Performance Sensitivity of Space-Sharing Processor Scheduling in Distributed-Memory Multicomputers

Sivarama P. Dandamudi and Hai Yu
Centre for Parallel and Distributed Computing
School of Computer Science, Carleton University
Ottawa, Ontario K1S 5B6, Canada
sivarama@scs.carleton.ca

ABSTRACT - Processor scheduling in distributed-memory systems has received considerable attention in recent years. Several commercial distributed-memory systems use space-sharing processor scheduling. In space-sharing, the set of processors in a system is partitioned and each partition is assigned for the exclusive use of a job. Space-sharing policies can be divided into fixed, static, or dynamic categories. For distributed-memory systems, dynamic policies incur high overhead. Thus, static policies are considered as these policies provide a better performance than the fixed policies. Several static policies have been proposed in the literature. In a previously proposed adaptive static policy, the partition size is a function of the number of queued jobs. This policy, however, tends to underutilize the system resources. To improve the performance of this policy, we propose a new policy in which the partition size is a function of the total number of jobs in the system, as opposed to only the queued jobs. The results presented here demonstrate that the new policy performs substantially better than the original policy for the various workload and system parameters. Another major contribution is the evaluation of the performance sensitivity to job structure, variances in inter-arrival times and job service times, and network topology.

1. INTRODUCTION

Distributed-memory multicomputer systems are an important class of parallel processing systems for building large scale computer systems. The Intel Paragon, Teraflop and NCUBE2S systems are examples of commercial multicomputer systems. For instance, the Intel Teraflop system uses more than 9000 Pentium Pro processors to achieve 1 Teraflop processing rate. Similarly, the Ncube2S system can be expanded up to 8192 processors. These systems do not provide shared memory and use explicit message passing for communication among processors. In this paper, our focus is on processor scheduling policies suitable for such large multicomputer systems.

It has been observed that processor scheduling policies for multiprocessor systems should facilitate sharing the processing power equally among the jobs [5,7]. Due to the presence of multiple processors in these systems, processor sharing can be done in one of two basic ways: sharing them *spatially* or *temporally*. Policies that belong to the first category are called space-sharing policies in which the system of P processors is partitioned into N partitions and each partition is allocated for the exclusive use of a job. Sharing processing power equally among the jobs implies using equal partition sizes. In policies that are temporal (called time-sharing policies), jobs are not given the exclusive use of a set of processors; instead, several jobs share the processors as in, for example, round robin fashion. Researchers have studied both types of policies extensively. Several commercial distributed-memory systems such as the Intel iPSC system use space-sharing processor scheduling policies. The focus of this paper is on the space-sharing policies.

Space-sharing policies can be divided into fixed, static or dynamic categories. In fixed space-sharing policies, the partition size is fixed on a long-term basis. For example, partition sizes could be determined based on the expected workload characteristics at the system start-up time. The advantage of this type of policy is that the implementation is simple. However, a major disadvantage is that it does not adapt to changes in the system load conditions and resource requirements of jobs. For example, there can be a mismatch between the allocated partition size and the processor requirement of the job (this is referred to as the *internal fragmentation*).

Static policies eliminate the mismatch between a job's processor requirements and the partition size. In these policies, partition size is determined at the time of allocation. Thus, a match can be achieved eliminating the wastage of resources that is a major drawback of the fixed policies. However, as in the fixed policies, once a partition is allocated to a job, it remains fixed during the lifetime of that job. Several static policies have been proposed for distributed-memory systems [2, 9, 10].

While static policies are better than the fixed policies, there is still the fragmentation problem. One main reason is that the allocations are made for the lifetime of a job. Dynamic space-sharing policies eliminate this problem. The idea behind dynamic policies is that idle processors should be taken away from a job and allocated to another job that can fruitfully utilize the processor cycles [8]. One way to reduce the overhead (associated with taking a processor away from a job and allocating it to another job) in distributed-memory systems is to wait until the computation reaches a "desired" point such that the overhead involved in taking a processor away is small. This implies that the application notifies the central allocator that it is "willing to yield" a set of processors once the computation has reached a desired point (e.g., end of one phase of the computation).

For distributed-memory systems, dynamic policies incur high overhead. Thus, static policies are considered as these policies provide a better performance than the fixed policies. Several static policies have been proposed in the literature. In this paper we identify the deficiencies of a previously proposed policy and propose a new policy and study its performance under various workload and system parameters. The results presented here demonstrate that the new policy performs substantially better than the best policy proposed in the literature. Another major contribution is the evaluation of the performance sensitivity to job structure, variances in inter-arrival times and job service times, and network topology.

The remainder of the paper organized as follows: Section 2 describes the proposed space-sharing policy. Details about the job models and the workload are given in Section 3. Performance of these polices is presented in Section 4. The paper concludes with a summary.

2. SPACE-SHARING POLICIES

2.1. Basic Adaptive Policy

As explained in the last section, fragmentation is a serious problem with space sharing. In this section we describe the space sharing policies proposed by Rosti et al [9]. Rosti et al. try to reduce the impact of fragmentation by attempting to generate equal sized partitions while adapting to transient workload. The basic adaptive policy is shown below:

Given: a specific implementation of *compute_target_size*

target_size ← *compute_target_size*

while (queue_length > 0) **and** (free_processors ≥ target_size) **do**

free_processors ← free_processors – target_size
queue_length ← queue_length – 1
schedule_a_job (target_size)

od

Rosti et al. have proposed five adaptive policies (AP1 through AP5) and evaluated their performance under various workload and system conditions. The "adaptive" nature of the policies adjusts to various workloads by computing the partition size of each job at schedule time. On each such partition, a program exclusively runs until completion. These policies differ in how the target size is computed (i.e., in the *compute_target_size* procedure). Overall, their AP2 has performed well under various workload and system parameters. For this reason, we have selected this adaptive policy in our study.

2.2. The Original Policy

In the AP2 policy, which we refer to as the adaptive policy (AP), the partition size is computed as follows:

$$partition\ size = \mathbf{Max}\left(\mathbf{1},\ \mathbf{ceil}\left(\frac{total\ \ processors}{queue\ \ length\ +\mathbf{1}}\right)\right)$$

where **ceil** is the ceiling function. The policy always reserves one partition for future job arrivals (that is why it uses (*queue_length* + 1) rather than *queue_length*). This reservation, therefore, alleviates the performance deterioration associated with non-preemptive first-come/first-served scheduling of jobs. However, from the expression above, we notice that AP tends towards premature queueing. For example, on a 32-node system, when two jobs arrive at an idle system, the first job is allocated 11 processors and the second 16 processors. However, on the arrival of a third job, a partition size of 16 is assigned for this job. As there are only 5 free nodes, the job waits until a partition is freed. The main reason for this behaviour is that this policy considers only the queued jobs to determine the partition size. This policy also contravenes the equal allocation principle mentioned in Section 1.

2.3. The New Policy

In order to obviate the problems of the original policy, we have modified the AP policy by taking into account the total number of jobs in the system, which is defined as the sum of the queued jobs and the scheduled jobs. This modification is shown below.

$$partition\ size = \mathbf{Max}\left(\mathbf{1},\ \mathbf{ceil}\left(\frac{total\ \ processors}{queue\ \ length\ +\mathbf{1}+f\ *S}\right)\right)$$

where S is the number of scheduled jobs and $0 \le f \le 1$. The value of f can be used to control the contribution of the already scheduled jobs to the partition size. Note that we get the original AP policy by setting f to 0. The motivation behind the modification is two fold. One, the goal of equal partition for each job is better realized by taking all the jobs into account (this is the case when $f = 1$). Two, it provides a good heuristic during processor allocations. We show that the modified policy provides appreciable improvement in performance (relative to the original policy). The amount of improvement obtained is a function of the parameter f, system load, and workload. We refer to this policy as the modified adaptive policy (MAP). Section 4 presents the performance sensitivity of this policy to parameter f.

2.4. The Eager Release Policy

One of the problems with the two previous policies is that they keep all the processors in a partition until the job is completed. This all-or-none release policy can potentially cause performance problems, particularly if a job's parallelism varies. However, we have noted that releasing and requesting processors during a job's execution, as in the dynamic policy described in Section 1, involves high overhead in distributed-memory systems. We have also noted in Section 1 that a computation can release processors when the computation reaches a convenient "yield" point. What this implies is that processor allocation in such a policy is done in phases that matches the phased behaviour exhibited by the application. In such a case, it is also possible to release some of the processors that the application will not use in the current phase without waiting for the application to reach the yield point. We call this Eager Release policy. The policy can be applied both to the original AP policy and the modified MAP policy. We report the performance implications of this policy on the MAP policy when the job structure exhibits variable parallelism.

3. Job and Workload Characterization

One of our major objectives in conducting this research is to study the effect of job structure on the performance of space-sharing. To meet this objective, we have considered three job structures as in [1]. This section presents three types of job structures (Section 3.1) and the workload model (Section 3.2) used in this study.

3.1. Types of Parallel Programs Considered

We have selected three job structures as they cover the spectrum of variable parallelism. The fork-and-join structure exhibits constant parallelism while the other two applications exhibit variable parallelism. In the divide-and-conquer job structure, parallelism increases initially to a maximum job parallelism and then decreases progressively. In the Gaussian elimination job structure, the job starts with its maximum parallelism and progressively decreases to 1.

3.1.1. Fork-and-Join Programs

In this type of algorithms, work can be decomposed into independent sub-computations. A simple example is the algorithms that consist of a single loop and each instance of the loop can be executed independently. The parallel implementation of these types of algorithms can be done by creating tasks to work on the sub-computations. The structure of this type of parallel job is shown in Figure 1(a), which shows one phase of the fork-and-join structure.

The fork-and-join (FJ) job structure, for example, is reasonable for the class of problems with a solution structure that iterates through a communications phase and a computations phase. This class is exemplified by the n-body simulations of stellar or planetary movements, in which the movement of each body is governed by the gravitational forces produced by the system as a whole [4]. In this case, the model used here can be considered to represent one computation phase. In this paper, only a single phase that involves one synchronization point is assumed. Similar job structure has been used in several previous studies [5, 7].

3.1.2. Divide-and-Conquer Programs

Large numbers of parallel programs have been developed that are based on the divide-and-conquer (DC) strategy, especially in the area of symbolic computation, where the computation works on a discrete data structure. In general, the divide-and-conquer strategy does not restrict the number of sub-problems into which a given problem is partitioned. However, for convenience, we focus only on the binary case, where a given problem is divided into two sub-problems. Thus, the parallelism in these programs steps through the sequence $2^0, 2^1, \ldots, 2^{n-1}, 2^n, 2^{n-1}, \ldots, 2^1, 2^0$ during their execution.

The parallel implementation of divide-and-conquer algorithms exhibits a partitioning structure shown in Figure 1(b) [6]. The tasks created by this type of programs perform one of the following three operations: 1) splitting the inputs (*DIVIDE* task)

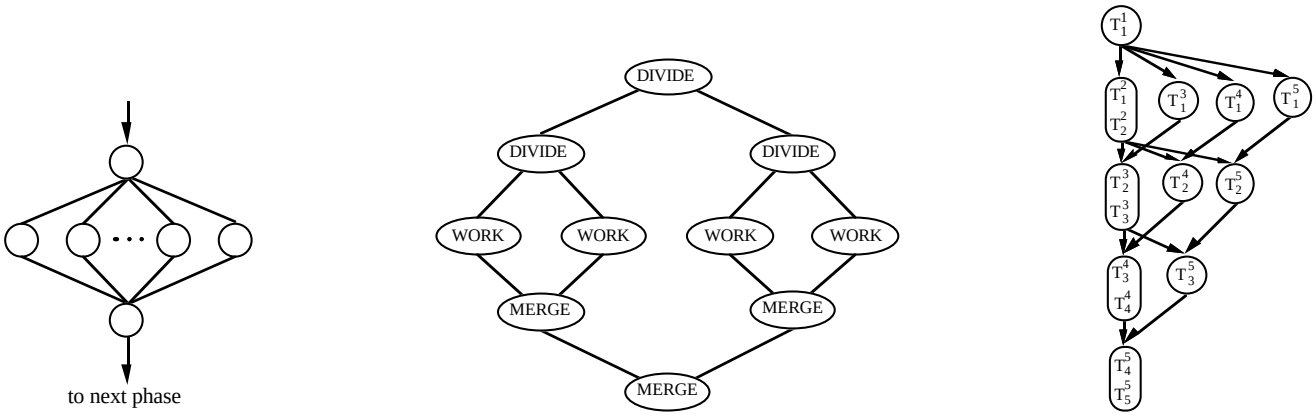


Figure 1 Three job structures considered (a) Fork-and-join (b) Divide-and-conquer (c) Matrix factorization

2) working on the inputs sequentially (*WORK* task) and 3) combining the outputs (*MERGE* task). Generally, the execution time of the tasks decreases level by level over the upper half of the structure, and increases level by level over the lower half of the structure. Synchronization is done at the *MERGE* stage. For some applications, such as the quick sort, MERGE stage does not involve much work; it simply provides synchronization. On the other hand, some applications, for example database queries, DIVIDE phase is trivial and MERGE nodes involves work (for example, sorting the answer).

3.1.3. Matrix Factorization Programs

To solve a matrix equation of the form $\mathbf{A}x = b$, one can decompose **A** into upper **U** and lower **L** triangular matrices and then solve it by solving $\mathbf{L}y = b$ and $\mathbf{U}x = y$. The most computationally intensive part of the algorithm is to find the LU decomposition. For this reason, parallelizing the LU decomposition has been studied by a number of researchers. One approach is to derive a task system for the LU decomposition algorithm and perform static scheduling on this task system. Other related algorithms include the Cholesky factorization and Gauss-Jordan algorithms. The structure of these parallel programs can be represented by a task system shown in Figure 1(c). T_i^j represents the task that works on column *i* in the j^{th} iteration.

However, the actual function of the tasks is different for each algorithm. In the LU decomposition and Gauss-Jordan algorithms, T_k^k finds the pivot element. For the Cholesky factorization algorithm, T_k^k computes the square root of the diagonal element and then divides the lower part of column *k* by this square root. In all the three algorithms, T_k^j uses the elements of column *k* to modify those in column *j*. Since matrix factorization is a way of performing Gaussian elimination, this class of programs is referred to as the Gaussian elimination (GE) programs.

3.2. The Workload Model

The abstract model for parallel programs consists of a set of input parameters along with a job structure. The job arrival process is characterized by two parameters: λ and CV_a. The job arrival rate is represented by λ and CV_a represents the coefficient of variation of the inter-arrival times. The other parameters are given below:

Maximum Parallelism (*mp*)
: The maximum parallelism of a parallel program refers to the maximum number of tasks that can be run in parallel, if an unlimited number of processors is provided. Since we assume that the tasks in a fork-and-join job can be executed independently, its maximum parallelism is always equal to the number of tasks the job has. For the divide-and-conquer job model, the maximum parallelism is achieved when the job is in its work stage. For the Gaussian elimination job model, the maximum parallelism is at the second stage of the program, where independent tasks are created to work on each column of the matrix.

Mean Service Demand (*d*)
: There are two ways to model the service demand of a parallel program [5, 7]. One is to compute the overall service demand of a parallel program and distribute the overall service demand among the tasks of the program. We can also compute the service demand of each task of a parallel program and sum them up to get the overall service demand. The workload created from the first approach is called *uncorrelated* because the overall service demand of a parallel program is independent of the number of tasks. On the other hand, the model based on the second approach is called *correlated* because the overall service demand of a job is directly proportional to the number of tasks.

  For the uncorrelated workload, the parameter *D* represents the mean service demand of a parallel program. For example, if it is set to 20, then, on average, a parallel program will require 20 units of service time from the system. For the correlated workload, *d* is the mean service demand of the tasks in the parallel programs. For example, if *d* is equal to 1, then, on average, every job has a mean task service time equal to 1 time unit. The mean task service time is the sum of the service time of the tasks divided by the number of tasks in the job. We have conducted experiments with both correlated and uncorrelated workloads. The results are qualitatively similar. Therefore, for the sake of brevity, we report results for the uncorrelated workload only.

Coefficient of Variation of Service Demand (CV_d)
: CV_d is the coefficient of variation of service demand of parallel programs, if the uncorrelated workload is used. It is the coefficient of variation of service demand of the tasks, if the correlated workload is used. Measurements at some supercomputer centers indicate that the service time variance can be very high (as high as 30 in some cases) [3].

Synchronization Cost (syn)
: This parameter represents the processor service time required to perform synchronization. In the divide-and-conquer workload, both the divide and merge are assumed to take the same amount of processor time and is represented by this parameter. For the Gaussian elimination, this parameter represents the processor service demand for each serial phase T_k^k.

4. PERFORMANCE COMPARISON

We have conducted a detailed simulation study of the three job structures described in Section 4. This section presents results of the simulation experiments to show performance sensitivity of space sharing to job structure and to demonstrate the performance superiority of the modified adaptive policy. Unless otherwise stated, the following default parameter values are used: $f = 0.5$, $mp = 32$, $D = 16$, $CV_d = 3.5$ and $CV_a = 1$ for all three job models. The parameter *syn* is set to 0.1 for the FJ and DC models and to 0.2 for the GE job model. The *syn* is set to 0.1 in DC and to 0.2 in GE because DC application has two phases - one divide and one merge - that are mirror images. For the FJ application, we have set *syn* to 0.1 as we want to model only a single synchronization phase.

The maximum parallelism of a job is varied from 1 to *mp*, which is 32. The default service time CV is fixed 3.5 as empirical observations at several supercomputer centers indicated this to be a reasonable value [3]. However, these studies have also shown that the service time CV can be very high as well (as high as 30 in some cases). The arrival CV is fixed at 1 (i.e., we assume Poisson arrivals). However, we study the sensitivity of the performance to these variances in the next section. All simulation experiments assume a 64-processor system.

4.1. Adaptive Policy versus Modified Adaptive Policy

The relative performance of the AP and MAP policies is shown in Figure 2. The x-axis gives the system utilization (in %) and the y-axis represents the response time difference between the two policies and is computed as follows:

$$\text{Performance improvement (\%)} = \frac{RT_{AP} - RT_{MAP}}{RT_{MAP}} * 100$$

where RT_{AP} and RT_{MAP} are the response times of the AP and MAP policies, respectively. The data in Figure 2 shows that the improvement in performance with MAP policy can be as high as 48%. Also, the performance difference tends to be smaller at low and high system utilizations. At very low system utilizations, both policies use the same partition size as the probability of a job being in service is small. However, as the system utilization increases, the number of scheduled jobs increases. This increase causes the new policy to use smaller partitions than the AP policy. Smaller partitions improve system utilization as it decreases, for example, internal fragmentation described in Section 1. At high system utilizations, both policies tend to use the same partition size - for example, at very high system loads, each partition may have only one processor under both policies.

The data in this figure also shows that the job structure has a significant impact on the performance. Overall, the GE job structure shows the highest improvement in performance with the MAP policy. The DC job structure shows the lowest. The improvement in performance for FJ job structure is similar to that with the GE application. The GE shows better improvement than FJ with the MAP policy because MAP tends to assign smaller partition (than AP) as it takes the number of scheduled jobs into account. Since the job parallelism of GE varies from the maximum parallelism to 1, smaller partition size decreases internal fragmentation as there is a serial phase (i.e., T_k^k tasks). Furthermore, notice that GE exhibits a phased behaviour where each phase (i.e., tasks at the same level) have a single synchronization point as in the FJ application. On the other hand, DC structure has several synchronization points (i.e., a pair of tasks participate in synchronization). In addition, the parallelism increases from 1 to the maximum value and then decreases to 1. These two characteristics cause the performance difference between AP and MAP policies to be smaller with DC than with the other two job structures.

For example, consider the GE and DC applications with a maximum parallelism of 4 tasks (as shown in Figures 1(b) and 1(c)). Suppose that only a 2-processor partition is allocated by the MAP policy as opposed to a 4-processor one by the AP policy. For GE, the four tasks at T_1^k (i.e., T_1^2 to T_1^5) will take twice the time than on a 4-processor partition. But during the serial synchronization phases T_1^1 to T_5^5, only one processor idles as opposed to 3 under AP. This reduction in internal fragmentation helps reduce the waiting time of jobs by assigning a partition to these jobs earlier under MAP policy. Thus MAP decreases the response times of the jobs. Running the same scenario with DC application shows that internal fragmentation is less in DC to start with as, for example, there are two DIVIDE and two MERGE tasks (unlike a single synchronization task in GE). Therefore, DC shows less improvement in response time than the GE application.

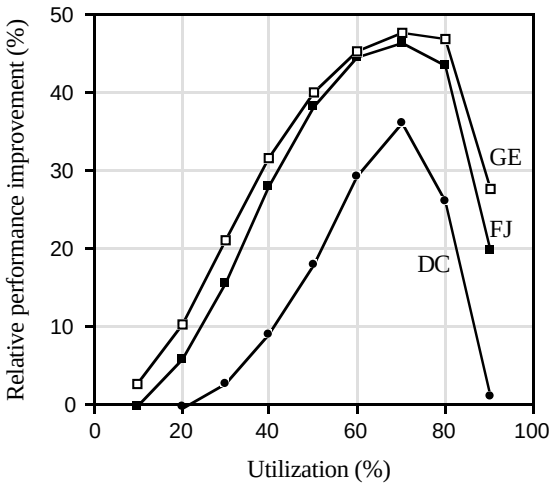


Figure 2 Relative performance of AP and MAP policies as a function of system utilization and job structure ($f = 0.75$)

To demonstrate that MAP policy consistently performs better than AP policy, we show the sensitivity of the two policies to variances in inter-arrival and service times. When the arrival CV is varied, the service CV is held at 3.5 and arrival CV is fixed at 1 when the performance sensitivity to service time CV is studied. The system utilization is fixed at 70%. Figures 3 and 4 show the relative performance of AP and MAP policies for the DC and GE job structures, respectively. Since FJ exhibits similar sensitivity to that shown for GE, we omitted it. We can make the following observations from the data presented in these two figures:

(a) MAP policy is always better than the AP policy. In some cases, the performance difference is marginal (for example, when the inter-arrival CV is high in Figure 3).

(b) Inter-arrival CV has more impact than the service time CV on both policies. Note that higher variance implies clustered arrival of jobs into the system. It also implies longer gaps in job arrivals (to get the same mean arrival value). This increase in variance affects performance in two ways.

- First, clustered arrival of jobs means jobs may have to wait longer to get to the execution stage. This increases the queueing time component of the response time.
- Second, clustered arrival also forces the scheduling policy to assign smaller partitions as clusters of jobs arrive into the system. But, the arrival pattern also includes long gaps (the higher the CV the longer the gaps). Thus, most jobs are stuck with the smaller partitions even though some processors are idle due to lack of jobs (because of the long gaps).

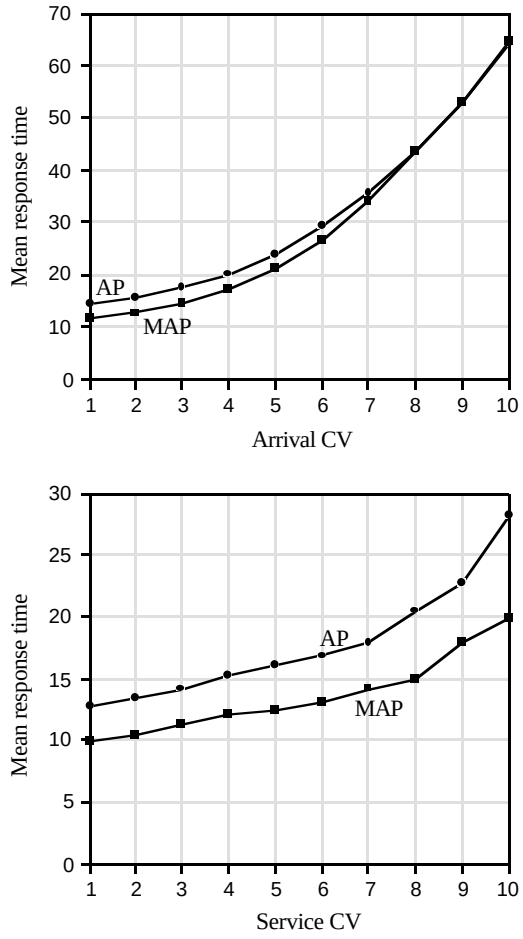


Figure 3 Performance of AP and MAP policies as a function of variance in inter-arrival and service times (DC application)

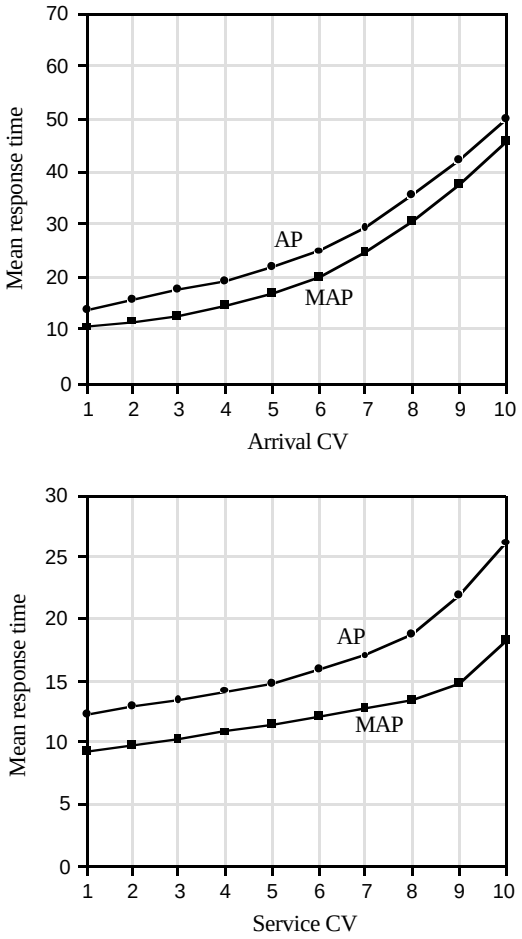


Figure 4 Performance of AP and MAP policies as a function of variance in inter-arrival and service times (GE application).

The overall effect of these factors is to decreases the efficiency of the system by leaving more processors idle. The efficiency decreases as the variance in arrivals increases, resulting in an increase in the response time.

(c) At high inter-arrival CV, the performance difference between the two policies decreases. This is because, higher CV implies that jobs are arriving into the system as clusters. The higher the CV, the more clustered the the arrival is. Thus, the queue length becomes the dominant factor in determining the partition size, resulting in decreased response time difference between the two policies.

(d) In contrast to the sensitivity to inter-arrival CV, the performance difference between the two policies increases with service time CV. This is because increased variance in job service times implies that there are fewer very large jobs and, therefore, taking the jobs currently receiving service would give a partition size that tends to reflect the system state more appropriately (by reducing the partition size).

4.2. Performance of Eager Release Policy

The impact of Eager Release policy is shown in Figure 5. All parameters are set to their default values. The difference between the standard policy and the Eager Release policy is that, if a job does not require the use of a set of processors before its termination, Eager Release policy immediately releases this set of processors such that other jobs can fruitfully utilize them. In contrast, the standard space sharing policies keep all processors until the job is complete. Eager Release policy is beneficial to applications that have decreasing job parallelism as in DC and GE job types. Since FJ does not have varying parallelism, we have not included it in Figure 5.

The y-axis in Figure 5 gives the relative response time improvement when the MAP policy switches from the standard release policy to Eager Release policy. This performance improvement (in percent) is computed as follows:

$$\frac{RT_{MAP} - RT_{Eager\text{-}Release\text{-}MAP}}{RT_{Eager\text{-}Release\text{-}MAP}} * 100$$

We make two observations for the this data:

(a) The performance improvement is better with the GE job structure compared to the DC application. The main reason is that the GE starts with the maximum parallelism and quickly starts releasing processors one at a time as the parallelism decreases from the initial maximum to the final 1. In contrast, the DC application will not release any processors during the first half as the parallelism increases during this phase.

(b) The relative improvement peaks at medium system utilizations. At low as well as high utilizations, the relative improvement is lower. It is low at low system utilizations as there may not be any jobs waiting to use the released processors. On the other hand, relative improvement is low at high system utilizations as the partition size tends to be small so the probability of releasing processors decreases.

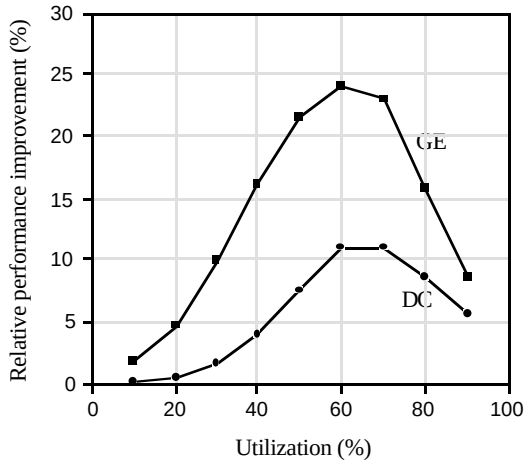


Figure 5 Impact of Eager Release policy on the performance of the MAP policy. The y-axis gives the response time improvement relative to that of MAP policy. Eager Release policy does not have any impact on the FJ application.

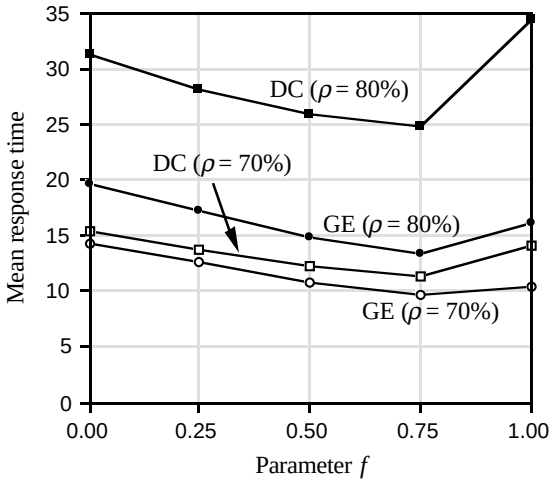


Figure 6 Performance sensitivity of the MAP policy to parameter *f*. Performance of the FJ application is similar to the GE application.

4.3. Sensitivity to Parameter *f*

The parameter *f* determines the contribution of the scheduled jobs to the partition size. In particular, when $f = 0$, MAP policy reduces to the original AP policy in which the partition size is determined solely by the waiting jobs. To see the impact of this parameter on the performance, we have conducted experiments for various values of *f* (in the range between 0 and 1) for all three job structures. Selected results of these experiments are shown in Figure 6. The mean response time for both DC and GE applications are shown when the system utilization is 70% and 80%. The results for the FJ applications is not shown as its performance is similar to that of the GE application.

It should be observed that at low system utilizations, we should assign larger partitions as there are fewer jobs in the system. This means the *f* value should be smaller at these utilizations. As the system utilization increases, the partition size should be reduced by increasing the *f* value. The results presented in Figure 6 suggest the best *f* value is around 0.75 when the system utilization is 70% or 80%. The difference in performance increases with system utilization. Furthermore, the DC application is more sensitive to using an appropriate *f* value. For this application, deviating from the optimum value results in severe performance degradation, particularly at high system loads.

Even though we have not presented the results for other system utilizations, in most cases, an *f* value of 0.75 appears to perform the best. However, in some instances, a value of 0.5 performed the best, but the relative difference between the response times for *f* values of 0.5 and 0.75 is small at low system utilizations.

The best value for the parameter *f* is dependent on several factors including the system utilization level, application structure, service time and inter-arrival time variances, whether the workload is correlated or not, and so on. In general, an *f* value in the range between 0.5 and 0.75 appears to be a reasonable choice. However, further study is needed to clearly understand the interplay among the various parameters that influence the *f* value.

4.4. Impact of Network Topology

In the previous sections, we treated the processors in the system as a pool. This assumption is valid if, for instance, the processors connected by a bus. But what happens if there is a different network? To study the impact of network topology, we model a 64-processor system that uses either a bus or ring type interconnection among the processors. As mentioned, in a bus-based system, we can treat the processor as a pool of processors. In this case, to assign a partition, we have to consider only the number of free processors in the system. On the other hand, when a ring network is used, we have to not only look at the number of free processors in the system but also check the condition they are all neighbors (i.e., connected component). For example, if we need a 8-processor partition and there are 10 free processors, but in groups of 7 and 3 processors, we cannot assign processors to the job. The adaptive policy, described in Section 2.1, is modified as follows to take the network topology into account:

max_set: the largest set consisting of all connected processors

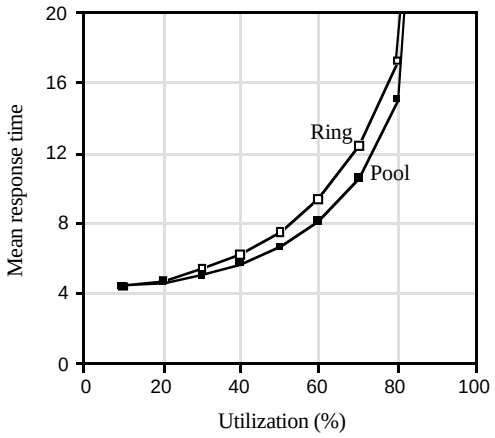
min_set: the smallest set consisting of all connected processors whose size is greater than or equal to the *target_size*

target_size ← *compute_target_size*

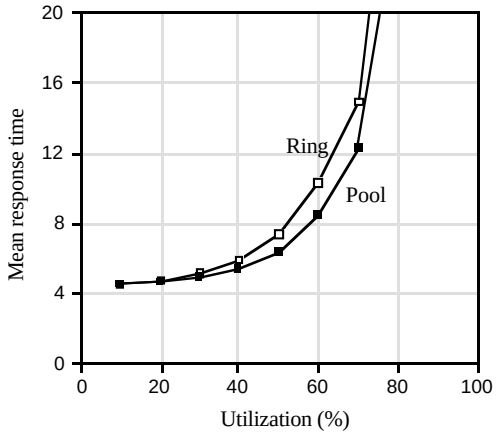
while (queue_length > 0) **and** (**max_set** ≥ target_size)
do
    search for **min_set**
    queue_length ← queue_length – 1
    schedule_a_job (**min_set**)
    search and set new **max_set**
od

The results for the three job types is shown in Figure 7. The MAP policy uses 0.5 as the *f* value. All other parameters are set to their default values. The main observation is that the network topology imposed constraints can seriously degrade performance at medium system utilizations. In this case, the mean response time degradation can be as high as 25%. At low and high utilizations, the degradation is small/negligible. At low utilizations, since most processors are free, the topology does not cause problems with processor allocation. At high utilizations, since the partitions is small (in the extreme case only one-processor partitions are assigned), again the network topology does not have any significant impact on the performance.

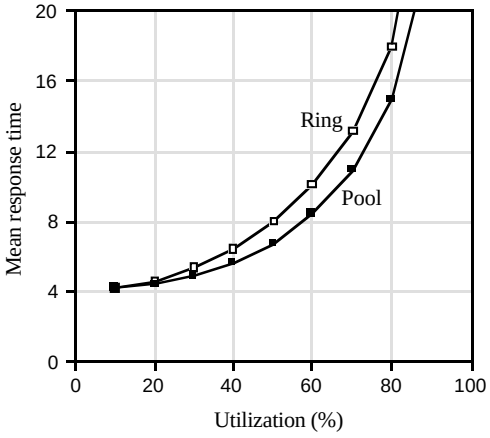
Even though we have used the ring network as an example, network topology-imposed constraints are serious in real systems. For example, the Intel iPSC and nCube systems are based on the hypercube topology. In these systems, partitions should form a subcube of the required size. These systems also impose an additional constraint: the partition size has to be a power of 2. This additional constraint causes a form of internal fragmentation.



(a) FJ



(b) DC



(c) GE

Figure 7 Impact of network topology on the performance of the MAP policy (AP policy exhibits a similar trend)

Our results indicate that significant performance gains can be obtained if we can treat the system as a pool of processors at medium utilizations, which is the range of interest for most systems. One way of reducing the impact of network topology is to use wormhole routing so that the network topology does not play a significant role in message passing. In such systems, it is possible to consider the processors connected by a specific topology to be treated like a pool of processors.

5. CONCLUSIONS

In this paper we have studied the performance sensitivity of space-sharing to various workload and system parameters. To conduct the study we have considered a previously proposed space-sharing policy and modified it to improve its performance. The results presented here indicate that the modified space-sharing policy consistently performs better than the previously proposed policy. The main point is that it is important to consider all jobs in the system (i.e., those waiting in the queue as well as those currently scheduled) in determining the partition size.

Another important contribution of this study is the sensitivity evaluation of space-sharing policies to the job structure. We have shown that the performance is sensitive to the job structure. While the absolute performance number vary significantly with the job structure, the trend in performance sensitivity is, however, largely independent of the job structure. An implication of this is that we can use the more simple fork-and-join type job structure to conduct performance sensitivity studies. In most cases, the conclusions obtained in such studies are valid with other types of job structure.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] S. L. Au and S. P. Dandamudi, "The Impact of Program Structure on the Performance of Scheduling Policies in Multiprocessor Systems,'' *J. Computers and Their Applications*, Vol.3, No.1, April 1996, pp. 17-30.

[2] Y. Chan, S. Dandamudi, and S. Majumdar, “Performance Comparison of Processor Scheduling Strategies in a Distributed-Memory Multicomputer System,” *Proc. Int. Parallel Processing Symp (IPPS)*, Geneva, 1997, pp. 139-145.

[3] D. G. Feitelson and B. Nitzberg, “Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860,” *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, and L. Rudolph (eds.), Vol. 949, Lecture Notes in Computer Science, Springer-Verlag, 1995, pp. 337-360.

[4] P. Jones and A. Murta, "Practical Experience of Run-Time Link Reconfiguration in a Multi-Transputer Machine," *Concurrency – Practice & Experience*, Vol. 1, 1989, pp. 171-189.

[5] S. T. Leutenegger and M. K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies," *Proc. of ACM Sigmetrics Conf.*, Boulder, 1990, pp. 226-236.

[6] S. Madala and J. B. Sinclair, "Performance of synchronous parallel algorithms with regular structures," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 2, 1991, pp. 105-116.

[7] S. Majumdar, D. L. Eager, and R. B. Bunt, "Scheduling in Multiprogrammed Parallel Systems," *Proc. of ACM Sigmetrics Conf.*, Santa Fe, NM, May 1988, pp. 104-113.

[8] C. McCann and J. Zahorjan, “Scheduling Memory Constrained Jobs on Distributed Memory Parallel Computers,” *Proc. ACM Sigmetrics Conf.*, Ottawa, Canada, 1995, pp. 208-219.

[9] E. Rosti, E. Smirni, G. Serazzi, L.W. Dowdy, and B.M. Carlson, "Robust Partitioning Policies for Multiprocessor Systems," *Performance Evaluation*, Vol. 19, 1994, pp. 141-165,

[10] S. Setia, M. Squillante, and S. Tripathi, "Processor Scheduling on Multiprogrammed, Distributed Memory Parallel Systems," *Proc. ACM SIGMETRICS Conf.*, May 1993, pp.158-170.