A Comparative Study of Load Sharing on Networks 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-97-14

Abstract

Networks of workstations (NOWs) can be used for parallel processing by using public domain software like PVM. There are significant differences between a real parallel system and a NOW-based system. As a result, load sharing on a NOW can be different from that on a parallel machine. Therefore, it is important to understand the factors that influence the performance of load sharing algorithms on NOWs. To gain this understanding, we study performance of a matrix multiplication application on a network of heterogeneous workstations that uses the PVM. We report performance of five load sharing algorithms that use fixed, variable, and adaptive task granularity.

1 Introduction

Exploitation of networked workstations for parallel processing has received substantial interest with the availability of Parallel Virtual Machine (PVM) [5, 11] and Message Passing Interface (MPI) [4] in the public domain. The chief advantage of such systems is their cost-effectiveness.

Parallel computers are expensive and their availability is not as ubiquitous as the network of workstations. It has been observed that, in practice, up to 80% of the workstations are idle depending on time of day [7]. Thus, networks of workstations (NOWs) provide an opportunity to emulate parallel machines with a large aggregate processing power and memory. An excellent justification for NOWs is given by Anderson et al. [2]. Several parallel machines are, in fact, based on workstation chips. Examples of such systems include the Cray T3D (based on DEC Alpha) and the Intel Teraflop system (based on Pentiums). Thus, from the hardware point of view, the main difference between a 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 such as ATMs. There are, however, some other significant differences between the two types of machines. Some of these are listed below:

1. *Node heterogeneity:* NOW-based systems typically consist of heterogeneous nodes whereas parallel systems use homogeneous nodes.
2. *Load on nodes:* Since local injection of load is possible by owners of workstations, there is a high variability of load on nodes over time in NOW-based systems. In addition, load imbalances can exist among the nodes. These problems are not as serious in parallel systems as the system is under the control of a scheduler.
3. *Load on network:* In a parallel system, the communication network is dedicated to the parallel workload. On a NOW-based system, the network supports the load generated by the parallel workload as well as that generated by local jobs submitted to the workstations.
4. *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.

Load Sharing

Load sharing plays an important role in improving performance of parallel applications in NOW-based systems. Load sharing and parallel task scheduling in distributed-memory parallel systems have received considerable attention [1, 3]. In multiprocessors, parallel loop scheduling has been studied extensively [6, 10, 12]. However, we are not aware of any studies in the context of NOW-based systems. The purpose of this study is to evaluate performance of various load balancing algorithms in a NOW-based parallel system environment. We study performance of five load sharing algorithms that use fixed, variable, and adaptive task granularity.

[*]Appears in *Proc. Int. Conf. Parallel and Distributed Computing Systems*, New Orleans, October 1997. This report can be obtained from http://www.scs.carleton.ca.

Objectives

Our goal is to evaluate performance of these five load sharing algorithms in a NOW-based system. As enumerated, there are significant differences between a NOW-based system and a real parallel system. Thus, it is interesting to see how the load sharing algorithms perform in NOW-based systems. In order to gain this understanding, we use a matrix multiplication application running on a network of Pentium and Pentium Pro nodes. The NOW-based system runs under PVM.

The remainder of this paper is organized as follows. The next section gives a brief description of the experimental environment used in conducting the experiments. Section 3 gives details on the algorithms implemented. The results of the experiments are discussed in Section 4. The paper concludes with a summary.

2 Experimental Environment

The network of nodes used in our experiments is based on inexpensive off-the-shelf components, both for processing and communication. The current configuration consists of 6 Pentium and 3 Pentium Pro nodes. All nodes have 32 MB 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 networks can be used for high-level communication under PVM. To an outside user, the system looks like a set of normal workstations.

3 Load Sharing Algorithms

We implemented five load sharing algorithms to investigate performance sensitivity of the matrix multiplication application to various system and algorithm related issues. We also conducted experiments on more general types of applications using a generic workload. However, for the sake of brevity, we present the results for the matrix multiplication application. Results for the generic workload are available in [9].

All algorithms are based on a 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.

All 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 to do. The same code is used to multiply matrices in every algorithm. The first matrix is broadcast to all slaves. Performance of a load sharing algorithm is a function of task size t (i.e., granularity of the task). For example, if the 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 $t = C/S$, where C is the number of columns in the second matrix and S is the number of slaves. Such a scheme results in no load sharing.

We divide the algorithms into three classes: *fixed granularity*, *variable granularity* and *adaptive granularity*. In fixed granularity algorithms, task granularity is fixed statically. In variable granularity algorithms, task granularity decreases as the computation progresses. While both these types of algorithms take the system state into account implicitly, task granularity itself is not dependent on load variations in the system. The adaptive task granularity algorithms vary task granularity depending on the current system load. We now describe five load sharing algorithms belonging to these three classes.

3.1 Fixed Task Granularity Algorithm

We describe Send algorithm that belongs to this class. 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. Each slave multiplies the columns that it receives by the first matrix it already has. When a slave finishes the multiplication, it sends the results back to the master. Master collects the results, writes it into appropriate place of the result matrix and then sends another t columns of the second matrix to the slave. This process continues until all columns of the second matrix have been sent. Performance of other fixed granularity algorithms is discussed in [8, 9].

3.2 Variable Task Granularity Algorithms

We describe three algorithms that use variable task granularity. These algorithms have been proposed for multiprocessor systems for loop scheduling. All three are based on the Send algorithm. The difference between fixed and variable task granularity algorithms is that the task size t decreases with time. For example, first task may have a size of 10 columns while the last one may have only one column (see Figure 2). This can improve performance in two ways:

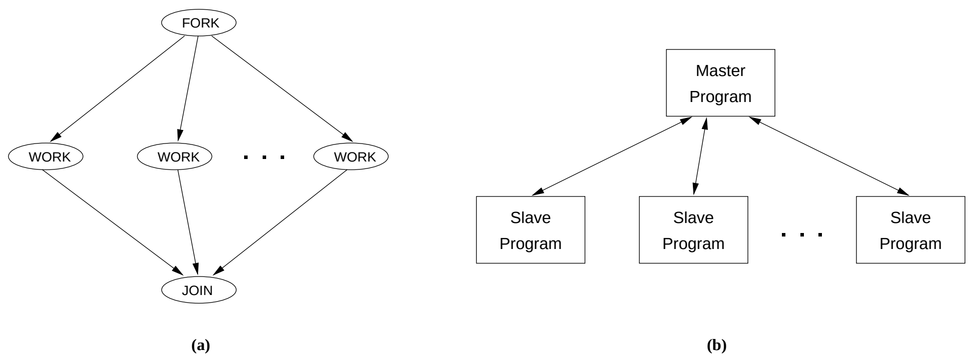

Figure 1: General structure of the algorithms (a) Fork-and-join structure; (b) Structure of the implemented algorithms

1. These algorithms attempt to minimize the time the master has to wait to receive the last results as task sizes at the tail end are much smaller.
2. It is well-known that sending several short messages results in worse performance than when a small number of large messages is sent. In case of matrix multiplication, depending on the parameters used, variable task size algorithms can potentially decrease the number of messages sent while increasing their average size. Even though the data in Figure 2 suggests that the variable task granularity algorithms tend to create more tasks than the fixed granularity algorithm, we show in Section 4 that the opposite is true for the parameters used in our experiments.

Several strategies have been proposed to decrease the task sizes [6, 10, 12]. The following three strategies were implemented: guided self-scheduling [10], factoring [6] and trapezoidal self-scheduling [12].

3.2.1 Guided Self-Scheduling

In guided self-scheduling (GSS), task size is a function of the remaining columns. Typically, task size is set to $1/G$ of the remaining columns. Figure 2 shows the task size distribution using $G = 4$ for a 70 column matrix. For example, we can select G to be equal to the number of slaves. Choosing G this way can lead to diminished load sharing as large tasks will take much more time on slow machines than on fast ones. Therefore, we have experimented with various G values. Section 4.1 presents these results.

3.2.2 Factoring

Factoring [6] creates smaller and more uniform task sizes than the guided self-scheduling. Tasks come in groups of S where S is the number of slaves. Each task in a group has the same size. Tasks size in each group is a constant factor of the size of tasks in the previous group. The optimal factor can be calculated based on the variance in task computation time [6]. However, in the type of systems that are being considered here, loads can vary dynamically, which makes it unreasonable to assume that variance is known *a priori*. As a good heuristic we set the task size in the subsequent group to be half of the previous group task size. Figure 2 shows the task sizes for a 70 column matrix. In this example, the initial task size F is set to 8 and the number of slaves is assumed to be 4. We have conducted experiments to study performance sensitivity to the initial task size F (see Section 4.1 for details).

3.2.3 Trapezoidal Self-Scheduling

In trapezoidal self-scheduling [12], task sizes decrease linearly unlike in factoring and guided self-scheduling where they decrease exponentially. Size of first task can be arbitrary. In our implementation, we have used the starting task size as total number of tasks/(2 * number of slaves). Figure 2 shows the task size distribution for a 70 column matrix. The number of slaves is assumed to be 4, which gives us the initial task size as 9 columns. The decrement value T used in this example is 1. We have experimented with various decrement values (see Section 4.1 for details).

A variation of these algorithms specifies the minimum size of a task. This can eliminate the long tail of tasks with size 1 (see Figure 2).

We have implemented these three algorithms (without the modification suggested above) in the Send type algorithm. For this reason, in our results section, we compare the performance of these algorithms with the Send algorithm.

3.3 Adaptive Task Granularity Algorithm

In this algorithm, the task granularity adapts to the slave node's response time. If a node is slow and/or heavily loaded, it tends to receive finer granularity tasks. The expectation is that performance can be improved by decreasing the number of messages while increasing their size. For example, if a node is fast, instead of it requesting several times fine granularity tasks, it could be given coarse granularity tasks fewer times. The algorithm works as follows: Initially, C columns are sent to each slave. Master measures

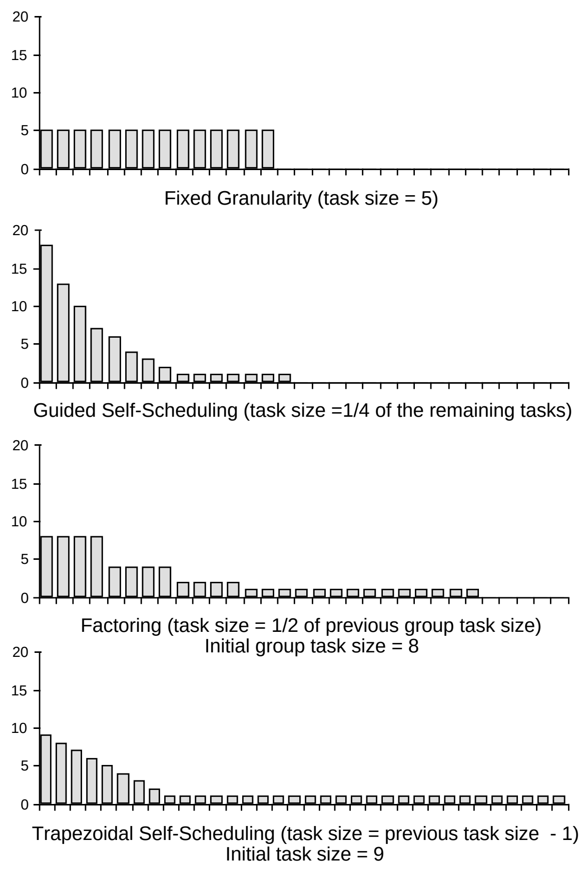

Figure 2: Task sizes for the fixed and variable task granularity algorithms (for a 70 column matrix)

the amount of time it takes for each slave to multiply the C columns and return the results. Next time a slave requests more columns, the slave's response time is compared with the average slave response time and the number of columns is adjusted based on this information. For example, if a slave's multiplication rate is 5 columns/unit time but the average is 10 columns/unit time, it will be given $C/2$ columns as the next task. In order to make sure that the task granularity will be within some reasonable bounds, the algorithm requires lower and upper bounds on the number of columns. Thus, the algorithm takes three parameters: the number of columns C, the minimum number of columns *min*, and the maximum number of columns *max* that can be sent to slaves.

4 Results and Discussion

This section presents the results of the experiments conducted to evaluate the performance of the five load sharing algorithms discussed in the last section. 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 multiplication algorithm requires 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 two controlled environments:

- *With no background load:* In this case, the entire system is dedicated to the matrix multiplication application. These experiments bring out the inherent tradeoffs associated with the algorithms. Further, it also represents the performance on an idle NOW (for example, running after mid-night). Also note that it has been observed that a large proportion of workstations are idle most of the time [7].

- *With background load:* In this case we introduce background load in a controlled manner so that we can obtain consistent results across multiple runs. The induced background load causes the processor to be shared between the matrix multiplication application and the background computation. Thus, the matrix multiplication application uses the processor half the time while the background load is active. To introduce variations in system load, we have turned the background load on for 1 second and off for 1 second cyclically. Note, however, that the background load does not induce network load.

Although we have conducted experiments with both slow and fast networks, for the sake of brevity, we report results only for the fast network. Other results are given in [9].

We use the execution time as the performance metric. The execution time values reported in this section are the averages of 30 runs (in most cases, with a standard deviation of less than about 1% of the mean value).

4.1 Results for Variable Granularity Algorithms

Variable task granularity has been shown to be useful for loop scheduling in multiprocessor systems [6, 10, 12]. These studies have shown that fixed task granularity algorithms perform worse than variable task granularity algorithms (by a wide margin depending on application). Since communication characteristics of multiprocessors and distributed parallel systems are quite different as discussed in Section 1, we want to investigate the effectiveness of variable task granularity algorithms. This section discusses the impact of varying task granularity on the performance of the matrix multiplication.

4.1.1 Relative Performance

This section compares the performance of Send and the three variable granularity algorithms. Figure 3 presents the results when the master node performs both coordination and computation (i.e., "master works" on matrix multiplication just like the other slaves) and when the master is restricted to coordination only (i.e., "master does not work"

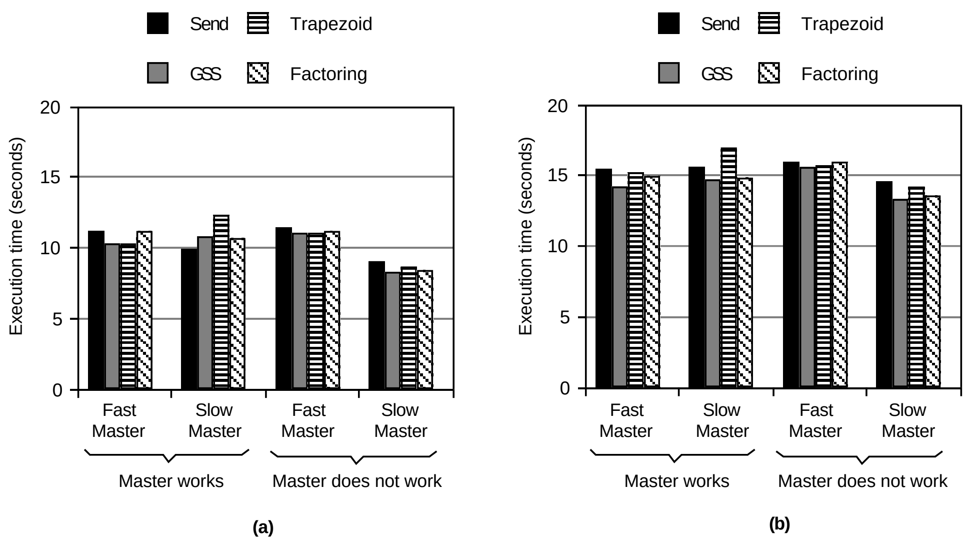

Figure 3: Performance of variable task granularity algorithms (a) Without background load; (b) With background load

on the multiplication). Send uses a fixed task granularity of 3 columns. In Factoring, task granularity starts with 40 columns and the task granularity of each task group is half of the previous group. In GSS, task granularity is set equal to 1/14 of the remaining tasks. In TSS, task granularity starts at 720/(2*9) = 40 columns and decreases linearly by 2 columns.

We can make the following observations from the data in Figure 3:

- *Fast master versus slow master:* Note that, in our set up, the fastest machine (200 MHz Pentium Pro) is about 3 times faster in performing the matrix multiplication than the slowest machine (133 MHz Pentium). We note the following from this data:
  - *When the master is dedicated to coordination only:* Mapping the master to the slowest node is better when master does not work on multiplication. The reason is that, if the fastest node is used for pure coordination task, it is taken out of the worker pool of nodes that actually perform the computation. Therefore, the execution time increases. This is true for all algorithms and whether there is background load on the system or not as shown in Figure 3.
  - *When master coordinates as well as computes:* The performance difference is less pronounced in this case as the master also participates in computing partial results.

- *Relative performance of variable granularity algorithms:* The data in Figure 3 suggest that, in general, the GSS appears to provide the best performance. However, in most cases, there is no substantial performance difference among the three algorithms. TSS performs worse when the master is mapped to the slowest node and master works on computation as well as coordination. This is due to the fact that TSS generates 320 tasks when T is 2 (about 300 of them have a task granularity of 1 column). With such large number of fine granularity tasks, the speed of the master becomes important in providing quick response to requests from the slaves. In contrast, GSS and Factoring generate 74 and 72 tasks, respectively. The performance of TSS is similar to the other two algorithms when $T = 1$ is used. In this case, TSS generates only 27 tasks. The performance sensitivity of these algorithms to their parameter values is discussed in Section 4.1.2.

- *Fixed versus variable granularity algorithms:* To compare the performance of fixed and variable task granularity algorithms, we use Send and the three variable granularity algorithms. Note that the Send algorithm with a task size of 3 columns generates 240 tasks. The data presented in Figure 3 suggests that, in most cases, the variable task granularity algorithms appear to provide a better performance than Send. However, the performance difference between Send and the variable granularity algorithms is not substantial. In some cases, Send is even better. A problem with variable task granularity algorithms is that they are sensitive to their parameter values (discussed in the next section). On the other hand, the data presented in [8] indicate that Send is fairly robust to task granularity.

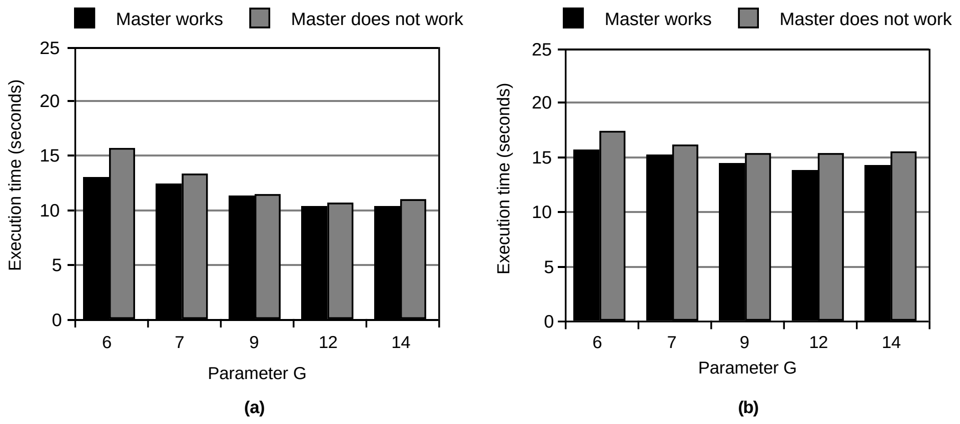

Figure 4: Performance sensitivity of Guided self-scheduling algorithm to parameter G (fast master) (a) Without background load; (b) With background load

4.1.2 Performance Sensitivity

GSS Algorithm

The GSS algorithm we implemented has a single parameter G. The results in the last section correspond to G = 14. The performance sensitivity of GSS algorithm to parameter G is shown in Figure 4. In this plot, we move towards finer granularity tasks as move to the right. These results show that the performance of the GSS algorithm is sensitive to the value of G. The optimum value appears to be about 12 whether or not there is background load. The sensitivity increases when the master is performing only coordination and is not working on the multiplication. For example, compare the execution times when G is 6 and 7. This is because coarse granularity affects the execution time and the impact is more severe when the master is taken away from the worker pool as it reduces the number of workers from 9 to 8. The execution time increases as we increase the G value (in our case, beyond 12). This is because we move towards finer granularity causing increased communication overhead.

Factoring Algorithm

The factoring algorithm we implemented uses a single parameter that sets the initial task size F. Choosing a larger F value moves the system towards coarse granularity. The performance sensitivity of Factoring algorithm to initial task size F is shown in Figure 5. As in the previous plot, moving to the right implies finer task granularity. For reasons discussed in the context of the GSS algorithm, there is an optimum task granularity that is obtained with an F value of either 30 or 40. Relatively, for the parameters considered, factoring exhibits more sensitivity to F value. In addition, the best result for factoring is worse than the best result for the GSS algorithm.

TSS Algorithm

The TSS algorithm uses a single parameter T that is the difference between successive task sizes. Thus, smaller T values results in coarse granularity tasks. The performance sensitivity of the factoring algorithm to decrement value T is shown in Figure 6. The performance of the TSS algorithm exhibits relatively marginal sensitivity to this parameter when the value of T is more than 1. The optimum value appears to be about 2.

4.2 Performance of Adaptive Algorithm

This section discusses the performance of the adaptive algorithm discussed in Section 3.3. Recall that this algorithm requires three parameters: a number of columns C that is the basis for the initial task granularity, minimum task granularity *min* columns, and maximum task granularity *max* columns. We use the notation (C, min, max) to represent these three parameters. We have implemented the adaptive algorithm on the basic Send algorithm. Figure 7 shows the relative performance of Send and the adaptive algorithms. The Send algorithm uses 3 column tasks and the adaptive algorithm uses (3, 1, 9) parameters.

It can be seen from these plots that, overall, the adaptive algorithms performs worse than the simple Send algorithm. While we do not want to generalize this conclusion just based on a single adaptive algorithm, we do offer some insights into the underlying behaviour that causes this relative performance. Adaptive algorithms assume that the past load information can be used to predict the future load and thus can be used to adjust the next task size. However, there may be several factors that may diminish this predictability capability. These include:

- The response time as seen by the master node includes the time required by the slave to complete the computation and the round-trip delay on the communication

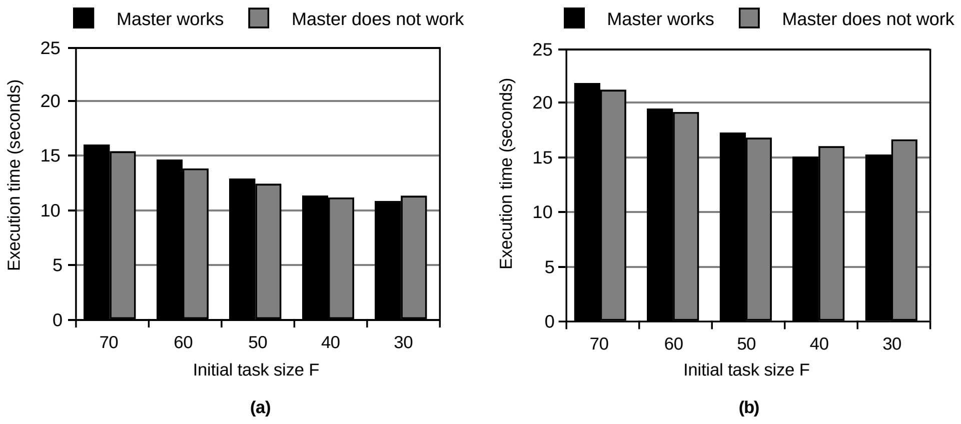

Figure 5: Performance sensitivity of Factoring algorithm to parameter F (fast master) (a) Without background load; (b) With background load

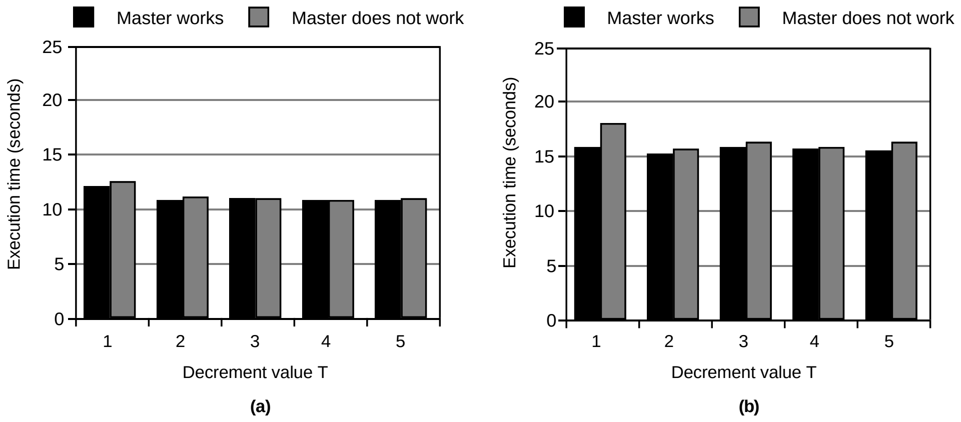

Figure 6: Performance sensitivity of Trapezoidal self-scheduling algorithm to parameter T (fast master) (a) Without background load; (b) With background load

network. Since the response time depends on the performance of these two system components, the capability of the response time to predict diminishes.

- The predictability of response time information also depends on the task granularity. As the task granularity increases, the system load (both CPU and network loads) changes slowly. However, the network traffic may be bursty and may not satisfy this requirement.

For these reasons we are not convinced that adaptive algorithms would work well for load sharing in this type of system. Further investigation is warranted to establish if adaptive algorithms would yield substantial performance benefit over simple load sharing algorithms; if so, what type algorithms are useful and under what system and workload conditions would they perform better?

5 Conclusions

We have conducted experiments to study the performance of five load sharing algorithms in a NOW-based parallel system using the PVM. These algorithms are: one fixed task granularity algorithm (Send), three variable task granularity algorithms (GSS, TSS, and Factoring), and an adaptive task granularity algorithm. We have used matrix multiplication to study the relative performance of these algorithms to various system and workload parameters.

Among the three variable task granularity algorithms, the GSS algorithm appears to perform the best in most cases. However, no substantial performance differences have been observed among the three variable granularity algorithms. In addition, the fixed granularity Send algorithm appears to provide performance comparable to that of the variable task granularity algorithms. A problem with variable task granularity algorithms is that it is important to set appropriate pa-

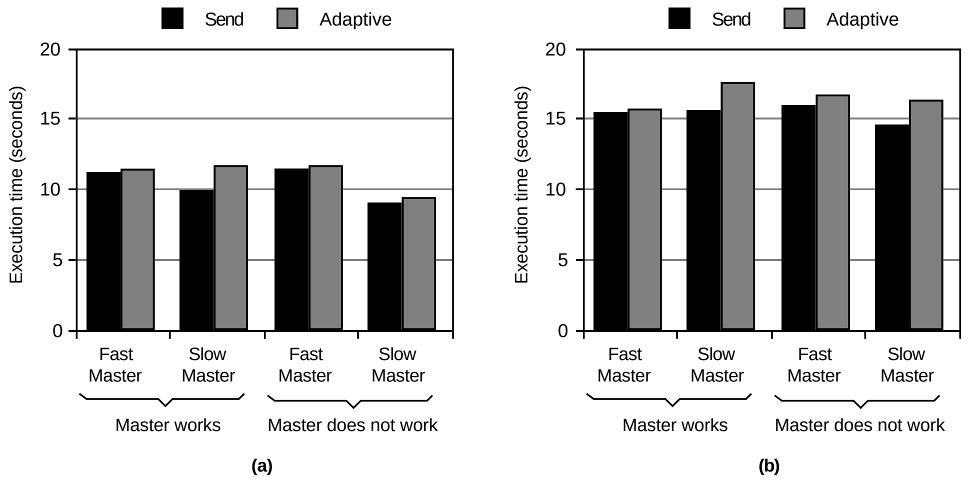

Figure 7: Performance comparison of Adaptive and Send algorithms (a) Without background load; (b) With background load

rameters to get good performance. Since NOW-based parallel systems are highly volatile in system load, it is probably difficult to predict appropriate parameters. Further investigation is required to study this problem.

The adaptive algorithm we have implemented did not provide performance improvements over Send. In fact, Send performed better in almost all cases. We have identified some possible reasons for this. The previous section identified some avenues for further research.

Acknowledgements

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

References

[1] I. Ahmad, A. Ghafoor, and G. Fox, "Hierarchical Scheduling of Dynamic Parallel Computations on Hypercube Multicomputers," *J. Parallel Distributed Computing*, Vol. 20, 1994, pp. 317–329.

[2] 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.

[3] S. P. Dandamudi and T. K. Thyagaraj, "A Hierarchical Processor Scheduling Policy for Distributed-Memory Multicomputer Systems," *4th Int. Conf. High Performance Computing*, Bangalore, India, December 1997.

[4] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "A Message Passing Standard for MPP and Workstations," *Comm. ACM*, Vol. 39, No. 7, July 1996, pp. 84–90.

[5] 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.

[6] 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.

[7] 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.

[8] A. Piotrowski and S. Dandamudi, "Performance of a Parallel Application on a Network of Workstations," *11th Int. Symp. High Performance Computing Systems*, Winnipeg, July 1997, pp. 429–440.

[9] A. Piotrowski, *Performance of Parallel Programs in Distributed Parallel Systems*, MCS Thesis, School of Computer Science, Carleton University, Ottawa, 1997.

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

[11] *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*).

[12] T. H. Tzen and L. M. Ni, "Dynamic Loop Scheduling for Shared-Memory Multiprocessors," *IEEE Trans. Parallel Dist. Syst.*, Vol. 4, No. 1, January 1993, pp. 87–98.