one task, it is unacceptable in a NOW environment, where the resources are shared by many users, the execution times of the tasks are not known *a priori*, and the machine configuration may change. In these cases, a static assignment policy might lead to a considerable degradation in the overall system utilization.

In order to highlight the potential speedup gains (loss) of PVM, we executed several benchmarks under PVM and the MOSIX operating system. MOSIX [3, 2] is an enhancement of UNIX that provides resource (memory, communication) sharing and even work distribution in a NOW, by supporting a preemptive process migration and dynamic load-balancing. The MOSIX enhancements are implemented at the operating system kernel, without changing the UNIX interface, and they are completely transparent to the application level. Executions under PVM, with its static allocation, in a configuration with hosts of different speeds resulted in a low utilization of the NOW, and speedups of tens, or even hundreds of percents, once a process migration is implemented.

Recently, a group at OGI developed MPVM [6], a process migration mechanism for PVM. Unlike the MOSIX implementation which is done at the operating system kernel, MPVM is implemented at the user-level, with its obvious limitations, e.g. relatively high migration costs. For example, process migration in MOSIX includes only the "dirty-pages" while in MPVM the entire virtual address space of the process is transfered. Another advantage of the MOSIX approach is its transparent process migration, which makes work distribution easier and achieve high overall utilization. Nevertheless, MPVM is an interesting development and we hope to compare its performance to that of MOSIX.

This paper is organized as follows: the next section presents an overview of MOSIX and its unique properties. Section 3 gives an overview of PVM. Section 4 presents the performance of several benchmarks of CPU bound processes under MOSIX, PVM and PVM with the MOSIX process migration. Section 5 presents the performance of communication bound processes. Our conclusions are given in Section 6.

2 The MOSIX Multicomputer System

MOSIX is an enhancement of UNIX that allows distributed-memory multicomputers, including LAN connected Network of Workstations (NOW), to share their resources by supporting preemptive process migration and dynamic load balancing among homogeneous subsets of nodes. These mechanisms respond to variations in the load of the workstations by migrating processes from one workstation to another, preemptively, at any stage of the life cycle of a process. The granularity of the work distribution in MOSIX is the UNIX process. Users can benefit from the MOSIX execution environment by initiating multiple processes, e.g. for parallel execution. Alternatively, MOSIX supports an efficient multi-user, time-sharing execution environment.

The NOW MOSIX is designed to run on configurations that include several nodes, i.e. personal workstations, file servers and CPU servers, that are connected by LANs, shared buses, or fast interconnection networks. In these configurations each node is an independent computer, with its own local memory, communication and I/O devices. A low-end configuration may include few personal workstations that are connected by Ethernet. A larger configuration may include additional file and/or CPU servers that are connected by ATM. A high-end configuration may include a large number of nodes that are interconnected by a high performance, scalable, switch interconnect that provides low latency and high bandwidth communication, e.g. Myrinet [4].

In MOSIX, each user interact with the multicomputer via the user's "home" workstation. The system image model is a NOW, in which all the user's processes seem to run at the home workstation. All the processes of each user have the execution environment of the user's workstation. Processes that migrate to other (remote) workstations use local resources whenever possible, but interact with the user's environment through the user's workstation. As long as the load of the user's workstation is light, all the user's processes are confined to the user's workstation. When this load increases above a certain threshold level, e.g. the load created by one CPU bound process, the process migration mechanism (transparently) migrates some processes to other workstations or to the CPU servers.

2.1 The Unique Properties of MOSIX

The MOSIX enhancements are implemented in the UNIX kernel, without changing its interface, and they are completely transparent to the application level, e.g. MOSIX uses standard NFS. Its main unique properties are:

- **Network transparency** - for all cross machine operations, i.e. for network related operations, the interactive user and the application level programs are provided with a virtual machine that looks like a single machine.

- **Preemptive process migration** - that can migrate any user's process, transparently, at any time, to any available node. The main requirement for a process migration is transparency, that is, the functional aspects of the system's behavior should not be altered as a result of migrating a process. Achieving this transparency requires that the system is able to locate the process and that the process is unaware of the fact that it has been moved from one node to another. In MOSIX these two requirements are achieved by maintaining in the user's (home) workstation, a structure, called the *deputy* [3], that represents the process and interacts with its environment. We note that the concept of the *deputy* of a process is based on the observation that only the system context of a process is site dependent. The migration itself involves the creation of a new process structure at the remote site, followed by a copy of the process page table and the "dirty" pages. After a migration there are no residual dependencies other than at the home workstation. The process resumes its execution in the new site by few page faults, which bring the necessary parts of the program to that site [3].

- **Dynamic load balancing** - that initiates process migrations in order to balance the loads of the NOW. The algorithms respond to variations in the loads of the nodes, the runtime characteristics of the processes, the number of workstations and their speeds. In general, load-balancing is accomplished by continuous attempts to reduce the load differences between pairs of nodes, and by dynamically migrating processes from nodes with a higher load to nodes with a lower load. The policy is symmetrical and decentralized, i.e., all of the nodes execute the same algorithms, and the reduction of the load differences is performed independently by any pair of nodes.

- **Memory sharing** - by memory depletion prevention algorithms that are geared to place the maximal number of processes in the main memory of the NOW, even if this implies an uneven load distribution among the nodes. The rational behind this policy is to delay as much as possible swapping out of pages or a whole process, until the entire, network wide main memory is used. The algorithms of the policy are activated when the amount of a workstation's free memory is decreased bellow a certain threshold value. The decisions of which process to migrate and where to migrate it are based on knowledge about the amount of free memory in other nodes that is circulated among the workstations. These decisions are geared to optimize the migration overhead.

- **Efficient kernel communication** - that was specifically developed to reduce the overhead of the internal kernel communications, e.g. between the process and its home site, when it is executing in a remote site. The new protocol was specifically designed for a locally distributed system. As such, it does not support general inter-networking issues, e.g. routing, and it assumes a reliable media. The result is a fast, reliable datagram protocol with low startup latency and high throughput. The protocol applies a "look ahead" packet acknowledgement scheme and run-time fine tuning in order to achieve near optimal utilization of the network media and the corresponding system resources.

- **Probabilistic information dissemination algorithms** - that are geared to provide each workstation with sufficient knowledge about available resources in other workstations, without polling or further reliance on remote information. The information gathering algorithms measure the amounts of the available resources at each workstation using suitable resource indices, which reflects the availability of the local resources to possible incoming processes from other workstations. The resource indices of each workstation are sent at regular intervals to a randomly chosen subset of workstations, by the information dissemination algorithm. The receiver algorithm maintains a small buffer (window), with the values of the most recently arrived index values and at the same time it flushs out older values. We note that the use of random workstation ID is due to scaling considerations, for even distribution of the information among the participating workstations, to support a dynamic configuration and to overcome partial (workstations) failures.

- **Decentralized control** - each workstation makes all its own control decisions independently and there are no master-slave relationships between the workstations.

- **Autonomy** - each workstation is capable of operating as an independent system. This property allows a dynamic configuration, where workstations may join or leave the network with minimal disruptions.

The most noticeable properties of executing applications on MOSIX are its network transparency, the symmetry and flexibility of its configuration, and its preemptive process migration. The combined effect of these properties is that application programs do not need to know the current state of the system configuration. This is most useful for time-sharing and parallel processing systems. Users need not recompile their applications due to node or communication failures, nor be concerned about the load of the various processors. Parallel applications can simply be executed by creating many processes, just like a single-machine system.

3 PVM

This section presents an overview of the Parallel Virtual Machine (PVM) [8]. PVM is an integral framework that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. The supported architectures include shared- and distributed-memory multiprocessors, vector supercomputers, special purpose computers, and workstations that are interconnected by a variety of networks. Below is a brief description of some aspects of PVM.

3.1 Heterogeneity

PVM supports heterogeneity at three levels: applications, machines and networks. At the application level, subtasks can exploit the architecture best suited for them. At the machine level, computers with different data formats are supported, including serial, vector and parallel architectures. The virtual machine can be interconnected via different networks, at the network level. Under PVM, a user-defined collection of computational resources can be dynamically configured to appear as one large distributed-memory computer, called "virtual machine"

3.2 Computing Model

PVM supports a straightforward message passing model. Using dedicated tools, one can automatically start up tasks on the virtual machine. A task, in this context, is a unit of computation, analogous to a UNIX process. PVM allows the tasks to communicate and synchronize with each other. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel. The model assumes that any task can send a message to any other PVM task, with no limit on the size or amount of the messages.

3.3 Implementation

PVM is composed of two parts. The first is the library of PVM interface routines. These routines provide a set of primitives to perform invocation and termination of tasks, message transmission and reception, synchronization, broadcasts, mutual exclusion and shared memory. Application programs must be linked with this library to use PVM. The second part consists of supporting software, that is executed on all the computers, that make up the virtual machine, called "daemon". These daemons interconnect with each other through the network. Each daemon is responsible for all the application components processes executing on its host. Thus, control is completely distributed, except one master daemon. Two crucial topics rise when discussing implementation issues: inter-process communications (IPC) and process control. These topics are discussed below.

3.3.1 Inter Process Communications

In PVM different daemons communicate via the network. PVM assumes existence of only unreliable, unsequenced, point-to-point data transfer facilities. Therefore, the required reliability as well as additional operations like broadcasts, are built into PVM , above the UDP protocol. For IPC, the data is routed via the daemons, e.g., when task A invokes a *send* operation, the data is transferred to the local daemon, which decodes the destination host and transfers the data to the destination daemon. This daemon decodes the destination task and delivers the data. This protocol uses 3 data transfers, of which one is across the network. Alternatively, a direct-routing policy can be chosen (depending on available resources). In this policy, after the first communication instance between two tasks, the routing data is locally cached (at the task). Subsequent calls are performed directly according to this information. This way, the number of data transfers is reduced to only one, over the network. Additional overheads are incurred by acknowledgment schemes and packing/unpacking operations.

3.3.2 Process Control

Process control includes the policies and means by which PVM manages the assignment of tasks to processors and controls their executions. In PVM, the computational resources may be accessed by tasks using four different policies: (a) a transparent mode policy, in which subtasks are automatically assigned to available nodes; (b) the architecture-dependent mode, in which the assignment policy of PVM is subject to a specific architecture constraints; (c) the machine-specific mode, in which a particular machine may be specified; and (d) a user's defined policy that can be "hooked" to PVM. Note that this last policy requires a good knowledge of the PVM internals.

The default policy used by PVM is the transparent mode policy. In this case, when a task initiation request is invoked, the local daemon determines a candidate pool of target nodes (among the nodes of the virtual machine), and selects the next node from this pool in a round-robin manner. The main implications of this policy are the inability of PVM to distinguish between machines of different speeds, and the fact that PVM ignores the load variations among the different nodes.

No. of Processes	Optimal Time	MOSIX Time	PVM Time	PVM Slow-down (%)	PVM on MOSIX
1	300	301.91	301.83	0.0	304.54
2	300	302.92	303.78	0.3	304.70
4	300	304.57	305.60	0.3	306.59
8	300	305.73	308.57	0.9	301.88
16	300	310.83	317.12	2.0	303.40
17	450	456.91	604.36	32.3	452.84
20	450	462.07	602.40	30.4	454.07
24	450	471.87	603.25	27.8	454.67
25	525	533.15	603.83	13.3	530.15
27	525	549.07	603.86	10.0	559.81
31	563	574.03	604.63	5.3	595.17
32	600	603.17	603.14	0.0	604.64
33	700	705.93	906.31	28.4	707.39
36	700	715.35	905.27	26.5	708.41
38	750	759.90	905.34	19.1	755.53
40	750	767.67	905.39	17.9	771.71
43	833	833.33	908.96	9.1	839.61
47	883	901.81	907.79	0.7	893.65
48	900	916.11	908.51	-0.8	907.71

Table 1. Optimal vs. MOSIX vs. PVM vs. PVM on MOSIX execution times (Sec.)

4 Performance of CPU-bound Processes

In this section we compare the performance of the execution of sets of identical CPU-bound processes under PVM, with and without process migration, in order to highlight the advantages of the MOSIX preemptive process migration mechanism and its load balancing scheme. Several benchmarks were executed, ranging from pure CPU-bound processes in an idle system, to a system with a background load. We note that in the measurements, process migration is performed only when the difference between the loads of two nodes is above the load created by one CPU bound process. This policy differs from the time-slicing policy commonly used by shared-memory multicomputers.

The execution platform for all the benchmarks is a NOW configuration, with 16 identical, Pentium-90 based workstations that were connected by an Ethernet LAN.

4.1 CPU-Bound Processes

The first benchmark is intended to show the efficiency of the MOSIX load balancing algorithms. We executed a set of identical CPU-bound processes, each requiring 300 seconds, and measured the total execution times under MOSIX (with its preemptive process migration), followed by measurements of the total execution times under PVM (without process migration), and then the execution times of these processes under PVM with the MOSIX process migration.

Table 1 summarizes the results of these benchmarks (all execution times are in seconds). In the table, the first column lists the number of processes. The second column lists the theoretical execution times of these processes using the optimal assignment algorithm with preemptive process migration and no communication overhead. Column three lists the measured execution times of the processes using the MOSIX load balancing algorithm. Column four lists the execution times of the same processes under PVM and column five gives the PVM slowdown, i.e. the ratio between column four and column three. Column six lists the corresponding execution times of the processes under PVM with the MOSIX load balancing.

By comparing columns 2 and 3 of Table 1, it follows that the average slow-down ratio of the MOSIX policy vs. the optimal execution algorithm is only 1.95% (consider that MOSIX imposes a minimal residency period of 1 Sec. for each new process before it can be migrated). Another result is that the execution times of PVM (forth column) can be significantly slower than PVM under MOSIX (sixth column). Observe that the initial allocation of PVM reduces the residency times imposed by MOSIX, as shown in column six.

Figure 1 depicts the results of Table 1. Comparison of the measured results shows that the average slowdown of PVM vs. MOSIX is over 15%, when executing more than 16 processes. This slowdown can become very significant, e.g. 32% for 17 processes and 28% for 33 processes. In contrast, the measurements show that PVM with the MOSIX process migration is slightly better than MOSIX itself, due to the residency period that is imposed by MOSIX.

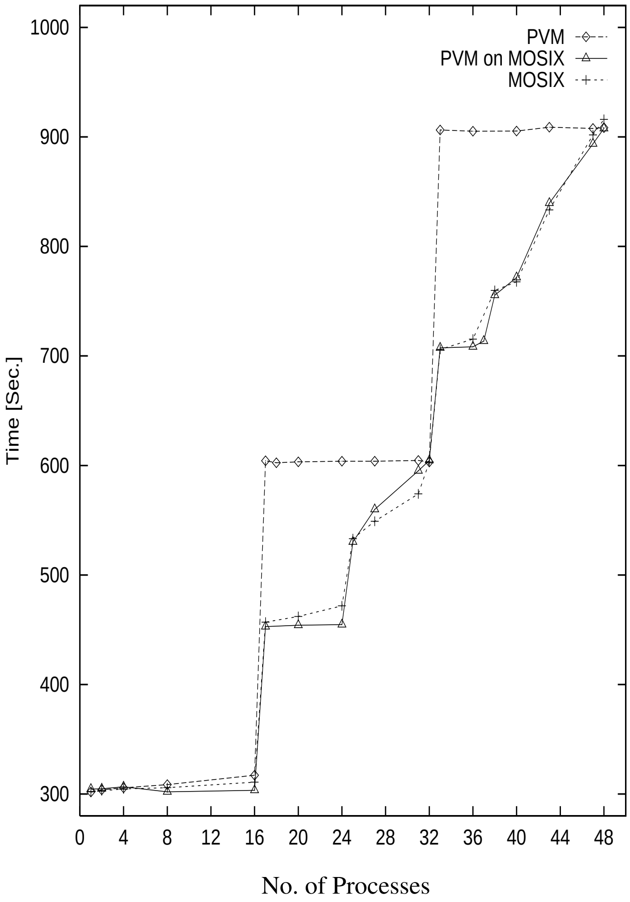

Figure 1. MOSIX, PVM and PVM on MOSIX execution times

As indicated earlier, one drawback of PVM is its inability to distinguish between machines of different speeds. To demonstrate this point, we executed the above set of processes on a cluster of Pentium-90 and several (three times slower) i486/DX66 based workstations. The results of this test show that PVM was 336% slower than MOSIX.

4.2 CPU-Bound Processes with Random Execution Times

The second benchmark compares the execution times of a set of CPU-bound processes that were executed for random durations, in the range 0 – 600 seconds, under MOSIX and PVM. These processes reflect parallel programs with unpredictable execution times, e.g. due to recursion, different amount of processing, etc., which are difficult to preschedule. In each test, all the processes started the execution simultaneously and the completion time of the last process was recorded. In order to obtain accurate measurements, each test was executed five times, with different random execution times. We note that the same sequence of random execution times were used in the MOSIX and the PVM executions.

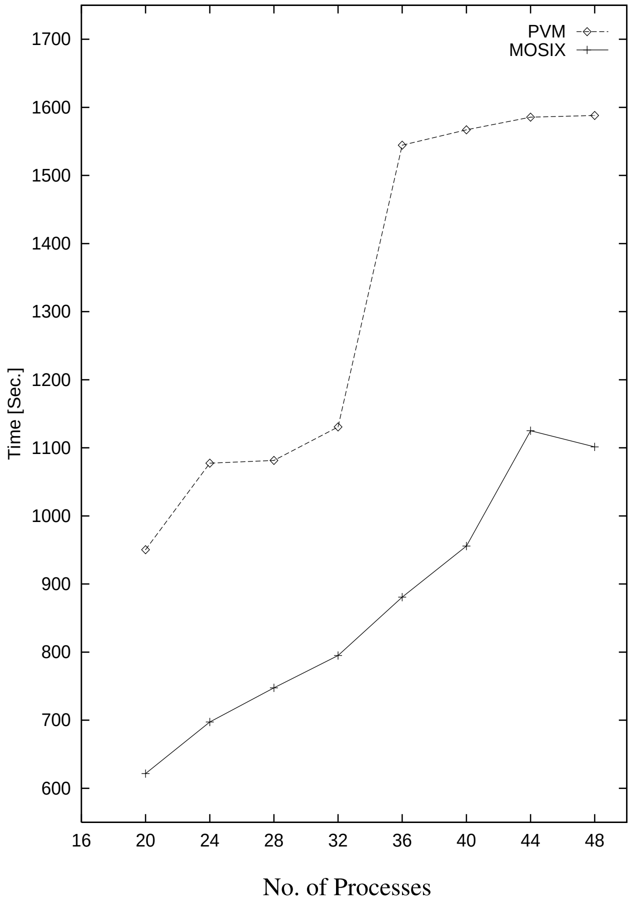

Figure 2. MOSIX vs. PVM random execution times

The results of this benchmark are presented in Figure 2, From the corresponding measurements it follows that the average slowdown of PVM vs. MOSIX is over 52%, with an averaged standard deviation of 13.9%. This slowdown reached as much as 75% for 36 processes, and over 600% when the above benchmark was executed on a cluster of Pentium-90 and i486/DX66 based workstations.

4.3 CPU-bound Processes with a Background Load

The third benchmark compares the execution times of a set of identical CPU-bound processes under MOSIX and PVM, in a system with a background load. This additional load reflects processes of other users in a typical time-sharing computing environment.

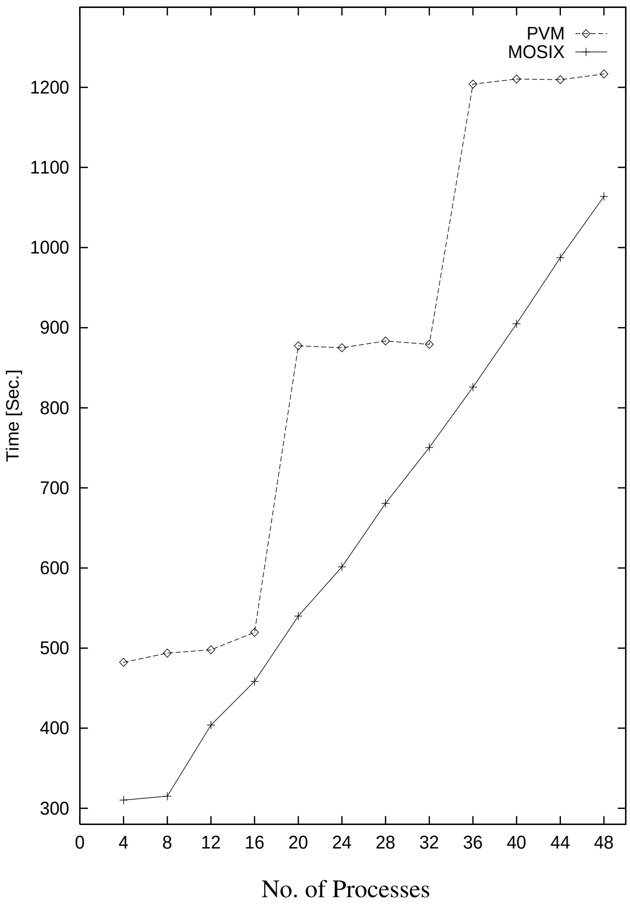

Figure 3. MOSIX vs. PVM with background load execution times

The specific background load consisted of 8 additional CPU-bound processes that were executed in cycles, where each cycle included an execution period followed by an idle (suspended) period. The background processes were executed independently, throughout the execution time of the benchmark, and the durations of the execution and suspension periods were random variables, in the range of 0 to 30 seconds. In order to get accurate measurements, each test was executed five times.

The results of this benchmark are presented in Figure 3. Comparison of the corresponding measured results shows that the average slowdown of PVM vs. MOSIX is over 35%, with as much as 62% slowdown, in the measured range, for 20 processes. From these measurements it follows that in a multi-user environment, when it is expected that background processes of other users are running, execution of parallel programs under PVM may result in a significant slowdown vs. the same executions with a preemptive process migration.

5 Communication Bound Processes

This section compares the performance of inter-process communication operations between a set of processes under PVM and MOSIX. First, we show the overhead of using the PVM communication layer by comparing the execution times of a set of identical communication bound processes, that were statically assigned to different nodes and were communicating along a ring topology. We note that this benchmark did not involve process migration and that both of the executions under PVM and MOSIX used standard Internet-domain sockets.

In the benchmark, in each iteration, each process sends and receives a single message to/from each of its two adjacent processes, then it proceeds with a short CPU-bound computation. In each test, 60 cycles were executed and the net communication times, without the computation times, were measured. Thus each measurement reflects the execution time of 240 one-way messages by each process.

The results of this benchmark, for message sizes of 1K bytes to 256K bytes, are shown in Table 2. From the table it can be seen that the MOSIX communication times are consistently better than PVM for almost all message sizes. This is due to the relatively complex protocols used by the PVM daemons, and the message handing mechanism that supports heterogeneity. Note that in few cases the PVM times are better than the MOSIX times. This can be accounted for better synchronization mechanisms of PVM.

No. of	1KB Messages		16KB Messages	
Processes	MOSIX	PVM	MOSIX	PVM
4	0.77	4.17	10.66	10.91
8	1.15	4.59	18.62	20.31
12	1.67	4.61	24.95	30.65
16	1.58	5.13	30.31	41.80
	64KB Messages		256KB Messages	
4	54.2	34.7	148.8	132.5
8	79.2	71.6	253.1	298.1
12	94.4	113.5	297.5	507.2
16	97.6	172.2	403.3	751.5

Table 2. MOSIX vs. PVM communication bound processes execution times (Sec.)

The next benchmark shows the overhead imposed by the MOSIX internal migration mechanisms over Unix domain IPC. In this test we executed a similar (to the above) set of communicating processes which were created in one machine and were forced to migrate out to other machines. We note that due to the use of the home model in MOSIX, processes that migrate to remote nodes, perform all their Unix domain IPC via their home sites. The main implication is a reduced communication bandwidth and increased latency due to possible bottlenecks at the home sites. For example, the communication time between two processes, one of which was migrated away from their common home site, was 10% slower than the communication time between two processes that did not have a common home site. The above overhead, of the two processes with the common home site, reached as much as 50% when both processes were migrated away.

The phenomenon presented in the previous paragraph may lead to a substantial communication overhead, when a large number of processes are created in one node, and later migrate to other nodes. To overcome this potential bottleneck, our current policy is to spawn communicating processes using PVM and then to refine the (static) PVM allocation by the MOSIX preemptive (dynamic) process migration.

6 Conclusions

In this paper we presented the performance of several benchmarks that were executed under MOSIX, PVM, and PVM with the MOSIX preemptive process migration. We showed that in many executions, the performance of PVM without the process migration was significantly lower than its performance with the process migration. We predict that in a typical multi-user environment, where each user is executing only a few process, users may loose hundreds of percents in the performance due to lack of preemptive process migration mechanisms, as discussed in [10]. We note that the choice of PVM was based on its popularity. We predict that the speedup ratios presented here characterize many other parallel programming environments that use static process assignments.

The NOW MOSIX is compatible with BSDI's BSD/OS [11], which is based on BSD-Lite from the Computer Systems Research Group at UC Berkeley. The current implementation has been operational for over 3 years on a cluster of 32 Pentiums and several i486 based workstations. It is used for research and development of multicomputer systems and parallel applications. Its unique mechanisms provide a convenient environment for writing and executing parallel programs, with minimal burden to the application programmers.

Currently we are researching the idea of migrateable sockets to overcome potential bottlenecks of executing a large number of communicating processes. We are also developing optimization algorithms for memory sharing, by using competitive, on-line algorithms to utilize available remote memory. Another area of research is optimization of the communication overhead by migrating communicating processes to common sites, to benefit from fast, shared memory communication.

After we install the Myrinet LAN [4], we intend to start several new projects that benefit from its fast communication speed. One project is to develop a *memory server* that can swap portions of a large program to "idle" memory in remote workstations. This mechanisms could benefit from our process migration mechanism, that is capable to page across the network. This project is similar to the network RAM project described in [1]. Another project is to develop a shared memory mechanism based on network RAM and process migrations.

Finally, we note that a limited (up to 6 processors) version of MOSIX, called MO6, is available on the Internet: WWW: http://www.cs.huji.ac.il/mosix. MO6 allows users of BSD/OS to build a low-cost, distributed memory multicomputer.

References

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

[2] A. Barak, S. Guday, and R. Wheeler. The MOSIX Distributed Operating System, Load Balancing for UNIX. In *Lecture Notes in Computer Science, Vol. 672*. Springer-Verlag, 1993.

[3] A. Barak, O. Laden, and Y. Yarom. The NOW MOSIX and its Preemptive Process Migration Scheme. *Bulletin of the IEEE Technical Commitee on Operating Systems and Application Environments*, 7(2):5–11, Summer 1995.

[4] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J.N.Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[5] R. Butler and E. Lusk. User's Guide to the p4 Programming System. Technical Report TM-ANL-92/17, Argonne National Laboratory, October 1992.

[6] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM. *Computing Systems*, 8(2):171–216, Spring 1995.

[7] F. Douglis and J. Ousterhout. Transparent Process Miration: Design Alternatives and the Sprite Implementation. *Software – Practice and Experience*, 21(8):757–785, August 1991.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM - Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[9] G. Gerrity, A. Goscinski, J. Indulska, W. Toomey, and W. Zhu. Rhodos- A Testbed for Studying Design Issues in Distributed Operating Systems. In *Toward Network Globalization (SICON 91): 2nd International Conference on Networks*, pages 268–274, September 1991.

[10] M. Harchol-Balter and A. Downey. Exloiting Process Lifetime Distributions for Dynamic load Balancing. In *Proc. ACM SIGMETRICS*, June 1996.

[11] R. Kolstad, T. Sanders, J. Polk, and M. Karles. *BSDI Internet Server (BSD/OS 2.1) Release Notes*. Berkeley Software Design, Inc., Colorado Springs, CO, January 1996.