The Interaction between Memory Allocation and Adaptive Partitioning in Message-Passing Multicomputers *

Sanjeev K. Setia

Department of Computer Science
George Mason University
Fairfax, VA 22030

Abstract. Most studies on adaptive partitioning policies for scheduling parallel jobs on distributed memory parallel computers ignore the constraints imposed by the memory requirements of the jobs. In this paper, we first show that these constraints can have a negative impact on the performance of adaptive partitioning policies. We then evaluate the performance of adaptive partitioning in a system where these minimum processor constraints are eased due to the provision of support for virtual memory. Our primary conclusion is that any performance benefits resulting from the easing of minimum processor constraints imposed by the memory requirements of jobs will be negated by the overhead due to paging.

1 Introduction

In recent years, several adaptive partitioning strategies [6, 19, 5, 18, 17, 3, 14] have been proposed for scheduling parallel jobs on message-passing multicomputers. A key characteristic of these policies is that they reduce the number of processors allocated to individual jobs as the load on the system increases. The motivation behind this policy is to take advantage of the "operating point" effect [2]. Parallel applications typically experience a diminishing return in speedup as the number of processors allocated to them is increased. In a multiprogrammed environment, reducing the number of processors allocated to a job results in increased efficiency for that application and also frees up processors for use by other jobs. By allocating smaller partitions of the system's processors to jobs during periods of high load and bigger partitions at times of low load, adaptive partitioning policies can outperform policies that employ fixed-size partitions.

Most studies on adaptive partitioning, however, have ignored the constraints imposed by the memory requirements of parallel applications on the processor allocation policy. On message-passing multicomputers such as the Intel Paragon, the amount of memory available to a job depends upon the number of processors allocated to it. Therefore, the number of processors allocated to a job has to be large enough so that its data and code segments can fit in the local memories of

* This work was partially supported by NSF grant CCR-9409697

the processors. Thus, the ability of adaptive scheduling policies to take advantage of the "operating point" effect by reducing the number of processors allocated to jobs at high loads is constrained by the minimum processor requirement imposed by the memory requirements of jobs.

In this paper, we examine this interaction between processor allocation and memory allocation that arises while implementing adaptive partitioning policies. We first examine the implications of job memory requirements for the performance of adaptive scheduling policies. Next, we examine the issue of whether the provision of support for virtual memory can improve the performance of adaptive partitioning policies by easing the constraints imposed by memory requirements.

In uniprocessors, virtual memory support allows programs to execute with only part of their data and code in main memory. Similarly, in the case of multicomputers, virtual memory support *has the potential* to decouple (to some extent) processor allocation decisions from memory requirement considerations by allowing adaptive policies to allocate fewer processors than the minimum required to fit a program's data and code into memory. However, the performance benefits (if any) of this have to be weighed against the performance overheads of paging. This tradeoff is the main subject of this paper.

We note that the main motivation for providing virtual memory support in multicomputers is the convenience offered to the application programmer, and not to the system scheduler. In this paper, however, we examine whether the provision of virtual memory can also be taken advantage of by the system scheduler.

We consider the performance of two classes of adaptive job scheduling policies under various workload conditions. The first policy, which we call Adaptive Partitioning with Memory Constraints (APMC) always allocates at least as many processors to a job as are needed to fit its code and data into memory. Under the second policy, which we call Adaptive Partitioning with Virtual Memory (APVM), this minimum processor requirement is relaxed.

Using a simulation model, we first address the question: what is the maximum permissible overhead (due to paging) for individual applications in order for APVM to perform better than APMC? We then address the issue of whether paging overheads can be kept below this threshold. Using another simulation model, we examine the impact of factors such as the page fault rate, synchronization granularity and page fault service time on the overall paging overhead.

In the next section, we discuss related work on this subject. In Section 3, we describe the APMC and APVM policies in more detail. In Section 4, we compare the performance of the APMC and APVM policies. Finally, Section 5 summarizes our conclusions.

2 Related Work

Job scheduling strategies for parallel computers have been studied actively during the last few years [6, 19, 20, 22, 11, 18, 17, 3, 14, 12]. A recent paper by

Feitelson [8] surveys research in this area. Numerous studies have examined adaptive partitioning policies – among these, Parsons and Sevcik [15] and Chiang *et al* [3] consider preemptive versions of adaptive partitioning similar to the policy considered in this paper.

Only a few papers have examined the interaction between memory management and scheduling. Peris *et al* [16] developed analytical models for studying processor allocation tradeoffs while taking into account the impact of memory requirements. Our work differs from this study in that we examine the same tradeoff for a specific adaptive partitioning policy, taking into account factors such as fragmentation overhead. Secondly, our study also differs in that we take into account the impact of synchronization between threads on the overall paging overhead incurred by a parallel application.

McCann and Zahorjan [13] propose and evaluate several scheduling policies that take into account the constraints imposed by the memory requirements of parallel applications. Our work differs in that we consider memory-constrained adaptive partitioning policies, whereas they assume a dynamic scheduling discipline. Secondly, we also consider the feasibility of relaxing memory constraints.

Finally, some recent studies [1, 21] have examined the working set characteristics and paging behavior of scientific programs on distributed memory parallel computers.

3 Scheduling Policies

The policies considered in this paper have elements in common with both adaptive space-sharing policies that have been proposed in [18, 14, 17], and gang-scheduling polices[7, 8] in use on systems such as the Intel Paragon. They resemble gang-scheduling in that they are quantum-based policies in which all the processors in the system always context-switch synchronously at the end of a quantum. They resemble space-sharing policies in that the system's processors are divided into non-overlapping partitions that are dedicated to individual jobs for the the length of a quantum. More specifically, they can be classified as adaptive partitioning policies because each job arriving to the system is configured to execute on a certain partition size at the time it is first dispatched, and this decision is based on the job's requirements as well as current system load conditions.

It has been shown that for adaptive partitioning policies to perform well for workloads with high variability in job demand, either the policies have to be preemptive in nature [3] or they have to utilize user-supplied information about job demand [14]. In this paper, we assume that no information (about job demand) is supplied by the user and consider policies that use preemption. Thus, the policies proposed in this paper are time-sharing versions of the adaptive partitioning policies that have been proposed and analyzed in [18, 14, 17].

The system scheduler maintains a job queue consisting of jobs submitted to the system. Each job j arriving to the system submits a request for a minimum of min_j and a maximum of max_j processors, where min_j is equal to the minimum

number of processors required so that the job's data and code can fit into the local memories of the processors, while max_j is equal to the maximum number of processors desired by the application. At this time, the job is configured to execute on partition of p_j processors based on a decision procedure described below in Section 3.1, and is placed in the job queue.

The order in which jobs get dispatched in the system is based on a negative-feedback priority scheme similar to that commonly used in uniprocessors [4]. Each job has a priority associated with it which is inversely proportional to the processing time accumulated so far by the job. At the start of a quantum, the scheduler examines the job queue and selects the job with the highest priority to run with its configured partition size (i.e., p_j). It repeats this step as long as there are processors available. If the number of processors left after scheduling one or more jobs is less than the partition size of the job with the highest priority, it examines other jobs in the queue in the order of their priority until it finds a job whose partition size is less than the number of available processors. If there are no such jobs, the remaining processors are left idle [2].

All the jobs selected using the procedure above are dispatched at the beginning of the quantum. If a job completes before its quantum expires, the freed processors are allocated to the waiting jobs in a similar manner. At the end of a quantum, all the jobs executing in the system are preempted, their priorities are updated, and the scheduling procedure is repeated.

Under both the APMC and APVM policies, a job uses all the available memory of the processors in its partition. Under the APMC policy, when a job is dispatched, its data and code have to be loaded into the local memories of the processors in its partition. Similarly, when it is preempted, the current contents of the local memories have to be saved on disk. Under the APVM policy, we assume that the system keeps track of the active pages of memory and only loads those pages when a job is dispatched. Clearly, the length of a time slice has to be long enough to amortize these costs.

Periodically, the scheduler recalculates the priorities of all the processes in the system (e.g., by dividing them by 2) so that jobs are not penalized forever for past processor usage. The scheduler also maintains a variable L that captures the load on the system as reflected in the average number of jobs in the system, which it updates periodically[3].

---

[2] An underlying assumption here is that once a job is configured to execute on p_j processors, it needs at least that many processors to execute. While it is possible in some systems to use a two-level scheme whereby the virtual processors (or threads) of an application can be scheduled on any number of processors, this is not universally true.

[3] In our implementation of this policy, the scheduler periodically samples the number of jobs in the system, and updates this variable according to the formula $L_{i+1} = 1/2L_i + 1/2L_s$, where L_i,L_{i+1}, and L_s are the old, new and sampled values of L respectively.

3.1 Processor Allocation Policy

We consider two processor allocation policies, APMC and APVM. APMC assumes no support for virtual memory and thus the number of processors allocated to a job is at least min_j, while APVM assumes support for virtual memory and can allocate fewer than min_j processors.

Adaptive Partitioning with Memory Constraints(APMC) Under this policy, the processor allocation of a job p_j is determined as follows. Let $C = 2^i, i \geq 0$, such that $2^i \leq P/L < 2^{i+1}$, where P is the number of processors in the system and L is average number of jobs in the system. Then if $min_j \leq C$, the job is configured to execute on C processors, otherwise it is configured to execute on $(k \times C)$ processors , where $k > 1$ and $(k-1) \times C < min_j \leq k \times C$. In other words, if $min_j > C$, the partition size of job j is equal to smallest multiple of C greater than min_j.

For example, consider a system with $P = 128, L = 5$. Then $C = 32$. Now if a job arrives with $min_j = 20$ it is configured to execute with 32 processors. On the other hand, if $min_j = 45$, it is configured to execute with 64 processors.

The basic idea underlying the algorithm above is that as the load, L, increases, the processor allocation p_j of arriving jobs is reduced, subject to the constraint that no processor is allocated fewer processors than can satisfy its minimum memory requirement, min_j. The motivation for allocating processors in units of C processors, and for selecting C such that it is a power of 2, is to minimize fragmentation as much as possible. (We note that we have assumed that P is also a power of 2 but it is possible to change the algorithm so that C is always a divisor of P).

Adaptive Partitioning with Virtual Memory (APVM). This policy is similar to the APMC policy except that the minimum number of processors that can be allocated to a job is not min_j but some fraction, f of min_j. The fraction f is a fixed parameter of the policy. For convenience, we denote a APVM policy with fraction f as APVM(f). Note that the APMC policy can be considered to be APVM(1).

For example, consider the same situations described above, i.e., $P = 128, L = 5$, and $C = 32$. Under the APVM(0.5) policy, a job with $min_j = 20$ would be allocated 32 processors, but a job with $min_j = 45$ would be also be allocated 32 processors. Under the APVM(0.75) policy however the job with $min_j = 45$ would be allocated 64 processors.

4 Performance Comparisons

We used a discrete event simulation to compare the performance of the policies described above under a variety of workload conditions. For all our simulations, we assumed that the underlying system was a message-passing multicomputer with $P = 128$ processors. We did not model the interconnection network. All the processors are assumed to have the same amount of local memory.

4.1 Workload Model

Jobs are assumed to arrive according to a Poisson process. Each job is characterized by the following random variables – (i) its minimum memory requirements (M), (ii) its maximum parallelism (N), (iii) its total processing requirement (D), i.e., execution time on one processor, and (iv) its speedup function (S).

In our simulations, we assume that $N = P$, i.e., each job can execute on 128 processors. The total processing requirement D is chosen from a hyper-exponential distribution to model the high variability that is expected in parallel supercomputing environments [9]. We assume that jobs can be considered to belong to two classes - small and large. Small jobs have a mean processing requirement of 300 seconds while large jobs have mean processing requirement of 3600 seconds. The probability of a job belonging to the class of small jobs is assumed to be 0.75. Within each class, the demand of an individual job is selected from an exponential distribution. These assumptions result in an average job demand (considering both classes) of 1125 seconds, with the coefficient of variation of job demand (C_D) of 2.05.

The speedup function used in our simulations is given by $S(p) = (1+\beta)p/(\beta+p)$. This speedup function has been used by several studies [13, 3, 5] and is shown in Figure 1. For a given number of processors (p) and given job demand (on one processor), the speedup function is used to compute the processing requirement of the job on p processors. In our simulations, we assume that β is uniformly distributed between 30 and 300, the range considered in [13, 3].

For the minimum memory requirements, M, of a job we assumed three distributions. Under the first distribution, we assumed that M is uniformly distributed in the range (1,128). Under the second distribution, we assumed that M is distributed in the range (1,64) with probability 0.75 and in the range (65,128) with probability 0.25. Under the third distribution, we assumed that M is uniformly distributed in the range (1,64). Henceforth, we refer to these three distributions for M as distributions A, B, and C respectively.

In order to evaluate the performance of the APVM policy, we have to model the overhead due to paging when a job is allocated fewer processors than its minimum processor requirement min_j. We define this overhead as the ratio of the increase in execution time (due to paging) when executing on p_j processors and the processing requirement on p_j processors. Thus, if the increase in execution time due to paging when a job is allocated p_j processors is denoted by $T_{paging}(p_j)$ and the execution time (neglecting paging) for the job on p_j processors is denoted by $T(p_j)$ then

$$Overhead(p_j) = T_{paging}(p_j)/T(p_j)$$

Here $T(p_j)$ is computed using the speedup function (S) and the job demand distribution (D) described above. Then, the run-time of job when allocated p_j processors is equal to $(1 + Overhead(p_j)) \cdot T(p_j)$.

For a given parallel application, the paging overhead depends upon a number of factors – working set size, physical memory allocated, page fault service time, synchronization behavior, and page replacement policy. In our simulations,

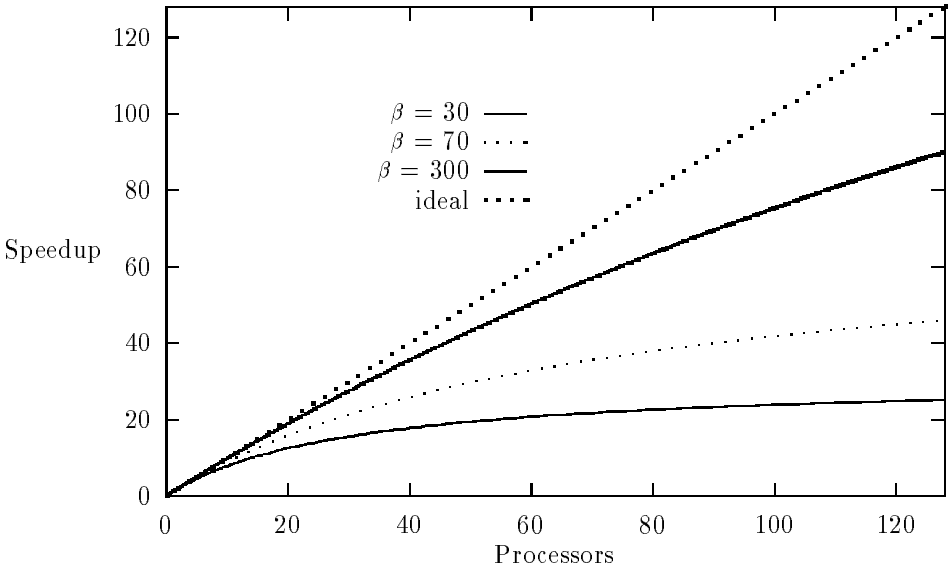

Fig. 1. Workload Speedup function

we make the simplifying assumption that the paging overhead of a job is a linear function of the fraction, denoted by c, of its memory requirement that is allocated to it. Thus, if a job j is allocated p_j processors, and the job's memory requirements are satisfied by min_j processors, then $c = p_j/min_j$. A given APVM(f) policy will allocate at least $f \times min_j$ processors to each job. Thus, under APVM(f), $c \geq f$. In our simulations, we assumed many different values for the overhead (O) for a job when allocated a fraction f of its memory requirement. For values of c between f and 1, we assumed that the overhead decreased in a linear fashion. Thus, the overhead for paging for a job that is allocated a fraction c of its memory requirement (i.e., $p_j = c \cdot min_j$ procesors) is given by

$$Overhead(c \cdot min_j) = \begin{cases} 0, & c \geq 1 \\ \frac{1-c}{1-f} \cdot O, & 0 < f \leq c \leq 1 \end{cases}$$

This is a simplification because when c falls below the working set size of the application the overhead increases exponentially. However, since we consider only values of $c \geq 0.5$, and consider a wide range of overhead values O, we feel that this model is a reasonable approximation for the purpose of comparing the APVM and APMC policies.

4.2 Scheduling Policy Performance

We now examine the performance of the APMC and APVM scheduling policies. Using regenerative simulation, we obtained the average response times for a

policy for a given set of input parameters. We considered job arrival rates (λ) corresponding to system utilizations ($\rho = \lambda \cdot \overline{D}/P$) of 0.1, 0.25, 0.4, 0.55, and 0.7. In each case, the simulations were run long enough to obtain 95% confidence intervals that were within 5% of the mean. For each policy, the quantum length was fixed at 2 seconds. In addition, the queue lengths were sampled and job priorities recalculated every 100 seconds.

We first examine the performance of the adaptive partitioning policy described in Section 3 *without* taking memory requirements into account. We will use this policy (which we henceforth call the AP policy) as our baseline policy in order to illustrate the impact of memory constraints on the performance of the APMC policy (Alternatively, the average response times obtained for AP can be considered to be that of the average response times that would be obtained for APMC for a minimum memory requirement (M) equivalent to that of one processor).

In Figure 2, we compare the performance of the AP policy with that of four gang-scheduling policies with fixed-size partitions. The four gang-scheduling policies examined are policies with partition sizes of 16, 32, 64, and 128 processors (denoted as GS(16), GS(32), GS(64), and GS(128) respectively). The operation of the gang-scheduling policies is identical to that of the AP policy, except that the partition size for all jobs is fixed *a priori* whereas under the AP policy the partition size of a job is determined using the procedure described in Section 3.1.

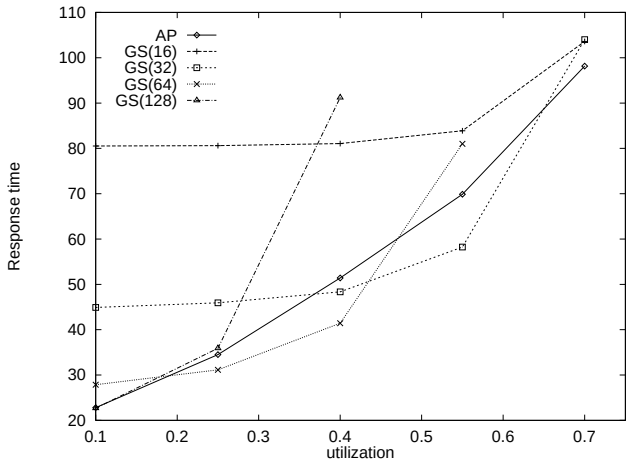

Fig. 2. Comparison of AP and Gang scheduling with fixed size partitions

The plots in Figure 2 show the importance of reducing processor allocation as the system load is increased, and the performance benefits of taking advantage of the "operating point" effect. This result is not new – it has been demonstrated in (by now) countless studies. However, we show this plot in order to illustrate two

points. First, the plots show that a time-sharing version of adaptive partitioning can be designed which performs reasonably well for a workload with a high coefficient of variation, and avoids the problems associated with a "pure" space-sharing version of the policy [14, 3]. Second, the plots in Figure 2 show that at moderate loads, the performance of the AP policy is worse than that of the best GS policy for a given load.

The reason for this behavior is that AP policy suffers from the effects of fragmentation whereas the GS policies do not. Under the AP policy, situations can arise where there are idle processors in the system, but jobs in the system queue cannot execute on these processors because their configured partition size is greater than the number of idle processors. This situation arises because at a given time, there can be jobs with different partition sizes executing concurrently. Under the GS policies, this situation can never arise since all jobs have the same partition size. This fragmentation effect is mainly due to our assumption that once a job has been configured for a certain number of processors it needs exactly that number to execute. If we assume a more dynamic environment (such as that considered in [13]), in which the job can adapt to the number of available processors, the performance of the AP policy should improve.

Next, we consider the impact of the memory requirements of arriving jobs on the performance of the APMC policy. In Figure 3, we plot the average response times for the APMC policy for the memory requirement distributions A, B, and C (described in Section 4.1) as a function of system utilization. We also plot the response time of the AP policy, i.e., a policy without memory requirement constraints. The plots show that the memory requirement constraints of the jobs in the workload can have a significant impact on the performance of an adaptive partitioning policy. The performance benefits of the APMC policies arise from their ability to take advantage of the "operating point" effect by reducing the partition sizes allocated to jobs. If this ability is restricted due to memory constraints, their performance degrades and the policies saturate quite quickly.

Another factor that contributes to the performance degradation of the APMC policies is the fragmentation effect mentioned above. When jobs have widely ranging minimum processor requirements (as is the case for memory requirement distributions A and B) the number of jobs with different partition sizes executing concurrently in the system increases, resulting in a higher fragmentation overhead. Unlike the (hypothetical [4]) situation described above (for Figure 2) where the fragmentation can be reduced if jobs can adapt to varying numbers of processors, in this case the ability of a job to adjust its partition size is restricted by the fact that it cannot reduce its partition size below its minimum memory requirement, min_j. This suggests that in order to reduce the effects of fragmentation in memory constrained environments, it may be necessary to provide language and run-time support for a programming model that allows dynamic application reconfiguration. Such dynamic run-time environments would allow the scheduler to use algorithms that avoid fragmentation, such as those

[4] because we do not take memory constraints into consideration

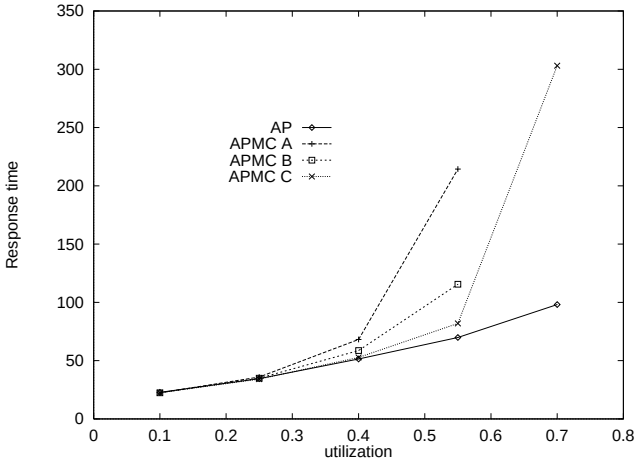

Fig. 3. Comparison of Adaptive Partitioning policies for different memory requirement distributions

described in [13].

At this point, a natural question that arises is: if the minimum processor constraint imposed by a job's memory requirements is relaxed as a consequence of the provision of support for virtual memory, how does it affect the performance of adaptive partitioning policies? If the scheduler is allowed to allocate fewer processors to a job than its minimum memory requirement, it could potentially use this flexibility to make decisions that reduce fragmentation and increase efficiency in situations with high system load. However, this flexibility comes at the cost of overhead due to paging. Thus, the key question is: how much overhead due to paging can be tolerated by individual applications (on average) before any performance benefits resulting from the easing of minimum processor constraints get negated?

To answer this question, we examine the performance of the APVM policy. We consider two APVM policies – APVM(0.5) and APVM(0.75). As discussed earlier, under the APVM(f) policy, the minimum number of processors allocated to a job has to be large enough so that at least a fraction f of its code and data can fit into memory. Thus the APMC policy is equivalent to APVM(1). In Figures 4–6, we plot the average response times of the APVM(0.75) policy for three overhead values (O) – 0%, 25%, and 50% – for the three memory distributions A, B, and C respectively. As described earlier, the plots for the 25% and 50% overheads represent the performance of the APVM(0.75) policy if the overheads for any job that is allocated 0.75 of its minimum processor requirement are 25% and 50% respectively. The 0% plot is obviously not realizable but is shown for the purposes of comparison.

The figures show that the APVM policy can provide a performance benefit at moderate to high loads provided the overhead of paging per job is 25% or less. The plots for the 50% overhead lie above that of the APMC policies in all

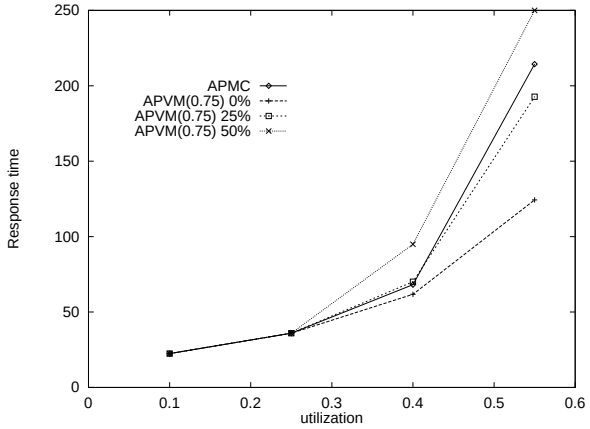

Fig. 4. Comparison of APVM(0.75) and APMC for memory requirements distribution A for different values of paging overhead O.

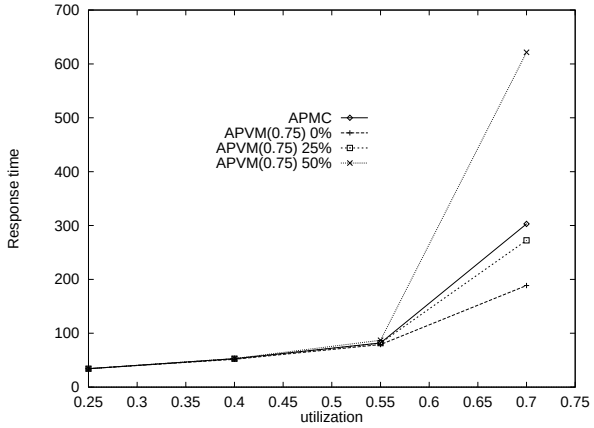

Fig. 5. Comparison of APVM(0.75) and APMC for memory requirements distribution B for different values of paging overhead O.

the figures, while the plots for APVM with 25% overhead lie below the APMC policy. As expected, the performance advantage of using APVM is greatest for distribution A and least for distribution C. Further, the advantages of APVM (for 25%) overhead) are only realized at moderate to high loads (0.4–0.7). Again, this is to be expected since at lower loads, the processor allocation decisions made by the APVM(0.75) and APMC policies are virtually identical with most jobs getting partition sizes larger than their minimum processor requirements.

In Figures 7 – 9, we compare the performance of the APVM(0.5) and APMC policies. The trends observed in these figures are identical to those observed in figures 4–6 for the APVM(0.75) policy. However, the performance advantage of

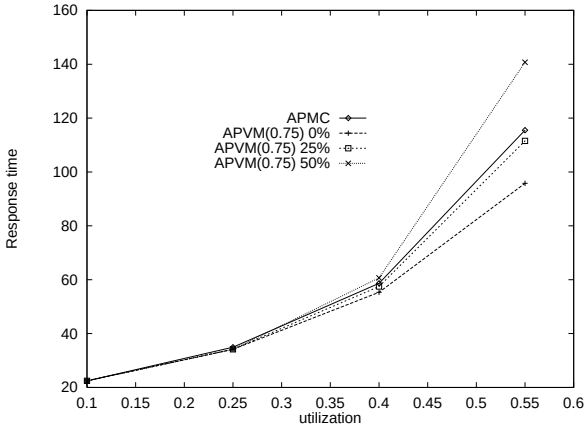

Fig. 6. Comparison of APVM(0.75) and APMC for memory requirements distribution C for different values of paging overhead O.

the APVM(0.5) policy for 25% overhead over the APMC policy is larger than the performance advantage of the APVM(0.75) policy for 25% overhead. This is because the APVM(0.5) policy provides greater flexibility to the scheduler than the APVM(0.75) policy. However, the overhead for the APVM(0.5) policy is also likely to be higher than that of the APVM(0.75) policy so a performance comparison cannot be made for these policies assuming the same overhead.

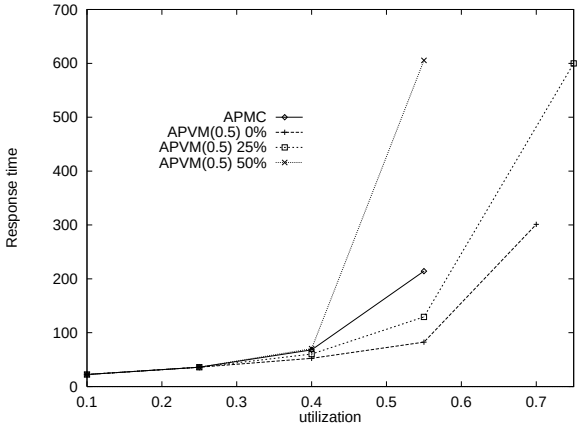

Fig. 7. Comparison of APVM(0.50) and APMC for memory requirements distribution A for different values of paging overhead O.

The results described above indicate that for the APVM policy to provide a performance advantage over the APMC policy, the overhead for paging has to

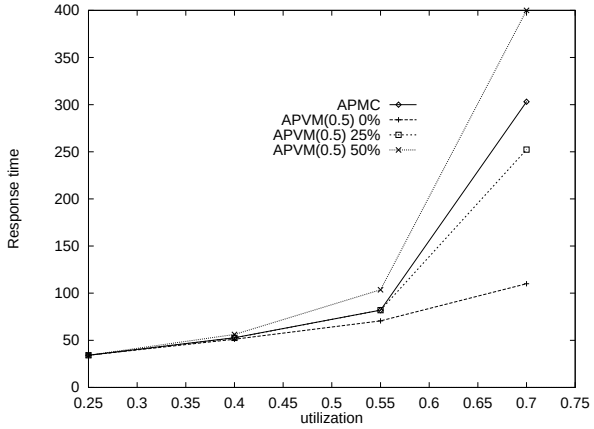

Fig. 8. Comparison of APVM(0.50) and APMC for memory requirements distribution B for different values of paging overhead O.

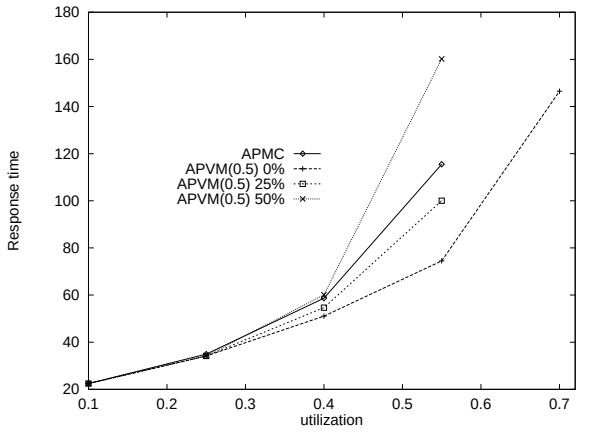

Fig. 9. Comparison of APVM(0.50) and APMC for memory requirements distribution C for different values of paging overhead O.

add at most 25% to the execution time of a job. In the next section, we consider the issue of whether it is reasonable to expect the overhead of paging to be below this threshold.

4.3 Paging Overhead

In uniprocessor systems, the paging overhead incurred by an application depends on the page fault rate and the page fault service time. The page fault rate in turns depends not only on the working set characteristics of the application and the amount of physical memory available in the system, but also the memory demands of *other applications*, and the page replacement policy. This is because

the physical memory of the system is shared among multiple applications that are executing concurrently. Under the APVM policy, however, there is only one application executing on a partition at a given time. Thus, the page fault rate on a system executing a parallel application depends on its working set size and memory reference pattern, and the amount of available memory.

Unlike sequential programs, however, the overhead incurred by a parallel application depends not only on the page fault rate but also on the synchronization patterns within the application. This is because when a thread that is part of a parallel computation has a page fault, any sibling threads that synchronize with the faulting thread are also delayed. Burger *et al* have shown that the slowdown experienced by several scientific applications when their available memory was constrained to 90% of their data set size ranged from a factor of two to a factor of eight. This suggests that achieving a slowdown of less than 1.25 (corresponding to the 25% overhead threshold in the previous section) will be very difficult, if not impossible. However, in this study, the backing store for the paged memory was assumed to be on disk, and an average disk service time of 16 ms was assumed. In this paper, we make the optimistic assumption that the average page fault service time is 1 ms. We note that researchers have proposed the use of dedicated "memory server" nodes [10] for implementing fast paging stores.

In the case of sequential applications, a page fault service time of 1 ms, and maximum overhead threshold of 25% would imply that the maximum page fault rate that is permissible (on average) is 250 page faults/second. In the case of parallel applications, the page fault rate that could be tolerated would be lower since the overhead of paging would also be affected by the synchronization between threads. Intuitively, the increase in overhead due to synchronization would be most dramatic for applications with fine-grain interaction between threads. In order to obtain a better understanding of the relative contribution of these factors – page fault rate and granularity of synchronization – on the overall overhead incurred by an application, we conducted a simple simulation-based experiment.

In this experiment, we simulated the execution of a SPMD parallel application with a simple *fork-join* structure, i.e., a SPMD application with multiple phases each involving a barrier synchronization. The inputs to the simulation that determine the synchronization behavior of the application are (i) the number of threads involved in the synchronization, and (ii) the granularity of synchronization (i.e., time between successive barriers). Further, we assume that the application was perfectly load balanced so that if none of the threads involved in the synchronization experienced a page fault, they would reach the barrier at the same time.

The inputs to the simulation that determine the paging behavior are (i) the page fault rate of a thread, and (ii) the *correlation factor*, which determines the correlation between the times at which page faults are incurred by different threads. We assume that the page fault rate of each thread is identical, and that the time between page faults is uniformly distributed with mean time equal to 1/(page fault rate). The correlation factor models the similarity in the pag-

ing behavior of different threads. (In case of SPMD programs it is reasonable to assume that there is some similarity in the memory reference patterns of the threads of the application [21].) Our assumption that the page fault rate of each thread is identical implies that each thread incurs the same number of page faults during the simulation. In our simulation, the *difference* in times at which a certain page fault occurred on different threads is uniformly distributed within time $(1 - correlation\ factor) * (1/page\ fault\ rate)$ of each other. Thus if the correlation factor is 1, all threads incur a page fault at exactly the same time. In this case, the overhead due to paging is solely due to the time taken to service the page fault, and the interaction between paging and thread synchronization patterns does not contribute to the overall delay incurred by the parallel application.

Using the model above, we simulated the execution of an application for a variety of input parameters. We varied the correlation factor between 0.5 and 1, the number of threads between 8 and 128, the synchronization granularity between 50 – 1000 microseconds, and the page fault rate between 10-500 page faults/sec. The metric of interest was the slowdown experienced by the application, i.e., the ratio of the execution time of the application with paging and the execution time without paging.

In Figure 10, we plot the slowdown experienced by an application with synchronization granularity 50 microseconds, and a page fault rate of 100 faults/sec, as a function of the number of processors (threads) involved in the barrier. Our results show that the slowdown experienced depends to a large extent on the correlation factors. For a correlation factor of 0.9, the slowdown can be as high as a factor 5 on 128 processors. We also note that even for a correlation factor of 0.99, the slowdown ranged between 1.38 and 1.5. We note that if the correlation factor is 1, the slowdown experienced by the application at 100 faults/sec will be 1.1. In Figure 11, we plot the slowdown for different correlation factors for an application with 64 threads and synchronization granularity 50 microseconds as a function of page fault rate. Again, we observe that the slowdown is quite high, e.g., for a correlation factor of 0.99 at 50 page faults/sec, the application experiences a slowdown greater than 1.4.

In Figure 12, we plot the slowdown experienced by an application with synchronization granularity of 500 microseconds, and a page fault rate of 100 faults/sec, as a function of the number of processors (threads) involved in the barrier. The plots show that even though the application has a relatively large granularity of synchronization, except for correlation factor of 0.99, the slowdown is much larger than 1.25. Figure 13 plots the slowdown for an application with 64 threads and synchronization granularity of 500 microseconds as a function of page fault rate. We observe that only the plot for correlation factor 0.99 lies below the desired threshold of 1.25.

These results indicate that the synchronization behavior of applications have a much greater impact on the overall paging overhead incurred by an application than the memory reference pattern (as reflected in the page fault rate). We are currently pursuing an experimental study to validate this conclusion.

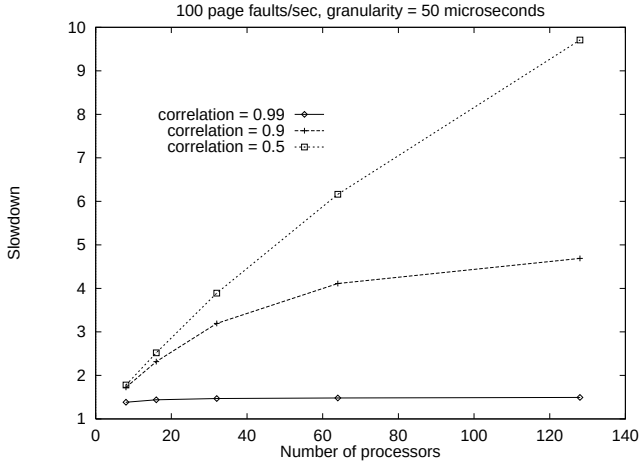

Fig. 10. Paging Overheads as a function of number of synchronizing threads for an application with fine-grain interaction.

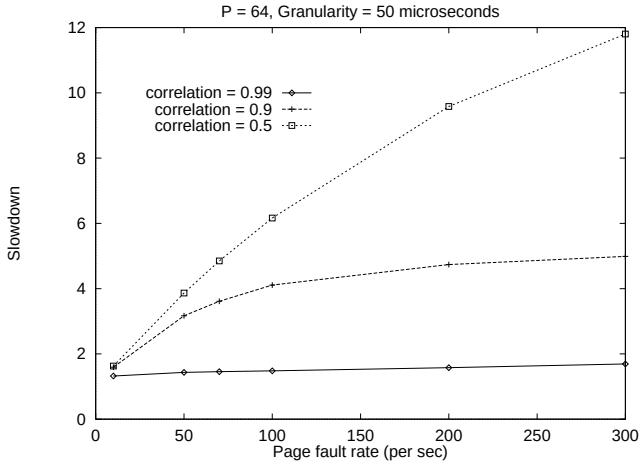

Fig. 11. Paging Overheads as a function of page fault rate for an application with fine-grain interaction.

Even though these results have been derived from a hugely simplified model of program execution, they do suggest that even for optimistic assumptions about paging overheads (e.g., page fault service times of 1 millisecond) and memory reference patterns (e.g., uniform page fault behavior across threads), the overall slowdown experienced by an application is large enough to overwhelm any performance benefits derived from decoupling processor allocation and memory allocation in the APVM policy.

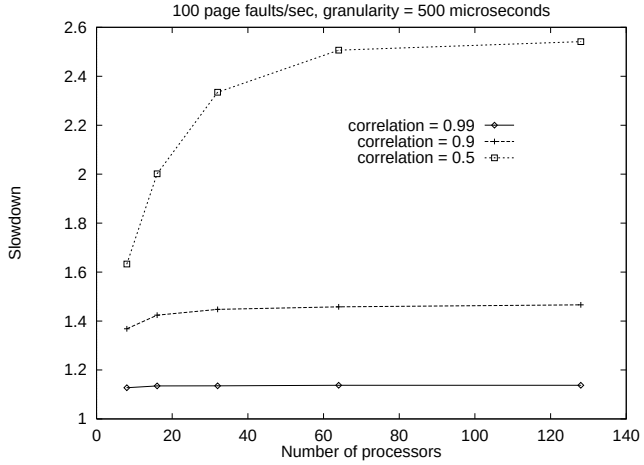

Fig. 12. Paging Overheads as a function of number of synchronizing threads for an application with coarse-grain interaction.

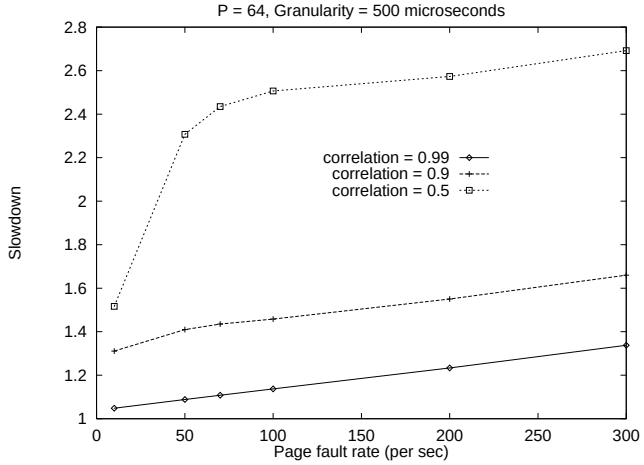

Fig. 13. Paging Overheads as a function of page fault rate for an application with coarse-grain interaction.

5 Conclusions

In this paper, we studied the interaction between processor allocation and memory allocation that arises in implementing adaptive partitioning on message-passing multicomputers.

We first showed that the minimum processor constraints imposed by the memory requirements of jobs submitted to the system can have a considerable negative impact on the performance of adaptive partitioning policies. This is due to two reasons (i) the increase in fragmentation overhead due to widely

ranging minimum processor requirements, and (ii) a decrease in the ability of the adaptive partitioning policy to reduce partition sizes at high loads in order to take advantage of the "operating point" effect.

We then considered the performance implications of decoupling processor allocation from memory allocation considerations (to some extent) through the provision of support for virtual memory. We showed that easing the minimum processor requirement imposed by a job's memory requirements could provide a performance benefit if the overhead due to paging was of the order of 25%. Next, we evaluated the impact of the various factors on the paging overhead incurred by a parallel application, and concluded that it would be difficult to keep the overhead below this threshold.

Overall, the main conclusion of this paper is that virtual memory offers little benefit to adaptive job scheduling policies. However, memory-constrained adaptive policies also perform poorly for workloads consisting of applications with widely ranging memory requirements. Thus, in order to obtain high system throughput for such workloads, dynamic partitioning policies, that can reduce the effects of fragmentation, are necessary.

References

1. D. Burger, R. Hyder, B. Miller, and D. Wood. Paging tradeoffs in distributed shared-memory multiprocessors. In *Proceedings of Supercomputing '94*. IEEE, November 1994.
2. Rohit Chandra, Scott Devine, Ben Verghese, Mendel Rosenblum, and Anoop Gupta. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of ASPLOS-VI*, pages 12–24. ACM, October 1994.
3. Su-Hui Chiang, Rajesh K. Mansharamani, and Mary K. Vernon. Use of application characteristics and limited preemption in run-to-completion parallel processor scheduling policies. In *Proceedings of 1994 ACM Sigmetrics Conference*, pages 33–44, Nashville, May 1994.
4. H. M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley, 1984.
5. L. Dowdy. On the partitioning of multiprocessor systems. Technical report, Vanderbilt Univ., Nashville,TN, July 1988.
6. D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38:408 – 423, March 1989.
7. D. Feitelson and L. Rudolph. Gang scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.
8. Dror Feitelson. A survey of scheduling in multiprogrammed parallel systems. Technical Report RC 19790, IBM Research Division, October 1994.
9. Dror Feitelson and B. Nitzberg. Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860. In *Proceedings of the IPPS 95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 215–227, April 1995.
10. Liviu Iftode, Kai Li, and Karin Peterson. Memory servers for multicomputers. In *Proceedings of the 1993 Spring CompCon*, pages 538–547, February 1993.
11. S. Leutenegger and M. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proc. of Sigmetrics '90*, May 1990.

12. Cathy McCann and John Zahorjan. Processor allocation policies for message-passing parallel computers. In *Proceedings of 1994 ACM Sigmetrics Conference*, pages 19–32, Nashville, May 1994.
13. Cathy McCann and John Zahorjan. Scheduling memory constrained jobs on distributed memory parallel computers. Technical Report UW-CSE-94-10-05, University of Washington, Department of Computer Science, 1994.
14. V. K. Naik, S. K. Setia, and M. S. Squillante. Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments. In *SuperComputing '93*, November 1993.
15. Eric W. Parsons and Kenneth C. Sevcik. Multiprocessor Scheduling for High-Variability Service Time Distributions. In *Proceedings of the IPPS 95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 76–88, April 1995.
16. Vinod G. J. Peris, Mark S. Squillante, and Vijay K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *Proceedings of 1994 ACM Sigmetrics Conference*, pages 5–18, Nashville, May 1994.
17. E. Rosti, E. Smirni, G. Serazzi, L. Dowdy, and B. Carlson. Robust Partitioning Policies of Multiprocessor Systems. *Performance Evaluation*, 9(2-3), 1994.
18. S. K. Setia and S. K. Tripathi. A Comparative Analysis of Static Processor Partitioning Policies for Parallel Computers. In *Proc. of MASCOTS '93*, January 1993.
19. K. C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proc. of the ACM SIGMETRICS Conf.*, May 1989.
20. A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proc. of the 12th ACM Symposium on Operating Systems Principles*, December 1989.
21. K. Y. Wang and Dan C. Marinescu. An analysis of the paging activity of parallel programs. Technical Report CSD-TR-94-042, Purdue University, Computer Sciences Department, June 1994.
22. J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *Proc. of ACM SIGMETRICS Conf.*, 1990.