Performance of a Parallel Application on a Network of Workstations

Anatol Piotrowski and Sivarama P. Dandamudi
Centre for Parallel and Distributed Computing
School of Computer Science
Carleton University
Ottawa, Ontario K1S 5B6, Canada

TECHNICAL REPORT TR-97-10[1]

Abstract — Performance of parallel applications on networks of workstations can be different from that on a parallel machine. Therefore, it is important to understand the factors that influence the performance of parallel applications on this type of systems. To gain this understanding, we study the performance of a matrix multiplication application on a NOW-based system that uses the PVM. We have used a network of heterogeneous workstations, which are connected by two networks—the standard Ethernet (10 Mbps) and the Fast Ethernet (100 Mbps). These two networks allow us to study the effect of network speed on the performance. We report the impact of various system and workload parameters, mapping, and load sharing algorithms on the performance of the matrix multiplication application.

Key words: Networks of workstations, Load sharing, Matrix multiplication, Parallel virtual machines, Performance evaluation.

1 Introduction

Exploitation of networked workstations for parallel processing has received substantial interest with the availability of Parallel Virtual Machine (PVM) [3, 10] and Message Passing Interface (MPI) [4] in the public domain. The chief advantage of such systems is the cost effectiveness.

Parallel computers are very expensive and their availability is not as ubiquitous as networks of workstations. It has been observed that, in practice, up to 80% of workstations are idle depending on the time of day [6]. Thus, network of workstations (NOW) provide an opportunity to emulate parallel machines with large aggregate processing power and memory. An excellent justification for NOW is given by Anderson et al. [1]. Several parallel machines are, in fact, based on workstation chips. Some examples of such systems are the Cray T3D (based on DEC Alpha) and the recent Intel system (based on Pentiums). Thus, from the hardware point of view, the main difference between a real parallel system and the one emulated on a NOW is the communication network. Parallel systems use better interconnection networks, better switching techniques (e.g., wormhole routing) whereas NOW-based systems use high-overhead, low-bandwidth LANs. The performance of a NOW-based system improves with the use of high-speed LANs (e.g., ATMs). There are, however, some other significant differences between the two types of systems. Some of these are:

1. *Node heterogeneity:* NOW-based systems typically consist of heterogeneous nodes whereas real parallel systems typically use homogeneous nodes.

2. *Load on nodes:* Since local injection of load is possible by the owner of a workstation, there is a high variability of load on nodes over time in NOW-based systems. In addition, load imbalances exist among the nodes. These problems are not as serious in parallel systems as the system is under the control of a scheduler.

[1]This paper appears in *Int. Symp. High Performance Computing Systems*, Winnipeg, July 1997. A copy of this paper can be obtained from `http://www.scs.carleton.ca`.

3. *Node failures:* NOW-based systems suffer from node failures as owners have control of their workstations. For example, a owner might reset his/her workstation. Thus, fault-tolerance and fault-recovery are important for these systems.

Objectives

Our objective in conducting this study is to gain an understanding of the factors that significantly influence the performance of NOW-based systems. As enumerated, there are significant differences between a NOW-based system and a real parallel system. In particular, our goal is to provide answers to questions like: How important is the network speed? Since the network is heterogeneous, what is the influence of task mapping? How beneficial is load sharing? What is the effect of frequency and amount of communication? A related question is: What is the impact of task granularity on the performance?

In order to gain this understanding, we use matrix multiplication application running on a network of Pentium and Pentium Pro nodes. The NOW-based system runs under PVM. In order to look at the intrinsic behaviour of the application, we have conducted experiments in a controlled manner with no background load. We have also done experiments with background load. In addition, we have used a generic workload in which we can vary the computation and communication characteristics of applications. These results, however, are not reported here due to space limitations (see [7] for details).

The remainder of the paper is organized as follows. Section 2 gives a brief description of the experimental environment used in conducting the experiments. Section 3 presents the algorithms we have implemented. The results are discussed in Section 4 and conclusions are given in the last section.

2 Experimental Environment

The network of nodes used in our experiments is based on inexpensive off-the-shelf components. The system runs PVM under Linux. The current configuration consists of 4 Pentium/133 nodes, 2 Pentium/166 nodes, and 3 Pentium Pro/200 nodes, each with 32MB of memory. These nine nodes are interconnected with a 10 Mbps “service” network, as well as with a 100 Mbps 100 Base TX high-speed network. The 10 Mbps network is used for booting the system, providing the basic communication and remote file service through NFS. The 100 Mbps network is connected with a switch that has a 2 Gbps backplane. Both the 10 Mbps network and 100 Mbps network can be used for high level communication under PVM. To an outside user, the system looks like a set of normal workstations.

3 Algorithms

We have implemented several algorithms to investigate performance sensitivity of the matrix multiplication application to various system and algorithm related issues. All algorithms are based on the master-slave strategy and follow the fork-and-join structure shown in Figure 1a. In this strategy, a master program acts as the coordinator and spawns as many slave programs as required (typically equal to the number of nodes in the PVM). The slave programs actually perform (part of) the required matrix multiplication. The master is responsible for distributing the work to the slaves and also for collecting the partial results returned by the slaves and processing them to get the final result. There is no communication among the slaves. In this model, all communication is between the master and a slave as shown in Figure 1b.

We divide the algorithms into two broad classes: algorithms that send one of the two matrices to all slave programs and those that send only a submatrix. We refer to the first class of algorithms as the *matrix-based algorithms* and the second class as the *submatrix-based algorithms*. The associated tradeoffs between the two classes are discussed in Section 3.2.

3.1 Matrix-Based Algorithms

These algorithms send one of the two matrices to all slaves. In the following discussion (and in our implementation), we assume that the first matrix is sent to all slaves. The effect of sending the second matrix

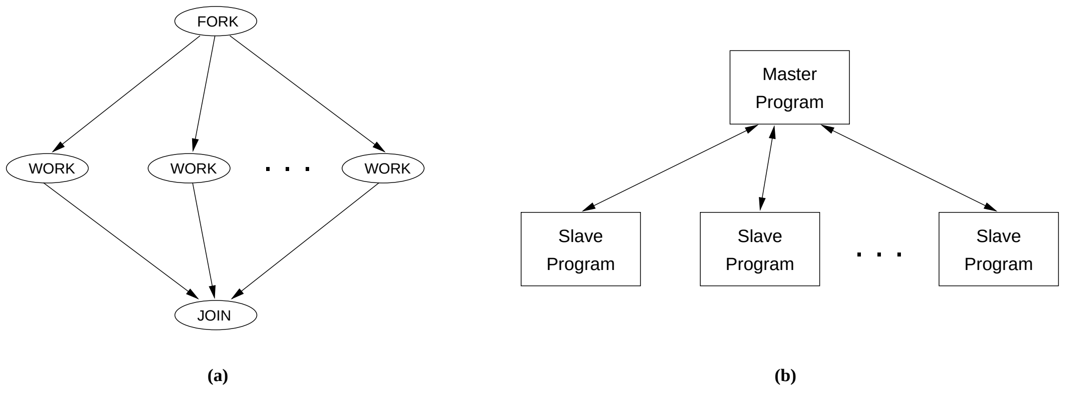


Figure 1: General structure of the algorithms (a) Fork-and-join structure; (b) Structure of the implemented algorithms

instead of the first is discussed in [7].

3.1.1 No Load Sharing (NLS) Algorithm

In this algorithm, the master program sends the first matrix to all slaves. The second matrix is distributed equally to all slaves i.e., the number of columns of the second matrix sent to each slave is C/S, where C is the number of columns in the second matrix and S is the number of slaves. If there is also a slave program running on the master node, S in the formula should be replaced by $S + 1$. The algorithm is based on the 'send' algorithm described in Section 3.1.2. That is, the master sends individual packets of data consisting of the whole second matrix plus its share of the first matrix to each slave sequentially.

3.1.2 Load Sharing Algorithms

All the remaining algorithms allow load sharing and are based on the "pool of tasks" paradigm. Each slave receives a task to work on and when the task is completed it sends the results back to the master. Master receives the results and then sends the slave another task. There is no communication between the slave programs. The same code is used to multiply matrices or submatrices in every algorithm. Performance of the algorithm is a function of the task size t (i.e., granularity of the task). For example, if task size is fixed at three columns, each time a slave requests work, the next three columns of the second matrix are sent. There is a tradeoff between task granularity and communication overhead. Finer granularity (e.g., one column) introduces more frequent communication but improves load sharing. Coarser granularity decreases the frequency of communication but also decreases the scope of load sharing. In the extreme, we can set $t = C/S$ as in the NLS algorithm with no load sharing. We will discuss the effect task granularity in Section 4.5. We can divide load sharing algorithms into two groups depending on whether the algorithm uses fixed or variable granularity. We have implemented three variable granularity algorithms: guided self-scheduling [9], factoring [5], and trapezoidal self-scheduling [11]. The focus of this paper is on fixed granularity algorithms. Performance of variable granularity algorithms is reported in [8]. We now describe three fixed granularity algorithms that are based on the pool of tasks paradigm.

Broadcast

In this algorithm, the first matrix is broadcast to all slaves. Then the first t columns of the second matrix are sent to the first slave, second t columns to the second slave, and so on. Each slave multiplies the columns that it receives by the first matrix and sends the results back to the master. Master collects the result, writes it into appropriate place of the result matrix and then sends another t columns of the second matrix to the slave. This continues until all rows of the second matrix have been sent.

Send
A potential problem with the Broadcast algorithm is that it takes some time for the first matrix to be received by all slaves. This is particularly so if broadcast is implemented as a series of point-to-point communications. As the master program is broadcasting the first matrix to all slaves, some slaves may receive it earlier than the others. Once a slave has received the first matrix, it needs to get t columns of the second matrix to start the multiplication. If the master is still busy broadcasting the first matrix to the other slaves, the slave will be idle until the master finishes the broadcast and sends it t columns of the second matrix.

To avoid this problem, this algorithm sends the first matrix and t columns of the second matrix together to the first slave so it can start the multiplication, then the first matrix and the next t columns to the second slave, and so on. We will compare the performance of these algorithms in Section 4.6.1.

Replicate
A problem with the previous algorithms is that if a task has been sent to a slave that is particularly slow or heavily loaded, the whole computation suffers. This can happen in a heterogeneous and/or dynamic environment where load variations are frequent. This problem causes more serious performance problems if the task granularity is coarse. Furthermore, the other algorithms (discussed so far) are not fault-tolerant. If a node fails before returning the results, the whole application suffers. Replicate algorithm addresses these problems by sending a task to more than one slave.

Our implementation of this algorithm is based on the Send algorithm[2]. Once all available tasks are distributed to the slaves as in Send, master must wait for all the results to be send back from slaves. Master keeps track of all tasks that are currently being executed by the slaves. When there are no more tasks to distribute, master queues all tasks that are being executed on the slaves. It also maintains an idle slave node queue. As long as there are still some tasks whose results are not yet received by the master, it performs the following in a loop:

- Priority is given to receiving the results. Therefore, as long as there are some messages carrying partial results from slaves, these results will be received and processed by the master. If a task result is no longer needed because it has already been received from another slave, it is discarded. If the result is needed, the task whose result was received is removed from the task queue. The slave from which the results were received is now idle and added to the idle slave queue.

- If the idle slave queue and the task queue are not empty, a task is dequeued and sent to the first idle slave. The salve is dequeued from the idle slave queue. The dequeued task is added to the end of the task queue. Algorithm tries to minimize the number of times a task is replicated but if there is a lot of idle slaves and relatively few tasks left, some tasks may be replicated several times on different slaves. When all task results were received by the master, it kills all the slaves.

3.2 SubMatrix-Based Algorithms

Algorithms SubMatrix and CachedSubMatrix are based on a different principle than the previous ones. These algorithms are based on the algorithms that are most often used for matrix multiplication in multiprocessors. We show the principle of these algorithms by means of an example. Assume that A and B are both $n \times n$ matrices where $n = 2k$. Then A and B can be thought of as conglomerates of four smaller matrices, each of size $k \times k$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

[2]Note that replication can also be incorporated in other algorithms.

Given this partitioning of A and B into blocks, result of matrix multiplication C = A×B is defined as follows:

$$C = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

A and B matrices can be divided into more than just four pieces. There are tradeoffs between matrix-based and this class of algorithms. Matrix-based algorithms tend to reduce the amount of communication but require more memory as one complete matrix has to be stored at each slave. On the other hand, submatrix-based algorithms tend to increase the amount of communication but the memory demand is less as only submatrices are stored at each slave. In addition, in submatrix-based algorithms, the master has to do more work as it has to perform additions on partial results returned by slaves.

We have implemented two algorithms based on the basic send policy. These algorithms, however, can also be used to augment other algorithms (e.g., the replicate algorithm).

3.2.1 SubMatrix Algorithm

In this algorithm, every submatrix multiplication is sent to a slave. The master just adds up the results. In the above example, A_{11} and B_{11} would be sent to the first slave, A_{12} and B_{21} to the second, and so on. Master collects the results, adds them to the results already received and then sends another set of submatrices of A and B to the slave for multiplication. We have conducted experiments to study how varying the number of submatrices (i.e., task granularity) affects the performance (see Section 4.5 for a discussion of this issue).

3.2.2 CachedSubMatrix Algorithm

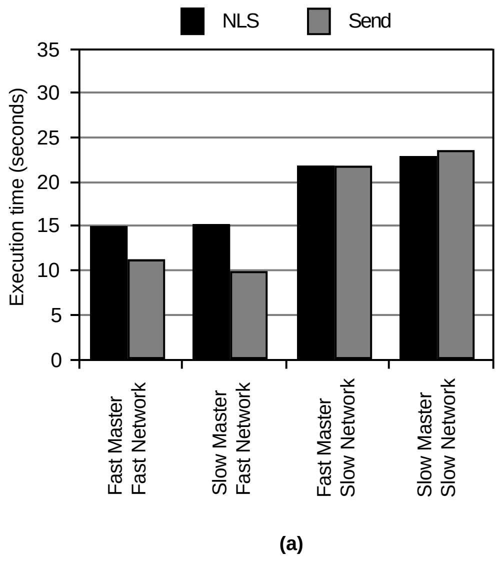
This algorithm is very similar to the one above. The major difference is that the master keeps track of the submatrices of A that have been sent to the slaves. This allows for caching submatrices of A on slaves. As far as possible master sends submatrices of B that needs to be multiplied by the submatrix of A that is cached on the slave. If no further multiplications are to be done with the cached submatrix of A at a slave, another submatrix of A is sent to the slave for caching. Care is taken to cache different submatrices of A on every slave, but once all of them are used it is possible that multiple slaves will be caching the same submatrices. The effectiveness of caching is discussed in Section 4.6.3.

4 Results and Discussion

This section presents the results of the experiments conducted. Throughout these experiments we have used a 720×720 matrix. The size of the matrix is determined by the amount of memory available at a node as the matrix-based algorithms require the storage of a complete matrix at each node. Since our objective is to study the performance sensitivity to various parameters and algorithms, we have conducted these experiments in a controlled environment that carefully introduces background load. However, the results reported here are obtained with no background load. We use the execution time as the performance metric. The execution time values reported in this section are the averages of 30 runs (with a standard deviation of less than about 1% of the mean value). In order to make fair comparisons among the algorithms, we have implemented matrix multiplication in exactly the same way in all the algorithms.

4.1 Load Sharing versus No Load Sharing

This section compares the performance with and without load sharing. Recall that the No Load Sharing (NLS) algorithm described in Section 3.1.1 is based on the Send algorithm. Figure 2 shows the relative performance of these two algorithms when the master node also works on the multiplication. Note that both algorithms send the complete first matrix to all slaves. The difference is in sending the second matrix data. Send uses “pool of tasks” whereas NLS sends all required columns in one message. The results for Send are obtained with a task granularity of 3 columns.



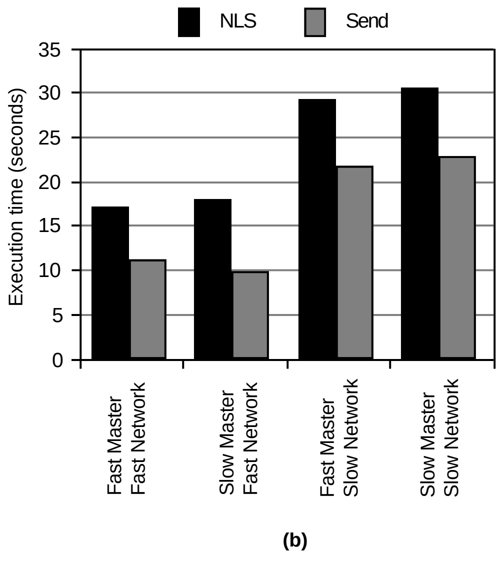


Figure 2: Performance of No Load Sharing (NLS) versus Send (a) task distribution is "slow first"; (b) task distribution is "fast first"

In the NLS algorithm, the first matrix along with 720/9 = 80 columns of the second matrix are sent to slaves serially. In a heterogeneous environment, assuming that we can rank order the nodes by their speed, task distribution can be done in one of two ways: slowest to fastest (i.e., "slow first") or fastest to slowest (i.e., "fast first"). To be consistent, we have also done the initial distribution of tasks in Send in both ways.

From the data presented in Figure 2 it can be seen that the task distribution order affects the performance of NLS much more than that of Send. Send exhibits a more robust performance to task distribution order. It is clear that Send does not depend heavily on the initial distribution as the load sharing aspect of this policy will even out the initial task distribution imbalances. The data also shows that the sensitivity of NLS to task distribution order increases when the network is slow.

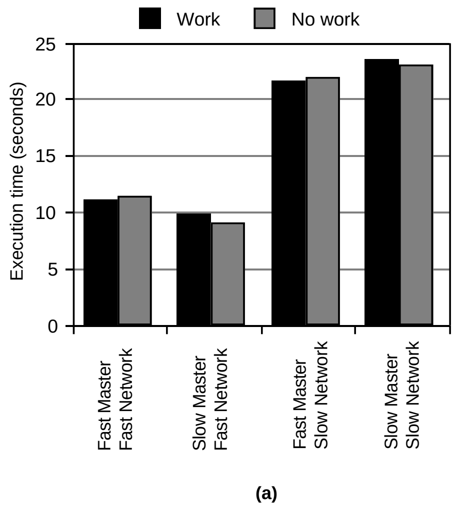
While we have done experiments in a controlled environment, in a real system, rank ordering of machines is not possible because load on machines can vary dynamically. A particularly heavily loaded fastest machine could, in fact, be the slowest from the application point of view. In this type of uncertain environment, the sensitivity of NLS to the task distribution order is a serious drawback and load sharing is preferred.

4.2 Fast Master versus Slow Master

Figure 3 shows the performance of the Send and Cached SubMatrix algorithms when the master works on the multiplication ("Work") and when not ("No work"). Note that, in our set up, the fastest machine (200 MHz Pentium Pro) is about 6 times faster in performing the matrix multiplication than the slowest machine (133 MHz Pentium). We note the following from the data presented in this figure:

- *Slow Network:* When the network is slow, it does not make a significant difference whether the master program is mapped to the fastest node or not. The difference in performance is marginal as shown in Figure 3. This observation is valid whether or not the master performs coordination only or coordination as well as computation.

- *Fast network:* When the fast network is used, the best mapping depends on several factors:



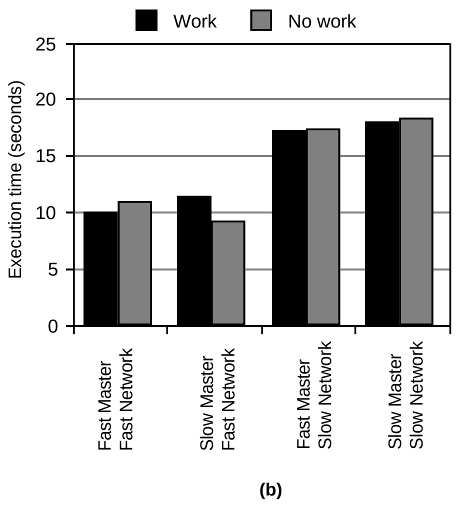


Figure 3: Performance sensitivity to system characteristics (a) Send algorithm (task granularity is 3 columns); (b) Cached SubMatrix algorithm (matrix is divided into 3 $\times$ 3 sub-matrices)

- *When the master performs coordination only:* In this case, mapping the master to the slowest node is better. This is because, if the fastest node is used for pure coordination task, it is taken out of the worker pool of nodes that actually performs the computation. Therefore, the execution time increases. This is true for both algorithms as shown in Figure 3.
- *When the master performs coordination as well as computation:* In this case, best mapping mainly depends on the amount of work master performs in processing replies from slaves. If reply processing takes negligible amount of time (as in Send algorithm — see Figure 3a), mapping the master to the slowest node is better as coordination activities do not overwhelm even the slowest node and faster nodes can be used for computation. If, on the other hand, reply processing time is not negligible (as in Cached SubMatrix algorithm — see Figure 3b), mapping the master to the fastest node is better as the faster node can respond better to requests from slaves. Mapping the master to the slowest node, for example, could affect the communication performance due to delays caused by the slow master node in processing replies. These mappings become more important with decreasing task granularity as finer task granularity increases the rate of replies from slaves.

4.3 Work versus No Work

The question that we address in this section is: whether the master node should strictly perform coordination (i.e., distribute tasks and process results from slaves) or should it also perform actual computation like a slave? From the data presented in Figure 3 we can observe that if the network is slow, it does not make a significant difference. On the other hand, with fast master, performance tends to improve, in most cases, when the master performs computation in addition to its coordination activities.

In practice, therefore, master should work on computation if it is on a fast machine[3]; otherwise, restrict itself to coordination activities. It is, however, not possible to determine if a machine is "fast" or "slow" statically due to dynamic load changes. Therefore, master should be adaptive to the perceived service that a slave is providing. One way of implementing this adaptiveness is to give a low priority to

[3]Fast from the application point of view, not the "raw" speed of the machine.

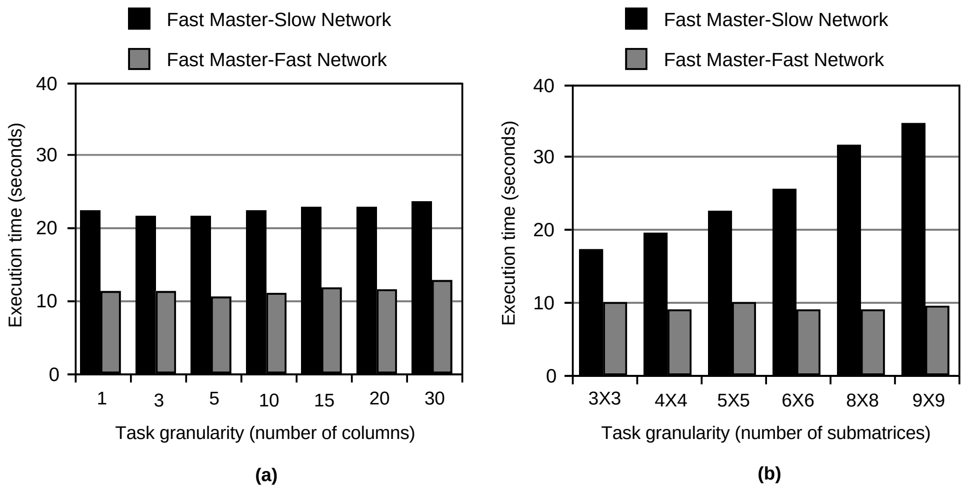


Figure 4: Performance sensitivity to task granularity (a) Send algorithm; (b) Cached SubMatrix algorithm

the computation process on the master so that the master works on the computation only when it is not busy with coordination activities.

4.4 Impact of Network

The data presented in Figures 2 and 3 indicate that the network speed has the most impact on the performance of the algorithms. The actual gain in performance depends on the type of algorithm used. This also demonstrates a major drawback of using networks of workstations (NOWs) for parallel processing. Parallel systems have a dedicated interconnection network that is specifically designed to provide good message-passing performance. This improvement in communication performance is achieved by a combination of higher bandwidth, better switching techniques (e.g., wormhole routing is used in current parallel systems from Cray, Intel and nCube). This communication bottleneck has been recognized widely as the chief drawback of NOW-based parallel computation (for example, see [1]). The emergence of high-speed, high bandwidth networks will help improve the performance of NOW-based systems.

4.5 Task Granularity

Task granularity is an important parameter of an algorithm that uses load sharing. Task granularity represents a tradeoff between frequency and amount of communication and scope of load sharing. Figure 4 shows the impact of task granularity on the performance of Send and Cached SubMatrix algorithms when the master works on the computation. The x-axis in Figure 4a gives the number of columns sent in response to a request from a slave. For the Cached SubMatrix algorithm (see Figure 4b) the x-axis gives the number of sub-matrices that the original matrix is divided into. For example, 3×3 indicates that the matrix is divided into 9 sub-matrices. Thus, in Figure 4a, we move towards finer granularity as we move to the left; in Figure 4b, moving to the right takes us to finer granularity.

We can make two observations based on the data:

- The performance of the matrix-based algorithms is fairly insensitive to task granularity. Also, these algorithms exhibit robust performance to task granularity independent of whether the network is slow or fast. This is because, while the number of messages depend on the task granularity, the amount of data sent is independent of the task granularity used. In our experiments, it is equal to 10*(720*720) data items.

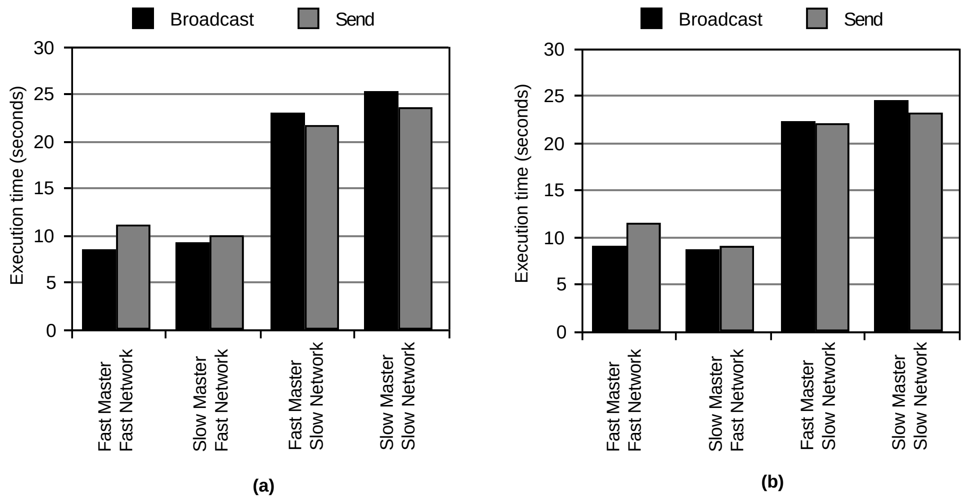


Figure 5: Performance of Broadcast and Send algorithms (a) Master works; (b) Master does not work

- SubMatrix-based algorithms are sensitive to task granularity when the network is slow. However, the use of a fast network makes this algorithm robust to task granularity. This is because, the amount of data sent by SubMatrix-based algorithms is a function of granularity. For example, the amount of data sent is 6*(720*720) data items when the task granularity is 3×3 and increases to 18*(720*720) data items for 9×9 granularity.

4.6 Relative Performance of Algorithms

4.6.1 Broadcast versus Send

This section presents the relative performance of Broadcast and Send algorithms. The main difference between the two algorithms is that Broadcast sends the first matrix to all slaves before giving columns of the second matrix. The Send algorithm, on the other hand, sends the first matrix and the t columns of the second matrix together to slave 1, first matrix and the next t columns of the second matrix to slave 2, and so on, where t is the task granularity. As a result, when there is a significant time lag in receiving the first matrix by the slaves, Send should perform better. This is the case if broadcast is not supported by the network. The relative performance of Broadcast and Send algorithms is shown in Figure 5. The task granularity used in both cases is 3 columns. It is clear from the data in this figure that Send performs better when the network is slow. On the other hand, broadcast gives better performance for the high-speed network. Data from other experiments (not shown here) indicates that if broadcast is supported by the network, Send is inferior to Broadcast even if the network is slow.

4.6.2 Replicate versus Send

The Replicate algorithm is proposed for two main purposes: (1) to perform effective load sharing towards the end of computation and (2) to improve fault-tolerance (see Section 3.1.2 for details). Figure 6 shows the relative performance of Replicate and Send algorithms when the task granularity is 3 columns and 30 columns. Note that the experimental environment used for these experiments did not involve any background load (and hence no load fluctuations). Under these conditions, Replicate algorithm shows improvements only in a scenario where the slow machines are delaying the job completion. This, clearly, depends on the task granularity. The larger the task granularity, the more probable is the scenario just described. The data presented in Figure 6 demonstrates that Replicate is not beneficial under no background load variations with small task granularity. Replicate, however, does provide performance benefit when

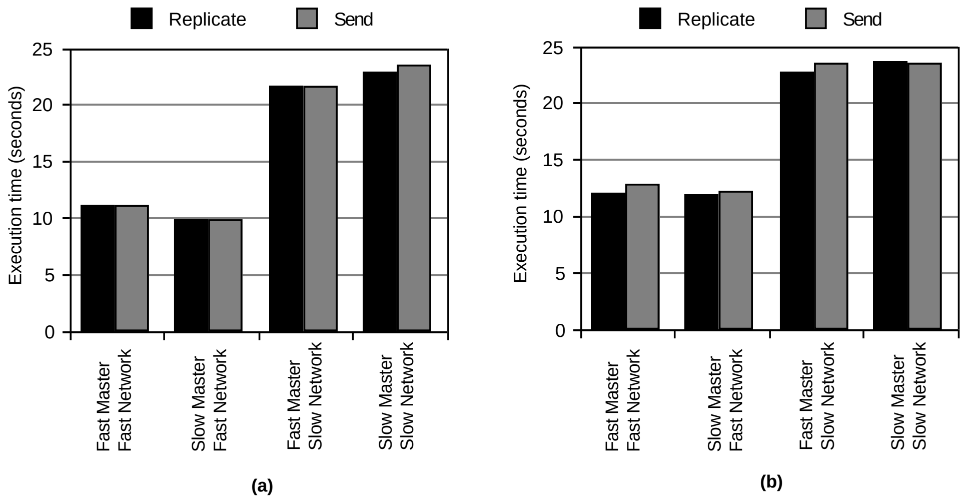


Figure 6: Performance of Replicate and Send algorithms (Master works) (a) Task granularity is 3 columns; (b) Task granularity is 30 columns

large task granularity is used. We have also performed experiments by introducing background load variations. Under such conditions, Replicate provides substantial performance improvements (these results are not presented due to lack of space). It should also be noted that the performance that we have reported is under no failure case. When there are failures, using Replicate is clearly advantageous.

4.6.3 Matrix-Based versus SubMatrix-Based Algorithms

We have identified two classes of algorithms: Matrix-based and SubMatrix-based. In Matrix-based algorithms, the first matrix is sent to all slaves and is useful only when there is enough main memory to hold it. In SubMatrix-based algorithms, sub-matrices are sent to the slaves, which reduces the memory requirement. The relative performance of these two algorithms is shown in Figure 7. The data in this figure suggests that the relative performance is dependent on the network speed. When the network is slow, Send tends to perform worse than SubMatrix. This is due to the fact that Send transmits the first matrix to slaves serially, which it may cause considerable delay in receiving the matrix by the last slave. This is particularly so if the network is slow. Since only sub-matrices are sent in SubMatrix-based algorithms, this problem is not as serious as in Send. When the network is fast enough to reduce the adverse effects of the initial transmission of the whole matrix, Send performs as well as or better than SubMatrix.

The performance of SubMatrix-based algorithms can be improved by reducing the amount of communication. In this improved algorithm, called Cached SubMatrix, the message length is reduced by half for majority of the messages as it sends only one sub-matrix as opposed to two sub-matrices. Figure 8 shows the relative performance of these two algorithms. The Cached SubMatrix algorithm shows significant improvements when the network is slow. This is a direct consequence of reduced message length in Cached SubMatrix algorithm.

5 Conclusions

We have conducted experiments to study the performance sensitivity of the matrix multiplication application to various system and workload parameters, mapping, and load sharing algorithms. We have used the standard Ethernet (10 Mbps) and Fast Ethernet (100 Mbps) to see the impact of network. Some of the conclusions derived from the results reported in this paper are:

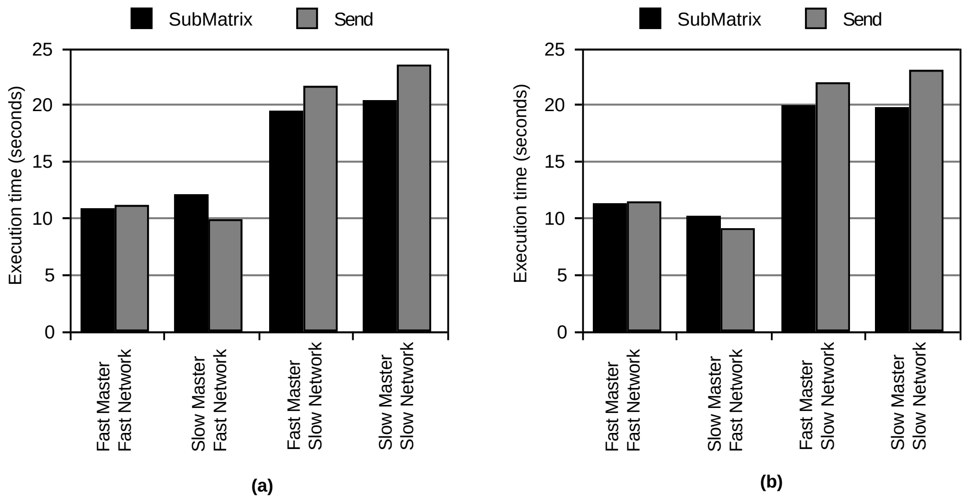


Figure 7: Performance of SubMatrix and Send algorithms (a) Master works; (b) Master does not work

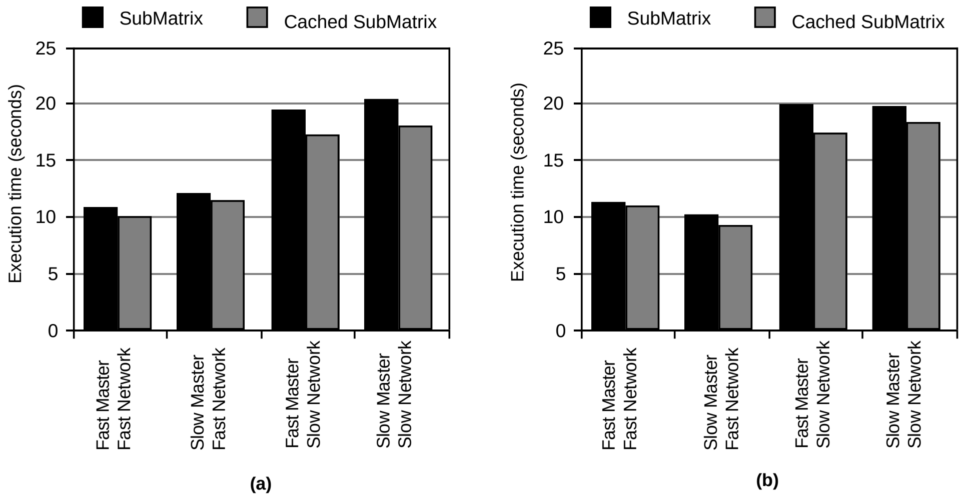


Figure 8: Performance of SubMatrix and Cached SubMatrix algorithms (a) Master works; (b) Master does not work

- The speed of the network has a substantial impact on the execution time of matrix multiplication than any other parameter we have studied.
- Performance benefits of mapping of the master program to faster or slower node depends on several factors including whether master node is strictly responsible for coordination or it performs computation in addition to coordination and speed of the network.
- Task granularity affects performance only if the network is slow and the amount of communication is a function of task granularity. In all other cases, task granularity does not significantly impact the performance.
- Load sharing algorithms provide better performance than the no-load sharing ones even when the network is slow and the communication overhead is high. However, no-load sharing performs as

well as the load sharing algorithms if the task distribution order is "proper" (i.e., from the fastest machine to the slowest one). Determination of this order, however, is not possible in practice due to dynamic load changes.

This study can be extended in several directions. First, there is a need to model synthetic applications in which computation and communication can be controlled precisely so that the impact these factors can be studied to gain an understanding of the underlying behaviour. What if the application is compute-intensive? What if it is communication-intensive? This information will be useful to programmers in designing their applications. We are currently investigating the effects of computation and communication by using a synthetic workload [7].

There is also a need to study other applications that exhibit different communication characteristics than the matrix multiplication application considered here. In addition, the beneficial effect of emerging network technologies (e.g., ATM) needs to be quantified. On such high-speed LANs, PVM communication primitives can be enhanced to improve its performance [2]. These enhancements can significantly affect application-level performance. Further study is also needed to understand the impact of I/O intensive applications.

Acknowledgement
We gratefully acknowledge the financial support provided by the National Sciences and Engineering Research Council and Carleton University.

References

[1] T. Anderson, David Culler, David Patterson, and the NOW team, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, 15(2), February 1995, pp. 54-64.

[2] S. L. Chang, D. H. C. Du, J. Hsieh, and R. P. Tsang, "Enhanced PVM Communications over a High-Speed LAN," *IEEE Parallel and Distributed Technology*, Fall 1995, pp. 20–32.

[3] G. A. Geist and V. S. Sunderam, "Network-Based Concurrent Computing on the PVM System," *Concurrency: Practice and Experience*, Vol. 4, No. 4, June 1992, pp. 293–311.

[4] W. Gropp. E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, Mass., 1994.

[5] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *Comm. ACM*, Vol. 35, No. 8, August 1992, pp. 90–101.

[6] M. Mutka and M. Livny, "The Avaialble Capacity of a Privately Owned Workstaion Environment," *Performance Evaluation*, Vol. 12, No. 4, July 1991, pp. 269–284.

[7] A. Piotrowski, *Performance of Parallel Programs on a Network of Workstations*, MCS Thesis, School of Computer Science, Carleton University, Ottawa, 1997 (can be obtained from `http://www.scs.carleton.ca`).

[8] A. Piotrowski and S. P. Dandamudi, "A Comparative Study of Load Sharing on Networks of Workstations," Techincal Report, School of Computer Science, Carleton University, Ottawa, 1997 (can be obtained from `http://www.scs.carleton.ca`).

[9] C. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers," *IEEE Trans. Computers*, Vol. 36, No. 12, December 1987, pp. 1425–1439.

[10] *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratories, 1993. (Manual and source code available via anonymous ftp from *netlib2.cs.utk.edu* in directory */pvm3*).

[11] T. H. Tzen and L. M. Ni, "Dynamic Loop Scheduling for Shared-Memory Multiprocessors," *Proc. Int. Conf. Parallel Processing*, Vol. II, 1991, pp. 247–250.