Memory Ushering in a Scalable Computing Cluster

Amnon Barak and Avner Braverman
Institute of Computer Science
The Hebrew University
Jerusalem 91904, Israel
E-mail: {amnon,braver}@cs.huji.ac.il

Abstract

Scalable Computing Clusters (SCC) are becoming an alternative to mainfraims and MPP's for the execution of high performance, demanding applications in multi-user, time-sharing environments. In order to better utilize the multiple resources of such systems, it is necessary to develop means for cluster wide resource allocation and sharing, that will make an SCC easy to program and use. This paper presents the details of a memory ushering algorithm among the nodes of an SCC. This algorithm allows a node which has exhausted its main memory to use available memory in other nodes. The paper first presents results of simulations of several algorithms for process placement to nodes. It then describes the memory ushering algorithm of the MOSIX multicomputer operating system for an SCC and its performance.

1 Introduction

Collections of computers, ranging from Scalable Computing Clusters (SCC) of (homogeneous) servers to Networks of (heterogeneous) Workstations (NOW), are becoming popular platforms for executing demanding applications, high performance and interactive computing [1]. Unlike MPP's, which allow a single user per partition, SCC's and NOW's are geared for multi-user, time-sharing environments. In order to make SCC 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. These can improve the overall performance by responding at run-time to uneven use of the resources, in order to better match the resource requirements and availability. This is necessary for performance scalability, for efficient utilization of the resources and to support multiple users in a time-sharing parallel execution environment.

An adaptive resource allocation mechanism that can respond to cluster-wide resource availability in a homogeneous configuration is preemptive process migration. This mechanism could be supervised by various algorithms, e.g., load or memory sharing, to move a process from one node to another, in order to take advantage of available resources. In spite of the advantages of preemptive process migration, few operating systems have implemented it for dynamic resource allocation. Instead, most existing systems and parallel programming environments provide only

Proc. IEEE Third Int. Conf. on Algorithms and Architecture for Parallel Processing, Melbourne, Dec. 1997.
Copyright 1997 IEEE.

explicit, user-controlled remote allocations, e.g., PVM [2] and MPI [3]. Other systems provide automated remote executions, but perform process migration at the explicit request of the user [4].

Load-balancing algorithms for SCC's were extensively researched in recent years and many schemes were simulated [5, 6]. In spite of the diversity of suggested methods, only few systems implement load-balancing [7, 8] or load-sharing [4] by preemptive process migration, and even fewer systems address other resource sharing in a NOW [1]. The availability of a preemptive process migration mechanism for dynamic work distribution creates an opportunity to manage other resources in an SCC.

This paper presents a memory ushering algorithm among the nodes of an SCC. This 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 disks. In order to avoid confusion with "shared-memory", we use the term "memory ushering" to describe this algorithm. 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 the selected nodes.

The paper begins with simulations of several algorithms for process placements in an SCC, including Round-robin, Best-fit and several adaptive schemes. We show that a Round-robin combined with process migration yields the best results among the simulated algorithms, thus confirming the results reported by Harchol-Balter and Downey [9]. We also show that this combined scheme scales up well with the size of the cluster. Based on these results we implemented a memory ushering algorithm for MOSIX, a UNIX based multicomputer operating system that already supports preemptive process migration. This algorithm is part of an overall resource sharing policy, which also includes an algorithm for load-balancing. We present the memory ushering algorithm and explain how it relates to the load-balancing algorithm. Performance measurements of the algorithm show a decrease by as much as 175% in the execution times for the tested benchmarks.

The organization of the paper is as follows: the next section gives a brief overview of the process placement problem and its complexity. Section 3 presents simulations results of of several algorithms for static, dynamic and adaptive process placements. Section 4 presents the details of the memory ushering algorithm of MOSIX, and Section 5 presents its performance. Our conclusions and future work are presented in Section 6.

2 What is Memory Ushering and Why it is Useful

There are several algorithms for assigning processes to the nodes of an SCC. Static assignment uses a predefined scheme to assign newly arrived processes, without using any run time information. Dynamic process placement, on the other hand, uses system-state information at the time of the assignment, to place new processes to nodes. In both methods, once a process is assigned, it will remain in the designated node until its completion. In contrast to the above, adaptive process placement algorithms reassign processes to nodes in response to changes in the system-state. This section presents the process placement problem and describes several scenarios where a memory ushering algorithm would be useful.

Formally, the problem of process placement to the nodes of an SCC can be defined as follows: given a set of processes by their (memory) sizes, arrival and execution times, place the maximal number of processes in the main memory of all the nodes. The rational is to avoid as much as possible disk paging, because access to a page on disk vs. main memory could be as much as four orders of magnitude slower. As stated, this problem has many variations, depending on the order of arrival and placement of the processes, completion time requirements, etc. Unfortunately, in almost all the cases the problem is NP-hard [10].

In this paper we develop a heuristic memory ushering algorithm for the approximation of the process placement problem in an SCC. Our algorithm is suitable for SCC environments that support preemptive process migration for load-balancing. This algorithm is responsible to place the maximal number of active (ready to run) processes in the "cluster RAM", across all the nodes. It attempts to avoid processes thrashing or the swapping out of processes, even if this causes the loads in the nodes to become imbalanced. The algorithm is activated only when a node's free memory falls below a certain threshold value. In this case the algorithm overrides the load-balancing algorithm, and it attempts to migrate processes to other nodes which have sufficient free memory. The algorithm is executed independently by each node, in a decentralized and non-synchronized manner.

The memory ushering algorithm is most useful in cases where many nodes are evenly loaded, but memory is not being used uniformly, and in cases where nodes have different amounts of memory. This is often the case when running parallel applications, e.g., as in PVM, with its static Round-robin process assignment. If, in this case there is a node with insufficient memory, the load-balancing algorithm will not be initiated, even though there may be spare memory available elsewhere. Therefore, migration must be initiated by other criteria.

A second 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 the selected nodes. We note that the memory ushering algorithm is not intended to deal with processes that are larger than the maximal installed memory in any single node.

3 Simulation of Process Placement Algorithms

This section presents the results of simulations of several algorithms for process placement to nodes of an SCC. In each case, we measure the maximal memory fill when the algorithm fail, i.e., the percentage of memory usage at the time when the algorithm could not place the next process in the sequence to the main memory of any of the designated nodes.

We assume that the size of each process does not change during its execution. This is justified since observations show that most of the processes reach a relatively fixed size after a short period of loading and initiation. Any major change in the process's size could be considered as a termination of one process and the initiation of another. Therefore, the algorithm is executed only when a new process arrives. The placement decision does not depend on the (unknown) remaining execution times of the process. Therefore, we assume that the execution times of the processes are infinite. To further simplify the simulations, we also assume that: (a) the size of each process is known at the time of its arrival; (b) the processes are assigned in order of their arrival; (c) a process can not be split among several nodes; and (d) once a process is placed in a node, then it could be migrated to another node, but would not be swapped out. The simulated environment included a number of nodes, all with 128MB of memory. New processes arrived using a random predefined sequence. The sizes of the process were chosen with a distribution e^{-x}, with an average process sizes ranging from 1MB to 16MB.

3.1 Static and Dynamic Process Placement

We simulated three process placement algorithms. The motivation is to expose the behavior of each algorithm, when the cluster-wide memory is nearly used up and new processes are created. The first algorithm uses a Round Robin (RR) placement, e.g., as in PVM. The second uses a Best-fit (BF) scheme, in which a new process is assigned to the node with the largest free memory. The third algorithm uses a Round-robin Next-fit (NF) policy, in which the nodes are scanned in a Round-robin fashion and a new process is assigned to the first node which has sufficient free memory. We note that we also simulated the First-fit policy. However, since this policy tends to distribute the load unevenly among the nodes, we do not consider it as a realistic alternative.

Figure 1 presents the results of the simulation of these three placement algorithms, for a set of processes with an average size of 1MB. We note that this set represent the distribution of processes in our departmental servers over a period of a month. The results are given for an increasing number of nodes, in a logarithmic scale. Each measurement represent the average of 500 different executions.

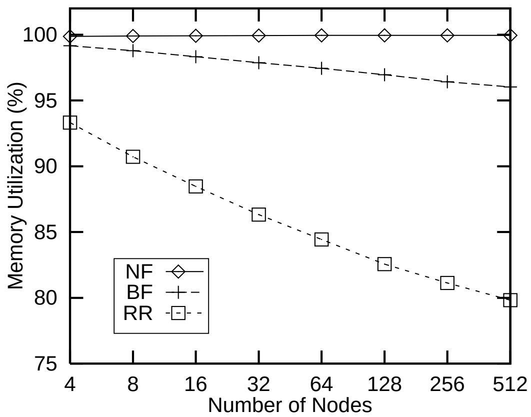


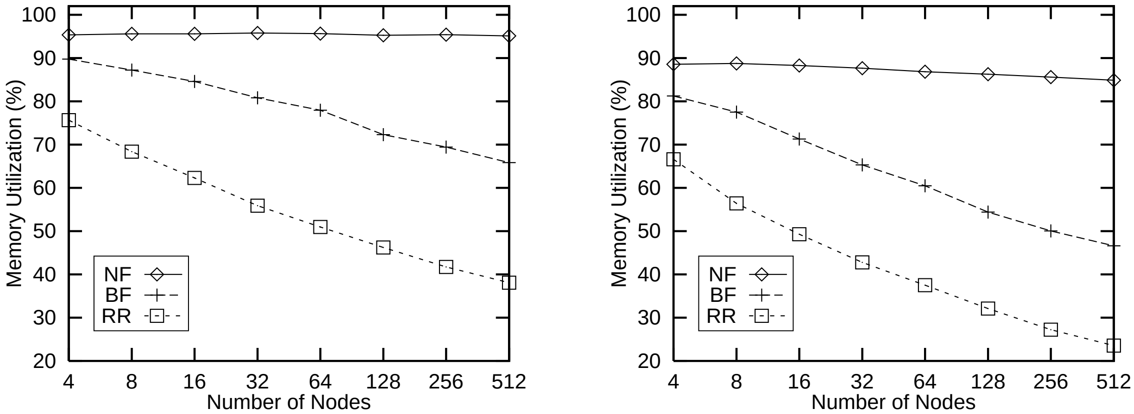
Figure 1: *Performance of process placement algorithms, 1MB average process size.*

From the figure it can be seen that the NF algorithm has a near maximal memory fill, for any number of nodes. This can be explained by its flexibility to skip nodes which has exhausted all their free memory. The corresponding results for the RR algorithm show a decrease in the maximal memory fill, from about 93% for four nodes to about 80% for 512 nodes. The reason for these lower values is that the RR algorithm fails as soon as it receives a process that does not fit into the memory of the next node. Observe that the performance of the RR algorithm drops linearly on a logarithmic scale, indicating a logarithmic decrease.

The performance of the BF algorithm is somewhat lower than the NF algorithm, with a logarithmic decrease of up to 4% below NF. The behavior of BF can be explained by the fact that this algorithm tends to distribute the memory allocations evenly among the all the nodes. Therefore, it is more likely to fail than NF, when a large process arrives, while still benefiting from more freedom than RR.

In order to explain the decrease in the performance of the RR and BF algorithms, recall that the processes are randomly chosen from a fixed distribution, with a given average size. Therefore, as the number of nodes is increased, the size of the cluster RAM is proportionally increased, thus requiring more processes to be created, in order to fill this memory. However, the creation of more processes implies that the number of large processes also increases. These large processes are more difficult to place, and thus are the main reason for the failure of these two algorithms.

In order to highlight this last observation we simulated the same three placement algorithms, using average process sizes of 8 and 16MB. Note that such process sizes may reflect scientific computation environments. The results of these simulations are shown in Figure 2. The figure shows that the performance of the RR algorithm is decreased from 75% to 40% for an average process size of 8MB, Figure 2(a), and up to 20% for an average process size of 16MB, Figure 2(b). The corresponding performance of the NF algorithm remains around 95% for 8MB, and gradually decreases from 90% to 85% for 16MB.



(a) Average process size of 8MB

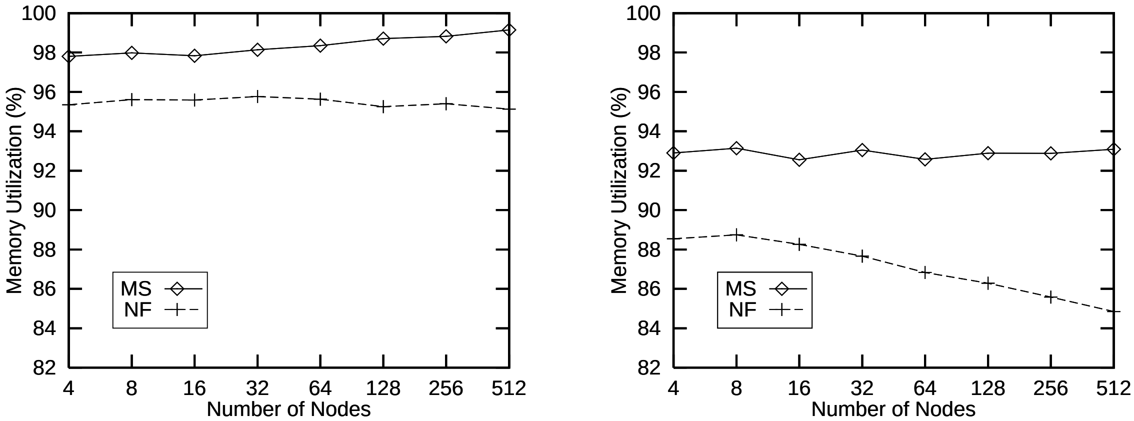
(b) Average process size of 16MB

Figure 2: *Performance of process placement algorithms with larger average process sizes.*

Comparison of the results of Figure 1 and Figure 2 further supports our claim that as more large processes are created, the decrease in the performance of the RR and BS algorithms, as the number of nodes is increased, is increased.

3.2 Adaptive Process Placement

Next, we simulated a variation of the RR algorithm using adaptive process placement, called MS (Migrate the Smallest process). The MS algorithm scans the nodes in a round robin fashion, and places a new process in the next node. Unlike RR, if that node runs out of memory, then the MS algorithm migrates a process away from that node. The chosen process is the smallest among all the processes which are large enough to undo the memory overflow caused by the arriving process, and could fit into any other node. This way, the migration incurs the lowest network overhead. If no such process exists (either because all the processes are too large or the other nodes are too full) the algorithm fails.



(a) Average process size of 8MB (b) Average process size of 16MB

Figure 3: *Performance of the MS and NF algorithms.*

Figure 3 shows the results of the simulation of the NF and MS algorithms with an average process size of 8MB and 16MB. We note that for an average process size of 1MB the performance of both algorithms was close to 100% memory fill. From Figure 3(a) it can be seen that the performance of the MS algorithm is between 3% and 4% above that of the NF algorithm. Observe that the performance of the MS algorithm increases with the number of nodes while that of the NF algorithm does not change. This can be explained by the greater choice of processes to be migrated by the MS algorithm with the increased number of nodes. The improved scalability of the MS algorithm is further highlighted by the simulations with an average process size of 16MB, shown in Figure 3(b).

We note that we also simulated two variants of the NF and MS algorithms, where the chosen process was placed in the node with the largest free memory. However, both these variants showed much poorer performance and a more steep decrease in performance as the number of nodes grew larger.

3.3 A Scalable Placement Algorithm with Decentralized Control

All the algorithms described so far use centralized schemes which rely on global information for making the placement decisions. These centralized mechanisms are not scalable and can not be efficiently implemented on clusters with a large number of nodes.

A simple decentralized scheme for process placement could be based on the MS algorithm, in which the migrated process is placed in a node chosen from an *information window*. This is a cache that holds information on the free memory of other nodes. The window could be updated at regular intervals, using a suitable information dissemination algorithm [7]. The relevant parameters of this algorithm are the size of the window and the rate of the information dissemination. Simulations of the above scheme, called "Windowed MS" are shown in Figure 4.

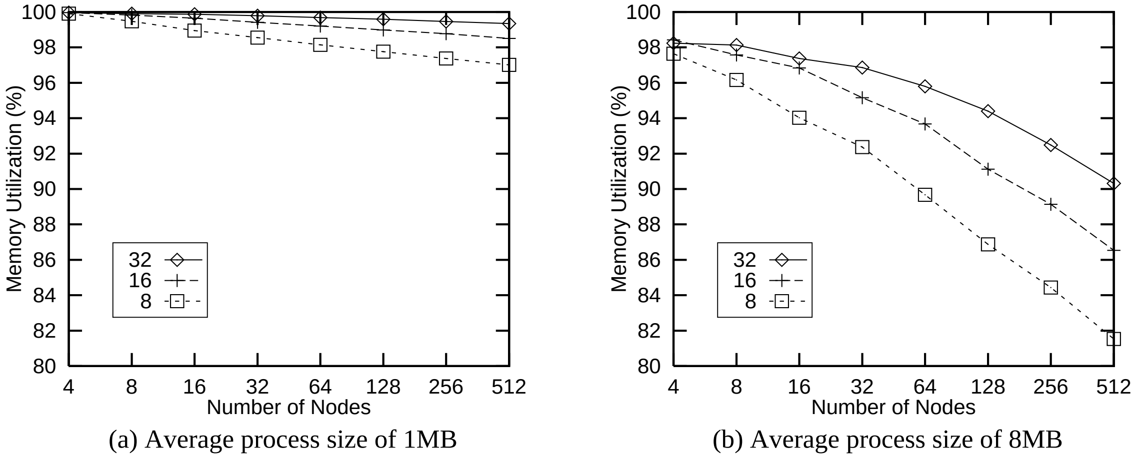


Figure 4: *Performance of the Windowed MS algorithm for different window sizes.*

Figure 4(a) depicts the simulation results with an average process size of 1MB. The results are shown for window sizes of 8, 16 and 32 entries. Figure 4(b) shows the corresponding results for simulations with an average process size of 8MB. As expected, the performance drops as the number of nodes increases or when the window size is decreased. This is due to the limited information available to the algorithm, which prevents global optimization. In spite of the decreased performance, this algorithm performs well, i.e., over 96% memory fill for an average process size of 1MB, and over 90% for 8MB processes with up to 128 nodes. The algorithm is simple to implement and it does not require any global knowledge.

4 Implementation of a Memory Ushering Algorithm

This section presents the details of an implementation of the memory ushering algorithm in the MOSIX multicomputer operating system. MOSIX [7] is an enhancement of the UNIX kernel with algorithms for resource sharing using a preemptive process migration mechanism. This mechanism can migrate any user process, transparently, any time, to any node [11]. The resource sharing algorithms respond to variations in the utilization of some resources by migrating processes from one node to another, to improve the overall performance. The most dominant resource sharing algorithm is load-balancing. The goal of the algorithm is to balance the loads of all the nodes. This algorithm prevails as long as there is no extreme shortage of resources such as free memory. The algorithm is decentralized in the sense that all of the nodes execute the same algorithm.

The memory ushering algorithm is responsible for making the decisions regarding process migrations from a node that has exhausted its free memory, as well as the prevention of migrations into such a node. The algorithm needs to decide *when* to migrate a process, *where* to place it and *which* process to migrate. In addition to these, the algorithm needs to decide whether to allow remote processes to move in due to memory depletion in other nodes.

In order to reach a decision, the algorithm uses the free memory index supplied by the information dissemination algorithm. The free memory index is a measure of the amount of the node's free memory that may be allocated to remote processes. This is the sum of the actual free memory less the free memory threshold value, i.e., memory reserved for local processes, and the amount of memory already committed to arriving processes. This value is measured several times a second so as to dampen transient fluctuations. The free memory index is sent at regular intervals to other nodes using the probabilistic information dissemination algorithm. This last algorithm is geared to provide each node with sufficient knowledge about available resources in other nodes, without polling or further reliance on remote information [7].

Below we describe the main parts of the memory ushering algorithm, its activation, the selection of the target node and the process for migration. These two last decisions are closely related and the corresponding parts of the algorithm inter-mesh. If the algorithm is successful, then at least one processes is migrated, otherwise the node starts paging.

4.1 Activating the Algorithm

The memory ushering algorithm is triggered when a node's free memory falls below a threshold value that may cause valuable pages to be paged-out. This threshold value is an area of memory reserved for the use of local processes while the decision to choose and then migrate away a process is made. Ideally, the size of this reserved memory should be a dynamic value that corresponds to the rate of the free memory allocation. This value should be increased when the page allocation rate is increased and vice versa, so as to ensure that the memory ushering algorithm chooses, and if necessary, migrates away a process before paging activities are triggered. At the same time, the size of the reserved memory should not be too large. This is to avoid early and unnecessary migrations of processes away from the node, and also the blocking of migrations into the node. In the current implementation the threshold value is set to a fix size, e.g., 1/4 MB, above the triggering level of the paging mechanism.

4.2 Choosing the Process

The choice of the process to be migrated is done in several stages, depending on the amount of the local memory shortage (overflow) and the amounts of available free memory in the other nodes. First, the total memory overflow is determined. It is defined as the difference between the current memory requirements and the available memory. The memory overflow value is monitored and updated regularly to determine whether this was a momentary event or if it is persistent.

In the first stage, the processes are sorted by the sum of their active dirty pages. The process with the lowest migration overhead is selected from the processes whose in-core dirty pages exceed the memory overflow value. If this stage fails, either because there is no node with sufficient free memory to accommodate the chosen process, or because all the processes of the sending node are smaller than the memory overflow value, then the algorithm proceeds to the second stage.

In the second stage, the algorithm attempts to migrate the largest possible process that fits into a remote node without causing that (target) node to exceed its free memory threshold value. First, the algorithm chooses the node with the largest known free memory index. Then it finds the largest process that fits into this node. The algorithm then checks if the chosen process would fit into any other nodes. From these, it chooses the one with the lowest load and then migrates the chosen process to this node. Afterwards, if the available free memory is still under the threshold value, then the algorithm waits one unit of time for the arrival of new free memory indices, and then returns to the first stage. Otherwise it terminates.

4.3 The Target Node

The target node is chosen from the subset of nodes whose free memory indices are available in the information window. The chosen node must have a free memory index larger or equal to the size of the process designated for migration. This is to prevent the receiving node from paging and also to prevent a situation where a process is repeatedly migrated from one node to another. From these, the node with the lowest load is normally chosen. If no such node is found then the algorithm is executed again after a short delay, e.g., 1 second, during which it is expected (with high probability) that new information about other nodes' free memory indices will be available.

In the above scheme, the node with the lowest load is chosen so as to avoid, as much as possible, further process migrations due to load imbalances after the migration. The general idea is to restrict the execution of the memory ushering algorithm to periods when free memory in the node becomes scarce. Otherwise, only the load-balancing algorithm is used.

We note that before any process migration starts, approval is required from the destination site. This is to avoid several simultaneous migrations from different nodes.

5 Performance Measurements

This section presents a benchmark that shows the performance gains of the memory ushering algorithm. In the benchmark, we compared the performance of MOSIX, with its load-balancing and memory ushering algorithms, to that of a computing cluster without these algorithms. In both cases, the initial process distribution was done by PVM's Round-robin algorithm. This benchmark represents cases where many nodes are evenly loaded, but memory is not being used uniformly, or nodes have different amounts of installed memory. In these cases there is no justification for load-balancing and another algorithm must be used. The execution configuration consisted of a cluster of 32 identical 200MHz Pentium-Pro servers running MOSIX. Each server had 128MB of main memory. The servers were connected by fast Ethernet through a hub.

The benchmark was executed to verify the results of the simulations presented in section 3. In each execution, a new set of processes was created and assigned to the nodes using PVM, such that some nodes run out of free memory. As in the simulations, the process sizes were chosen with a distribution e^{-x}, with an average process size of 8MB. During its execution, each process "touched" all its pages in a cyclic manner, thereby ensuring that its working set is equal to its size. 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. The execution time of each process was fixed to 80 Sec. and the processes were initiated 0.1 Sec. apart.

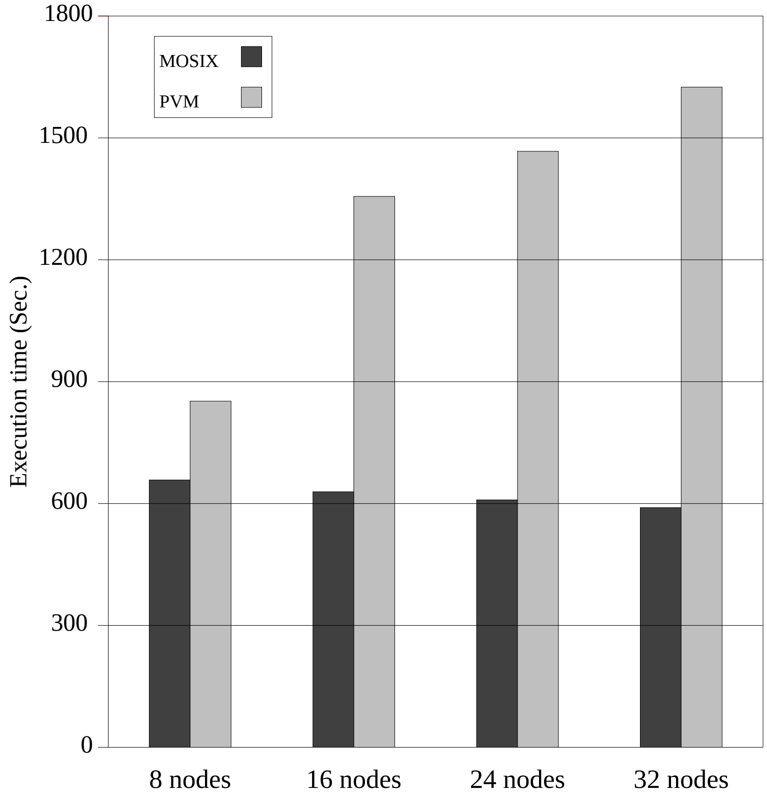


Figure 5: *Performance of PVM vs. MOSIX.*

The measured times of the executions are shown in Figure 5, where each bar represents the average measurement of five executions. From these measurements it follows that PVM without the process migration is consistently slower than PVM with the MOSIX process migration. The differences between these two execution environments range from 29% for 8 nodes, to 175% for 32 nodes. These results prove the usefulness of the memory ushering and load-balancing algorithms.

Another interesting observation from Figure 5 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 on 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.

6 Conclusions and Future Work

This paper presented the details of a memory ushering algorithm for a Scalable Computing Cluster (SCC). This algorithm allows a node which has exhausted its main memory to use available cluster memory by migrating processes to other nodes instead of paging to a local disk. The memory ushering algorithm is particularly suitable for an SCC connected by a high bandwidth communication networks, such as Myrinet [12]. This network has a transfer rate of over 8,000 (4KB) pages/Sec., an order of magnitude faster than a modern disk. This difference allows the memory ushering algorithm to improve the performance by better utilization of the cluster memory.

The work described in this paper could be extended by the development of algorithms that improve the overall utilization of cluster wide resources. Such an algorithm, which uses commodity market considerations is currently under joint development with the Metacomputing group at Johns Hopkins University. Another project is *network RAM* that can migrate portions (pages) of a large program to idle memory in remote nodes [1].

The memory ushering algorithm described in this paper could be implemented in any system that supports preemptive process migration. In MOSIX this algorithm is part of the comprehensive dynamic resource management policy, which includes algorithms for load-balancing and optimization of the inter-process communication. Together, these algorithms are geared to create an efficient execution environment, with improved performance and a high degree of overall resource utilization and sharing.

The SCC MOSIX described in this paper is compatible with BSD/OS [13]. This system has been operational for over 4 years using a cluster of 60 Pentium and Pentium-Pro servers. It is used for student course work, research and development of operating systems for scalable multicomputers, and the development of parallel applications. The unique mechanisms of MOSIX provide a convenient environment for writing and executing parallel programs with minimal burden to the application programmers. A limited (up to 6 processors) version of MOSIX, called MO6, is available on the Internet at `http://www.cs.huji.ac.il/mosix`.

Acknowledgments

We are indebted to Amnon Shiloh for his contributions and to Danny Braniss, Ilia Gilderman and Oren La'adan for their help. We are also grateful to Ariel Burton from Imperial College, London for his valuable comments.

This project was supported in part by grants from the Ministry of Defense and the Ministry of Science.

References

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

[2] 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.

[3] W. Gropp, E. Lust, and A.Skjellum. *Using MPI.* MIT Press, Cambridge, MA, 1994.

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

[5] M. Nuttall. Survey of Systems Providing Process or Object Migration. Technical Report DoC 94/10, Imperial College, London, May 1994.

[6] B.A. Shirazi, A.R. Hurson, and K.M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE Computer Society Press, Los Alamitos, CA, 1995.

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

[8] D. De Paoli, A. Goscinski, M. Hobbs, and P. Joyce. Performance Comparison of Process Migration with a Remote Process Creation and Execution in RHODOS. In *Proc. 16-th International Conference on Distributed Computing Systems*, May 1996.

[9] M. Harchol-Balter and A.B. Downey. Exloiting Process Lifetime Distributions for Dynamic Load Balancing. In *Proc. ACM SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems*, pages 13–24, May 1996.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completness.* W.H. Freeman, San Francisco, CA, 1979.

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

[12] 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):29–36, February 1995.

[13] *BSDI Internet Server (BSD/OS 3.0) Release Notes.* Berkeley Software Design, Inc., Colorado Springs, CO, February 1997.