Coordinated Allocation of Memory and Processors in Multiprocessors

Eric W. Parsons and Kenneth C. Sevcik
Computer Systems Research Institute
University of Toronto

{eparsons,kcs}@cs.toronto.edu

Abstract An important issue in multiprogrammed multiprocessor systems is the scheduling of parallel jobs. Most research in the area has focussed solely on the allocation of processors to jobs. However, since memory is also a critical resource for many parallel jobs, the allocation of memory and processors must be coordinated to allow the system to operate most effectively.

To understand how to design such coordinated scheduling disciplines, it is important to have a theoretical foundation. To this end, we develop bounds on the achievable system throughput when both memory and processing time are in demand. We then propose and simulate a simple discipline and relate its performance to the throughput bounds. An important result of our work is for the situation in which the workload speedup is convex (from above), but the speedup characteristics of individual jobs are unknown. It shows that an equi-allocation strategy for processors can achieve near-maximum throughput, yet offer good mean response times, when both memory and processors are considered.

1 Introduction

Multiprocessors are being used increasingly to support workloads in which some or all of the jobs are parallel. New scheduling algorithms are required to execute these workloads, allocating system resources to jobs with the goal of optimizing some objective function, such as minimizing mean response time. Past research in multiprocessor scheduling has tended to focus solely on the allocation of processors, even though limitations on other resources might also constrain performance. In fact, with current technological trends, processors are less likely to be the bottleneck resource in the future than either memory or I/O. Even with the large memory capacity of new machines, we can expect the combined memory demands of large-scale scientific applications to exceed capacity in multiprogramming environments [AKK+95]. Allocating processors to a job will not be effective if the scheduler does not also ensure that other resources the job requires are simultaneously available.

In this paper, we investigate the coordinated allocation of processors and memory in large-scale multiprocessors (which we term *processor-memory scheduling*). We derive bounds on system throughput when both memory and processors are needed for the execution of each job. These bounds provide, for the first time, some theoretical basis for assessing the performance of processor-memory scheduling disciplines. Although our primary objective is to minimize mean response time, understanding the throughput bounds is important for two reasons. First, it permits us to relate an arrival rate to the maximal sustainable throughput, enabling us to compare the performance of various disciplines under different workload conditions. Second, and more importantly, it provides new insight into how processor-memory scheduling disciplines must respond to increases in the load in order to avoid saturation.

Our first result in Section 3 applies to processor-only scheduling when the workload speedup is convex (i.e., has monotonically decreasing slope). It shows that a strategy where processors are evenly allocated among all jobs, a strategy which is already known to offer good response times, is also ideal from a throughput perspective if the scheduler has no knowledge of the speedup characteristics of individual jobs. Our subsequent result shows that this same strategy can also offer near-maximum throughput for processor-memory scheduling. We show that, although higher throughputs are theoretically feasible with more sophisticated schedulers, the gains that can be achieved are small and the computational costs high, making an equi-allocation strategy most attractive. Finally, we show that knowing speedup information of individual jobs is potentially more useful in processor-memory scheduling than in processor-only scheduling.

Based on these results, we have designed some simple disciplines for processor-memory scheduling aimed at minimizing mean response times. One such discipline is presented and evaluated here. We simulate this discipline under a variety of workloads and relate its throughput to the theoretical bounds we derive. We also examine how the response times for this discipline can be improved by choosing jobs for activation in such a way as to increase the average memory usage.

For our analysis, we make the simplifying assumption that job memory demands and speedup characteristics are uncorrelated. In practice, individual jobs will have speedup curves that are highly correlated with memory demand, as captured by memory-related scalability metrics [SN93]. However, such correlation may be significantly reduced in a highly varied workload, where our results are most clearly applicable.

In the next section, we summarize previous research that has been conducted on both processor-only and processor-memory scheduling. In Section 3, we present the throughput bounds and examine their implications in greater detail. The simulation results are presented in Section 4, and our conclusions in Section 5.

SIGMETRICS 96-5/96 Philadelphia, PA, USA

2 Previous Work

2.1 Processor-Only Scheduling

In multiprocessor scheduling, the efficiency with which a job utilizes its processors generally decreases as its processor allocation increases. As a result, a scheduling discipline must decrease the average number of processors allocated to each job as the load on a system increases in order to avoid premature saturation [Sev89]. Under the heaviest loads, each job should only be given a single processor (assuming memory is not a factor), as this maximizes the efficiency of the system.

Different systems provide various degrees of flexibility in allocating and reallocating processors. In this paper, we assume a system that supports *dynamic repartitioning*, where a job's processor allocation can be changed at any time during its execution. Although not commonly available in current system implementations, this level of flexibility can allow greatly improved system performance. For this reason, dynamic repartitioning is likely to be supported by more system implementations in the future.

Dynamic repartitioning can be achieved in two ways. First, if a job is initialized with many threads, then the operating system can vary the processor allocation by changing the number of threads running on each processor. Various techniques can be used to minimize the context-switch overheads associated with this approach, as long as threads do not synchronize too frequently [GTU91]. In message-passing systems, threads can be *folded* to keep the work assigned to each processor balanced [MZ94]. Second, the operating system can cooperate with the application when changing processor allocation so that the application can repartition and/or redistribute the data and change the number of active processes accordingly [TG89]. This allows the application to continue to execute efficiently after a change in allocation, even if threads synchronize frequently [FR92].

More constrained types of partitioning described in the literature are *fixed partitioning* and *adaptive partitioning*. In fixed partitioning, a system is pre-configured with a fixed set of partitions in which jobs can be executed. In adaptive partitioning, the system can allocate a job any number of processors up to the size of the system, but a job keeps the same allocation throughout its execution. Examples of this latter type of scheduling algorithm are PWS [GST91], ASP [ST93], and AVG [EZL89, CMV94].

With these non-dynamic disciplines, it is important to consider the service-time distribution of jobs. As in uniprocessor scheduling, if the variability in service times is large, which it generally is in multiprocessor workloads [FN95], then it is important that a discipline involving preemption be used [CMV94, PS95b]. Otherwise, long jobs can delay a large number of short jobs, leading to high mean response times.

2.2 Processor-Memory Scheduling

Compared to processor-only scheduling, there has been very little research conducted on processor-memory scheduling. In the context of an Intel Paragon system, McCann *et al.* examine how minimum processor allocation requirements due to memory might influence job scheduling [MZ95]. They suggest ways in which processors can be allocated so that each job receives exactly the same share of processing time. The metric by which disciplines are evaluated is the mean processor allocation; lower values are assumed to lead to higher processor efficiency values. The approach does not take into account jobs leaving the system, and it requires considerable computation just to ensure that each job receives exactly the same processing time in each scheduling cycle, when this may in fact not be necessary to obtain good response times.

Peris *et al.* propose a technique to model the paging behaviour of parallel jobs when operating with less memory than required [PSN94]. They apply their model to a real parallel job running on a parallel message-passing machine and study how the performance changes as a function of the processor allocation. Although this work does not consider the problem of scheduling jobs, it provides a model for varying processor allocations in a paging environment. However, it is still an open issue whether parallel scientific applications possess working set sizes smaller than their total memory demand [BHMW94].

Setia compares two preemptive, variable partitioning disciplines for distributed-memory machines, taking into account memory requirements of jobs [Set95]. The first always allocates a job as many nodes as it requires to satisfy its total memory requirements while the second relies on paging to increase the number of jobs that can be executed concurrently. He found that, with the paging overheads of current systems for scientific applications, scheduling disciplines that rely significantly on demand paging are not viable.

Finally, researchers from Tera Computer Company have described the way in which jobs with large memory requirements will be scheduled on the Tera MTA multiprocessor [AKK+95]. This work takes into consideration the overhead of swapping jobs in and out of memory, and presents an optimal algorithm that minimizes the overhead when swapping. The scheduler allows a space-time demand value to be associated with jobs, which it then uses to control the allocation of memory to jobs. No policy is defined for how these values should be set.

3 Bounds on the Throughput

In this section, we present analytic bounds on throughput imposed by processor and memory availability. First, we examine the maximal processor throughput as determined by the speedup characteristics of the workload and the average number of processors allocated per job. Next, we study the limitation on throughput due to the memory demands of jobs. In this case, the distribution of memory demands is significant, so we establish a bound that takes into account the minimum and maximum memory requirements per job.

Since the bounds are expressed in terms of the average processor allocation (to be precisely defined shortly), they can be combined to produce bounds of the form illustrated in Figure 1. As the average processor allocation increases, the processor bound decreases because the efficiency of jobs decreases. If all jobs require the same amount of memory, then the memory bound increases with the average processor allocation because jobs, on average, occupy memory for less time. For an arbitrary distribution of memory requirements, however, the throughput bound is constant over all average processor allocations because extreme cases exist where the average processor allocation does not matter. The maximum system throughput is obtainable where the processor and memory bounds intersect.

In our system model, there are P processors and M memory units. Each job, j_i, has a memory requirement m_i, which corresponds to its working set size. As such, the sum of the memory requirements of the jobs running at any point in time must not exceed M.

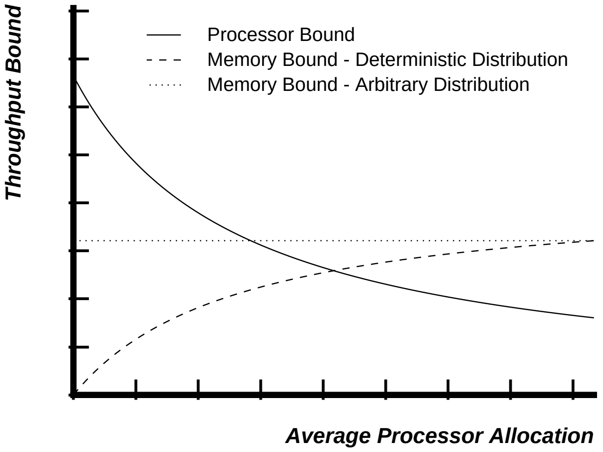


Figure 1: *Example of bounds for the case where the average memory demand is half of total memory. (The area beneath all three curves represents a system that is not saturated.)*

3.1 Processor Bound

Let there be a finite number of job classes distinguished by an execution-time function, $T_i(w,p)$, representing the execution time on p processors of a job whose computational demand is w (excluding overheads). Let the fraction of jobs generated for each class be f_i and let the statistical distribution of work for the jobs in class i be $b_i(w)$. Thus, the average execution time of the workload on p processors is

$$T(p) = \sum_i f_i \int b_i(w) T_i(w,p)\, dw$$

We define the *workload speedup* to be $S(p) = T(1)/T(p)$, which we assume to be a convex function (from above). Therefore, jobs can have different speedup characteristics, but we assume that the scheduler cannot distinguish jobs by their class, either upon arrival or during their execution[1].

A *processor allocation* assigns to each job some number of processors in the range $[0, P]$. Let the fraction of jobs to which the processor allocation assigns p_j processors be r_j. Since the scheduler cannot distinguish between job classes, the *average time-weighted processor allocation*, $\overline{p}$, is given by

$$\overline{p} = \frac{\sum_j p_j r_j T(p_j)}{\sum_j r_j T(p_j)} \tag{1}$$

(If the number of processors allocated to a job varies over time, then it can be treated as several separate job portions, each having a constant allocation.)

Proposition 1 *If $\overline{p}$ is the average time-weighted processor allocation, then the maximum throughput is bounded above due to processor availability by*

$$\frac{P}{\overline{p}T(\overline{p})} \tag{2}$$

Proof: First, observe that maximizing the system throughput corresponds to minimizing the average processor occupancy of the system, which is $O = \sum_j p_j r_j T(p_j)$. Our approach is to prove (by contradiction) that, if $\overline{p}$ is the average processor allocation, then O is minimized when all jobs are allocated $\overline{p}$ processors, which we assume to be an integral value.

Let there be an optimal processor allocation (with respect to processor occupancy) such that some jobs are allocated p_l and p_m processors, $p_l < \overline{p} < p_m$. Such an allocation pair must exist unless all jobs are allocated exactly $\overline{p}$ processors.

Choose a fraction s_l such that

$$\frac{p_l s_l r_l T(p_l) + p_m(1-s_l) r_m T(p_m)}{s_l r_l T(p_l) + (1-s_l) r_m T(p_m)} = \overline{p} \tag{3}$$

Such a value of s_l must exist in the interval $[0,1]$ by the intermediate value theorem of calculus. We now let a fraction s_l of jobs which were allocated p_l processors be allocated $\overline{p}$ processors, and the same for a fraction $s_m = (1-s_l)$ of jobs which were allocated p_m processors, again assuming that the scheduler cannot distinguish between job classes. By equations (1) and (3), the average time-weighted processor allocation remains unchanged from this reallocation (see **EQN A** in Figure 2).

The contribution to the average processor occupancy of the jobs selected from the original allocation is $O_1 = p_l s_l r_l T(p_l) + p_m s_m r_m T(p_m)$, which by (3) can be rewritten as $O_1 = \overline{p}[s_l r_l T(p_l) + s_m r_m T(p_m)]$. When allocated $\overline{p}$ processors, these jobs contribute $O_2 = \overline{p}(s_l r_l + s_m r_m)T(\overline{p})$ to the average processor occupancy. We show that the ratio O_1/O_2 is greater than one, which implies that the average processor occupancy under the new allocation is smaller than under the original one. This contradicts the presumption that the original processor allocation is optimal.

$$\begin{aligned} \frac{O_1}{O_2} &= \frac{s_l r_l T(p_l) + s_m r_m T(p_m)}{(s_l r_l + s_m r_m) T(\overline{p})} \\ &= \frac{1}{s_l r_l + s_m r_m} S(\overline{p}) \left[\frac{s_l r_l}{S(p_l)} + \frac{s_m r_m}{S(p_m)} \right] \end{aligned}$$

Since $S(p)$ is convex upward, its value at $\overline{p}$ is greater than the linear interpolation between the points $(p_l, S(p_l))$ and $(p_m, S(p_m))$ on the workload speedup curve (see **EQN B** in Figure 2). Given $T(p) = T(1)/S(p)$, we can rewrite equation (3) as:

$$\overline{p} = \frac{p_l s_l r_l S(p_m) + p_m s_m r_m S(p_l)}{s_l r_l S(p_m) + s_m r_m S(p_l)}$$

Substituting into **EQN B** and performing some algebraic manipulation, we obtain $O_1/O_2 > 1$ (see **EQN C** in Figure 2).

If every job is allocated exactly $\overline{p}$ processors, then the average processor occupancy is $\overline{p}T(\overline{p})$. If there are P processor-time units available per unit time, then the maximum achievable throughput is $P/(\overline{p}T(\overline{p}))$, assuming no loss occurs due to packing inefficiencies.□

This proof shows is that, if no information is known about the relative speedups of individual jobs and the workload speedup is convex upward, then an equi-partition policy for processors will maximize the sustainable throughput at heavy load. It has already been shown experimentally that dynamic equi-partition yields good

[1] For example, we do not allow a scheduler to infer the class of a job by the amount of time it has executed thus far.

$$\textbf{EQN A:} \quad \frac{\sum_i r_i T(p_i) p_i - [p_l s_l r_l T(p_l) + p_m s_m r_m T(p_m)] + [\overline{p} s_l r_l T(\overline{p}) + \overline{p} s_m r_m T(\overline{p})]}{\sum_i r_i T(p_i) - [s_l r_l T(p_l) + s_m r_m T(p_m)] + [s_l r_l T(\overline{p}) + s_m r_m T(\overline{p})]}$$

$$= \frac{\overline{p} \sum_i r_i T(p_i) - \overline{p}[s_l r_l T(p_l) + s_m r_m T(p_m)] + \overline{p}[s_l r_l T(\overline{p}) + s_m r_m T(\overline{p})]}{\sum_i r_i T(p_i) - [s_l r_l T(p_l) + s_m r_m T(p_m)] + [s_l r_l T(\overline{p}) + s_m r_m T(\overline{p})]} = \overline{p}$$

$$\textbf{EQN B:} \quad \frac{O_1}{O_2} > \frac{1}{s_l r_l + s_m r_m} \left[S(p_l) + (\overline{p} - p_l) \left(\frac{S(p_m) - S(p_l)}{p_m - p_l} \right) \right] \left[\frac{s_l r_l}{S(p_l)} + \frac{s_m r_m}{S(p_m)} \right]$$

$$\textbf{EQN C:} \quad \frac{O_1}{O_2} > \frac{1}{s_l r_l + s_m r_m} \left[S(p_l) + \left(\frac{(p_m - p_l) s_m r_m S(p_l)}{s_l r_l S(p_m) + s_m r_m S(p_l)} \right) \left(\frac{S(p_m) - S(p_l)}{p_m - p_l} \right) \right] \left[\frac{s_l r_l}{S(p_l)} + \frac{s_m r_m}{S(p_m)} \right]$$

$$= \frac{S(p_l)}{s_l r_l + s_m r_m} \left[1 + \frac{s_m r_m [S(p_m) - S(p_l)]}{s_l r_l S(p_m) + s_m r_m S(p_l)} \right] \left[\frac{s_l r_l S(p_m) + s_m r_m S(p_l)}{S(p_l) S(p_m)} \right]$$

$$= \frac{1}{s_l r_l + s_m r_m} \left[\frac{s_l r_l S(p_m) + s_m r_m S(p_l) + s_m r_m S(p_m) - s_m r_m S(p_l)}{S(p_m)} \right] = 1$$

Figure 2: *Large equations used in deriving bounds from Proposition 1.*

response times for a variety of workloads. Proposition 1 provides a theoretical basis for why equi-partition is also the best discipline to maximize throughput, enabling it to also yield good response times at high load.

3.2 Memory Bounds

Next, we address the corresponding bounds on throughput imposed by memory rather than by processors. While the processor bound depends only on the mean work demand, as captured in $T(p)$, the memory bound depends on the distribution of memory requirements. We start by addressing the simplest special case, where all jobs have the same memory requirement, $\hat{m}$.

Proposition 2 *If $\overline{p}$ is the average time-weighted processor allocation, and $\hat{m}$ is the amount of memory required by each job, then the maximum throughput is bounded above due to memory availability by*

$$\frac{M}{\hat{m} T(\overline{p})} \tag{4}$$

Proof Overview: The proof is similar to that for Proposition 1, except that we need to minimize the average memory occupancy instead of average processor occupancy. By equation (1), minimizing the average processor occupancy for a given average processor allocation, $\overline{p}$, is equivalent to minimizing the average execution time, which is equivalent to minimizing the average memory occupancy (since all jobs require the same amount of memory). In the last step of the proof, we have M memory-time units available per unit time and an average memory occupancy of $\hat{m} T(\overline{p})$, giving us our result. □

The realistic case, where jobs have different memory requirements, is more complex. Ignoring packing constraints, maximizing the throughput is equivalent to minimizing the average memory occupancy per job. In general, this is achieved by giving smaller jobs fewer processors and larger jobs more processors. Restricting memory requirements to the range m_L to m_U, where $0 \leq m_L \leq m_U \leq M$, permits us to represent a continuum of situations from that of arbitrary memory requirements ($m_L = 0$ and $m_U = M$) to equivalent memory requirements ($m_L = m_U$). Through experimentation, we found that, even for quite small values of m_L/M, the restriction that memory requirements lie in the interval $[m_L, m_U]$ meant that the bound on throughput was very close to that given by Proposition 2.

The next proposition shows how the throughput of a system S, in which jobs have arbitrary memory requirements in the interval $[m_L, m_U]$, cannot be any better than a related system S' in which all jobs require either m_L or m_U memory units. This proposition is useful because (1) it is relatively simple to compute the throughput bound for S', and (2) the bound applies to a wide range of distributions of memory demand (although it is tight only in the case where all memory requirements are either m_L or m_U.)

First, assume that we can classify jobs according to their memory demands, in addition to the previous classification based on execution-time function. Each memory class must have the same distribution of execution-time function classes because otherwise, the memory demand of a job might give information to the scheduler regarding the relative speedup of a job. However, each memory class c can account for a different fraction, g_c, of the job arrivals.

The *average work-weighted memory demand*, $\overline{m}$, is defined to be the amount of memory required by a job of class c, m_c, weighted by the fraction jobs in that class: $\overline{m} = \sum_c g_c m_c$. If all jobs require the same amount of memory, $\hat{m}$, then $\overline{m} = \hat{m}$.

Proposition 3 *Let S be a system having a finite set of job classes, where the memory demand corresponding to class c is m_c and in which the average work-weighted memory demand is $\overline{m}$. If the memory demands of the job classes are all in the interval $[m_L, m_U]$, then the throughput of this system for a particular average time-*

weighted processor allocation $\overline{p}$ is bounded above by that of another system S' having the same average work-weighted memory demand of $\overline{m}$, but only two job classes whose memory requirements are m_L and m_U, respectively. Preserving the value of $\overline{m}$, the fraction of work, g_L, corresponding to the m_L class in S' is $g_L = (m_U - \overline{m})/(m_U - m_L)$.

Proof: See Appendix.

For a system with exactly two memory-demand classes, the optimal processor allocation for each of these classes can be determined by minimizing the average memory occupancy[2]:

$$O_{mem}(p_L, p_U) = m_L g_L T(p_L) + m_U(1 - g_L)T(p_U)$$

subject to the constraint that

$$\frac{g_L p_L T(p_L) + (1 - g_L)p_U T(p_U)}{g_L T(p_L) + (1 - g_L)T(p_U)} = \overline{p}$$

This minimization problem has a real solution for p_L and p_U in the interval $[0, P]$. If we assume we have perfect packing of jobs into memory of size M, then our throughput bound is $M/O_{mem}(p_L^*, p_U^*)$ where p_L^* and p_U^* are the optimal processor allocations for the two memory classes, such that the above constraint involving $\overline{p}$ is still satisfied.

3.3 Application of the Bounds

To illustrate the bounds established in the previous sections, we use an execution-time function from an earlier study [ST93]:

$$T(w, p) = w(\alpha + \frac{\beta}{p}) \tag{5}$$

where $\alpha \approx 0.059$ and $\beta \approx 0.94$. These values of α and β were empirically obtained from the execution of a matrix-based application running on an iPSC/2 hypercube. With this execution-time function, all jobs have the same (convex upward) speedup curve. By varying the total number of processors, P, in the system, we can use the same execution-time function to study the throughput bounds given both a good and a poor speedup. Intuitively, this corresponds to the matrix-based application having good speedup when run on a small machine or for a large problem size, but poor speedup when run on a large machine or for a small problem size. The two values of P that we consider are $P = 128$, which leads to a maximum speedup of 15.0, and $P = 16$, which leads to a maximum speedup of 8.5.

In Figure 3, the throughput bounds are plotted as a function of the average time-weighted processor allocation for $P = 128$. In the first graph, the ratio of the average work-weighted memory demand to total memory, $\overline{m}/M$, has been set to 0.25 while in the second, it has been set to 0.50, with no statistical correlation between memory and processing demands.

The simple bounds on throughput obtained from Propositions 1 and 2 meet where $\overline{p}_o = \frac{\overline{m}}{M}P$, with a resulting throughput bound of $P/(\overline{p}_o T(\overline{p}_o))$. (In the case of the simple memory bound, we hereafter assume that $\hat{m} = \overline{m}$.) Also shown in the graph are the bounds obtained from Proposition 3, for values of $m_L/M = \{0, 0.1, 0.2\}$. For $m_L/M = 0.1$, the throughput bound for an arbitrary memory-demand distribution in the interval $[m_L, M]$ is already quite close to that given by Proposition 2; as m_L/M increases, the throughput bound further approaches that in which all jobs have the same memory demand. As a result, the throughput achievable for non-deterministic distributions of memory demands is only marginally higher than that achievable in the identical-memory-demand case unless m_L is very small relative to M.

These arbitrary-distribution bounds are for the most favourable distributions of memory demand within the constraint that all memory demands are in the interval $[m_L, M]$. For less ideal distributions, such as a uniform distribution, the throughput bounds are even closer to the bound of Proposition 2.

In the second graph in Figure 3, we can observe the effects of increasing $\overline{m}$. Since the slope of the processor bound decreases in absolute value with $\overline{p}$, the gains that can be achieved from sophisticated scheduling disciplines become smaller. In this particular situation, a relatively large range of average processor allocations, from about 50 to 65, will yield close to the same maximal throughput.

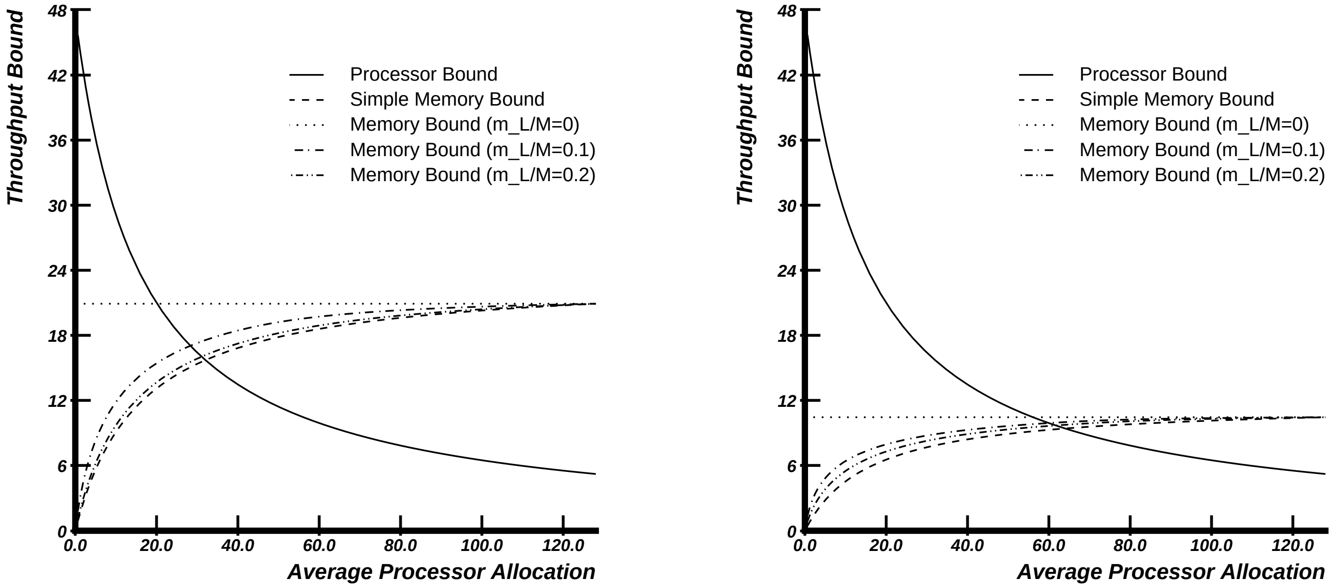
Consideration of Propositions 1 to 3 leads to three significant results:

1. *It is vital that memory demands be considered in processor-memory scheduling disciplines.* In processor-only scheduling disciplines, the average processor allocation must decrease as the load increases up to the point where, at the heaviest loads, each job receives exactly one processor. The graphs in Figure 3 show that, in processor-memory scheduling, one must also reduce the average processor allocation as the load increases, but only up the point where the processor and memory throughput bounds meet. If the average processor allocation decreases beyond this point, then memory becomes the primary constraint on throughput. As can be seen in the graphs, allocating too many processors on average is bad, but allocating too few is worse, as the bound on throughput drops rapidly for small average processor allocations.

2. *When no information is known about speedups of jobs, an equi-partition scheduling discipline is likely to be a good processor-memory discipline.* From the proofs of Proposition 1 and 2, the point at which the processor and simple memory bounds meet corresponds to an equi-partition scheduling discipline that attempts to always run $M/\overline{m}$ jobs at a time. We have found that determining a processor allocation that improves the throughput bound (as shown to be possible by Proposition 3) is computationally expensive and highly dependent on the speedup characteristics of the workload. An equi-partition strategy, on the other hand, is independent of the workload speedup function, leads to near-maximum throughput, and is already known empirically to yield good response times.

3. *Knowing the speedup characteristics of individual jobs is more important in processor-memory scheduling than in processor-only scheduling.* In processor-only scheduling, knowing speedup information can be helpful in minimizing the mean response time, but cannot improve the maximum throughput of the system. The reason is that maximizing efficiency is achieved by allocating each job one processor, regardless of the workload speedup. In processor-memory scheduling, however, it is necessary to try to maximize the

[2] It can be shown that the optimal processor allocation is obtained when jobs in each memory class are allocated the same number of processors as this minimizes the average memory occupancy of jobs in each class. The details can be found elsewhere [PS95a].



(a) $\overline{m}/M = 0.25$ **(b)** $\overline{m}/M = 0.50$

Figure 3: *Throughput bounds for* $P = 128$ *(poor speedup).*

processor efficiency for a given average processor allocation. If the scheduler knew the relative speedups of individual jobs, then it could better allocate processors (generally in a non-equipartition way) to increase the overall efficiency, and thus the throughput.

Figure 4 presents curves corresponding to those in Figure 3, for a smaller value of P. As previously mentioned, this corresponds to the case where jobs exhibit relatively good speedup. Because the processor throughput bound is much flatter than for $P = 128$, sophisticated scheduling disciplines that try to optimize for memory throughput will yield only a more limited increase in terms of total system throughput. Notice that it is even more important than before to allocate sufficient processors to jobs on average, as the throughput bound due to memory drops much more rapidly than the bound due to processors.

4 Scheduling Disciplines

Based on the results of the previous section, we have developed and evaluated a number of processor-memory scheduling disciplines. Previous experience has shown that the key to maintaining good mean response times is to ensure that the system does not approach the throughput bound. Otherwise, response times tend to increase quickly, regardless of the choice of discipline.

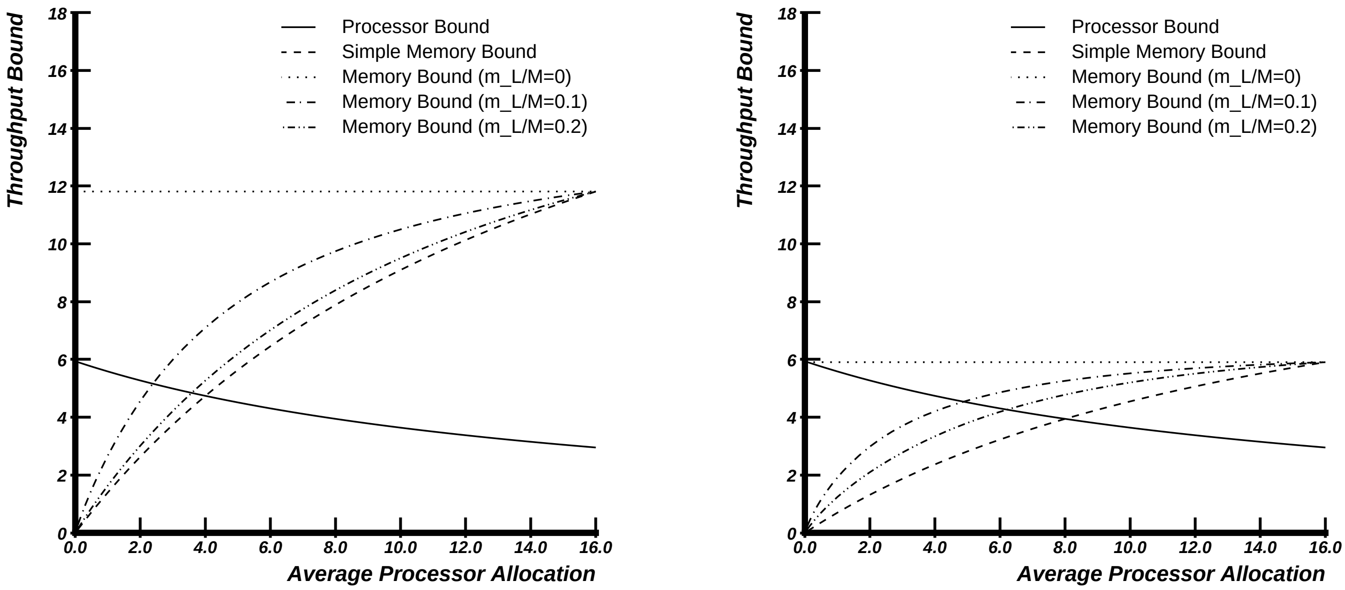
In this section, we present a simple processor-memory scheduling discipline that can be used for both shared-memory and distributed-memory architectures. We evaluate how well it performs as the load increases towards the throughput bounds. We then discuss how we can improve upon this discipline, in particular by improving the average memory utilization, which effectively increases the throughput bound (hence decreasing mean response times at high loads).

For the purposes of this study, we assume that the scheduler can use dynamic repartitioning with no overhead; repartitioning events occur only at job arrivals and job departures. Initially, we investigate the case where all jobs have the same speedup curve, and in Section 4.5, we comment on how different speedup curves influence the results.

4.1 Workload Model

As others have done in previous studies, we assume uniform distribution of memory demand with no correlation to service demand, focusing on values of $\overline{m}/M$ in the set $\{0.1, 0.25, 0.5\}$. The memory demands are generated from the uniform distribution over the interval $[0, 2\overline{m}/M]$. Based on informal evidence, we have found that workstations tend to have workloads in which a large fraction of jobs have small similarly-sized working sets. This is because an overwhelming majority of the jobs are invocations of small system commands. In a large-scale multiprocessor system, with batch submission of jobs, we have found that there is more variability in memory demand because there is a greater variety of applications. We conclude that a uniform distribution is no less realistic than other distributions that we might assume.

Interarrival times in our system are drawn from an exponential distribution, and processing demands from a hyper-exponential distribution of mean 1000 and coefficient of variation (CV) of 5. Such a service-time distribution is consistent with those that have been observed at high-performance computing centers [CMV94, FN95]. The execution-time function given by equation (5) is used again with $P = 128$.



(a) $\overline{m}/M = 0.25$

(b) $\overline{m}/M = 0.50$

Figure 4: *Throughput bounds for* $P = 16$ *(good speedup).*

4.2 Scheduling Disciplines

In order to minimize the mean response time, a scheduling discipline must strive to run the shortest jobs first. When the execution times of individual jobs are not known, the best discipline is one that runs jobs having the shortest expected remaining execution time first. With a hyper-exponential service-time distribution, this is equivalent to running the jobs having the least acquired service time first, a discipline commonly known as multi-level feedback or preemptive processor sharing [Cof67]. Our multiprocessor scheduling discipline relies on this approach to achieve good response times.

In a uniform memory access time (UMA) multiprocessor system, the processor allocation can be done independently of a job's memory requirement, as all jobs share the same global memory. In a non-uniform memory access time (NUMA) shared-memory system or in a message-passing system, on the other hand, a job's memory requirement places a lower bound on the number of processors that must be allocated to it. Our baseline memory-processor allocation (MPA) scheduling discipline deals with both cases as follows:

MPA At each job arrival or departure, or whenever the current quantum has expired, the scheduler re-assesses the jobs to run. It scans the jobs in order of increasing acquired service time, identifying the ones that can fit in memory (i.e., first fit). It then allocates each job its minimum processor requirement (which is one for UMA systems), and distributes the remaining processors among the jobs with the smallest allocations[3].

[3] In one of the disciplines that we investigated, processors were allocated to jobs in proportion to their memory requirements. Although such an approach seems to follow the principle of allocating smaller jobs fewer processors, we found the mean response times for this discipline to be consistently higher than those for either the UMA or NUMA variant of MPA.

4.3 Simulation Results

Our MPA scheduling discipline was simulated using the model and workload parameters described earlier (Poisson arrival process, hyper-exponential service times with mean 1000 and CV 5, and uniform memory requirement distribution). The load factor on the system is defined to be the ratio between the arrival rate and the throughput bound obtained from Propositions 1 and 2: $X(\overline{p},\overline{m}) = \min(\frac{P}{\overline{p}}, \frac{M}{\overline{m}})\frac{1}{T(\overline{p})}$. As discussed earlier, this throughput bound does not represent an absolute upper bound on throughput, but rather a good approximation to this value. As such, it is possible for a discipline to handle load factors greater than one, although this did not occur in any of our experiments.

In Figure 5, we plot the mean response times of the disciplines for different memory demands. As shown, smaller average memory demands lead to a higher overall relative throughput. The reason is that as the average memory demand decreases, memory is packed more efficiently by the first-fit algorithm. For $\overline{m} = 0.5$, our MPA discipline only manages to achieve about an 81.5% average memory usage, whereas for $\overline{m} = 0.1$, this value increases to about 94%.

A remarkable feature of Figure 5 is that the NUMA constraint ($p_i \geq m_i$) has only a very minor negative impact on performance. Nearly the same throughput is achieved with the NUMA constraint as without, and response times are only slightly higher. Although the constraint means that sometimes more processors are allocated to a job than necessary, this appears to be compensated for by the reduction in average memory occupancy. (Recall that, to minimize the average memory occupancy, one must in general give jobs with small memory requirements smaller processor allocations.)

It is possible to use the memory-packing efficiency to strengthen the throughput bound. If a scheduling discipline can use no more than a fraction s of the total system memory, on average, then the

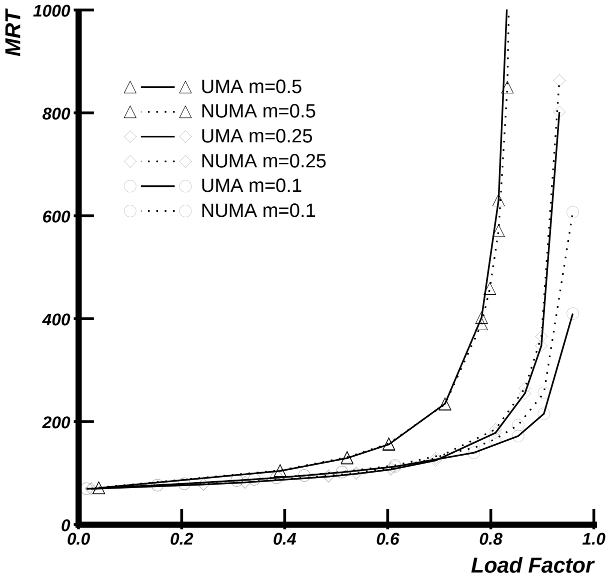


Figure 5: *Performance of basic UMA and NUMA scheduling disciplines for* $\overline{m} = \{0.1, 0.25, 0.5\}$.

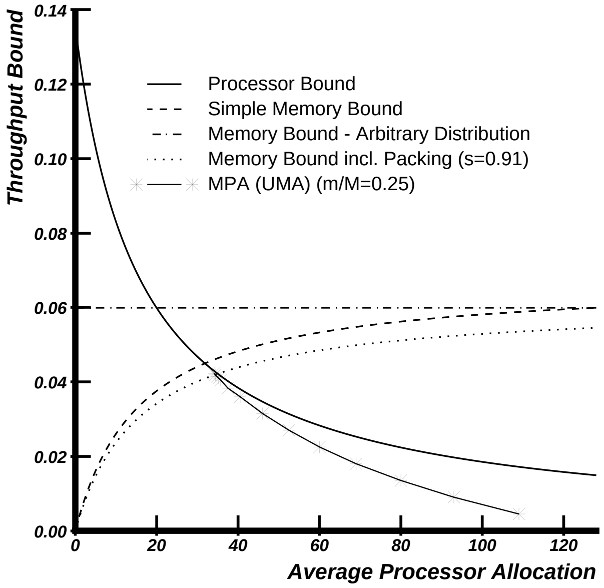


Figure 6: *Average processor allocation as a function of load for* $\overline{m} = 0.25$*, overlayed on the throughput bound graph.*

bound from Proposition 2 can be tightened to

$$\frac{sM}{\overline{m}T(\overline{p})} \tag{6}$$

If we set s to the memory usage observed in our experiments at saturation, then we find that the discipline comes very close to the modified bounds for all memory sizes.

This behaviour is illustrated in Figure 6. In this graph, the throughput of the system (which is equal to the arrival rate) is plotted against the average processor allocation exhibited by our discipline, for the case $\overline{m} = 0.25$. The throughput bounds are also plotted for comparison. As expected, the average processor allocation decreases as the load increases until it hits the processor bound. The loss due to packing inefficiency prevents the discipline from approaching the memory bound. However, using a value of $s = 0.91$ that we observed in our experiments, our scheduling discipline reaches both the processor and the modified memory bound simultaneously.

4.4 Improving the Disciplines

There are essentially two ways of improving the disciplines with respect to the memory throughput. Either the processor allocation can be optimized to reduce the average memory occupancy per job or the average memory utilization can be increased. The former is highly dependent on the workload speedup function involved, and requires considerable computation, so in this section, we concentrate on the latter approach.

We introduce two modifications to MPA:

MPA-Fill As in MPA, we scan the list of jobs and use first-fit to select the ones to run next. We then attempt to replace the last selected job (only) with another one not selected that achieves a higher memory utilization. The processors are allocated to the selected jobs in the same fashion as in MPA.

Since jobs with the least acquired service are still executed first, this modification does not noticeably degrade the average response times of jobs. But by increasing the average memory utilization, it results in a significant increase in the sustainable load for $\overline{m} = 0.5$, from a load factor of 0.85 to one of 0.89 in the UMA variant of MPA.

MPA-Pack Once again, we scan the list of jobs (call it R) and use first-fit to select the ones to run next (call this second list R_s). We then attempt to replace all jobs in R_s by other jobs in R, subject to a constraint that a job only be replaced by another if the latter has an acquired service time within a factor F ($1 \leq F$) of the former.

We use a branch-and-bound algorithm to traverse all possible combinations of jobs in R to find the one that leads to the best memory utilization while satisfying the constraint. More precisely, at each node of the decision tree, the partial candidate list of jobs R_c to use instead of R_s must (1) fit in available memory, and (2) have the property that the i^{th} job in R_c have an acquired service time no greater than F times that of the i^{th} job in R_s (assuming both lists are sorted by acquired service time)[4]. Note that if $F = 1$, then this discipline behaves identically to MPA, as a job can only be replaced by itself. The processors are then allocated to the newly-selected jobs in the same fashion as in MPA.

[4] If there are more jobs in R_c than in R_s, then only as many jobs in R_c as there are jobs in R_s need satisfy the criteria.

Discipline	*MemUsage*	*MaxLoad*
Basic	81.5%	0.85
MPA-Fill	86.5%	0.89
MPA-Pack	91.2%	0.93

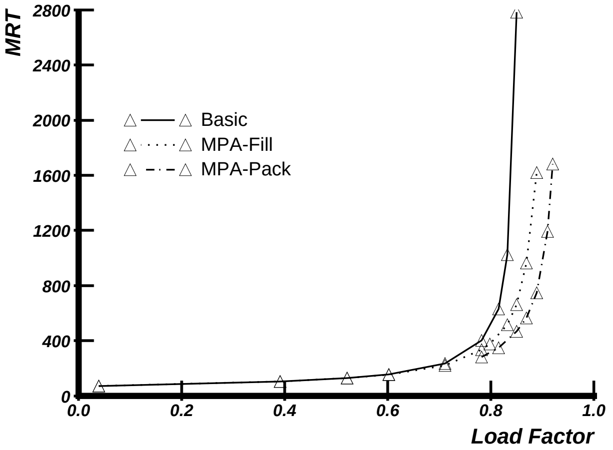


Figure 7: *Performance of UMA scheduling disciplines with packing improvements for* $\overline{m} = 0.5$.

Although more computationally expensive, the MPA-Pack discipline can achieve a much higher memory utilization, up to 91.2% instead of 81.5% for MPA in the case of $\overline{m}/M = 0.50$. The constraint ensures that jobs which have close to the least acquired service time are run first, thereby keeping the mean response times low. The difficulty with this discipline, however, is in finding a good value for F, as the optimal value depends on the load on the system, and can be as low as 1 to as high as 40.

The performance of the MPA-Fill and MPA-Pack disciplines, assuming an UMA environment, is shown for $\overline{m}/M = 0.5$ in Figure 7. In the accompanying table, the maximum observed memory utilization is given for each of the disciplines, along with the predicted maximum sustainable load. In each case, the observed maximum throughput matches perfectly with the predicted value. MPA-Fill yields a significant improvement in achieved throughput, but MPA-Pack achieves a much better mean response time at high loads (e.g., about 54% improvement over MPA-Fill at a load factor of 89%).

4.5 Different Speedup Curves

In practice, no system will have a workload in which all jobs have the same speedup curve. In fact, for any given job and problem size, speedup will decline after a certain level of parallelism due to contention or overhead.

To investigate the effect of distinct speedup curves, we carried out simulation experiments to determine average response time as a function of load factor for MPA for the same average memory requirements as before. For this purpose, we used three empirically-derived execution-time functions whose efficiency depend on both the number of processors and on the amount of work,

Job Class	ϕ	α	β
1	1.01	3.4	0.0042
2	1.10	8.0	0.66
3	1.16	30.	0.050

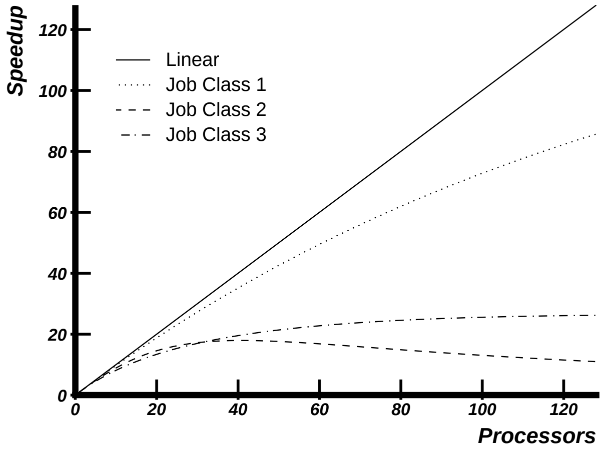


Figure 8: *Job-class speedup functions used for experiments where jobs have varying speedup characteristics.*

w, involved [Sev94, Wu93]:

$$T(w,p) = \frac{\phi w}{p} + \alpha + \beta p$$

In this function, ϕ represents imbalance in the distribution of work, α represents the constant per-processor overhead, and β represents contention delay, which is assumed to grow linearly with the number of processors. The speedup curves averaged over all jobs sizes for each of the three execution-time functions, which differ in ϕ, α, and β, are shown in Figure 8. Although only one curve exhibits slowdown on average, all jobs will exhibit slowdown given a sufficiently small value of w.

In Figure 9, we show the performance of the both the UMA and NUMA variants of the MPA scheduling disciplines, again for $\overline{m} = \{0.1, 0.25, 0.5\}$. There are two important differences from the case where all jobs shared a common speedup function. First, in the case of $\overline{m} = 0.25$, the system saturates much more quickly than before. The reason is that imbalances in processor allocation have a much greater effect, since it is possible for a single poor-speedup job to be allocated all the processors in the system if it has a high memory requirement. This is the effect that was shown to occur in the proof to Proposition 1. Second, the NUMA constraint seems to impede performance more than before, because the minimum processor requirements lead to even more imbalance, producing an even greater loss in processor efficiency.

5 Conclusion

We have investigated the coordinated allocation of processors and memory in the context of multiprocessor scheduling. By first establishing some bounds on the achievable system throughput, we were

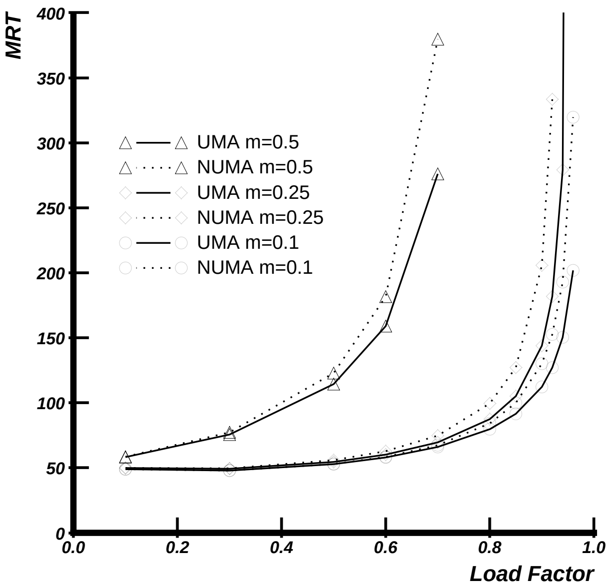


Figure 9: *Performance of basic MPA with and without the NUMA constraint for* $\overline{m} = \{0.1, 0.25, 0.5\}$ *when jobs have varying speedup characteristics.*

able to gain insight into how to design scheduling disciplines for this problem. Our significant observations include:

- It is very important that memory be considered in scheduling decisions. Allocating too few processors to jobs on average will lead to memory saturation, significantly degrading system performance.
- If the workload speedup function is convex upward, but no information is known about the relative speedup characteristics of individual jobs, then an equi-partition strategy favouring jobs with the least acquired service will yield good response times and achieve a throughput close to the maximum. If memory can be well utilized, then the limit on maximum throughput will be
$$\frac{M}{\overline{m}T(\frac{\overline{m}}{M}P)}$$
- Knowing speedup information is more important in processor-memory scheduling than in processor-only scheduling for increasing the maximum attainable system throughput.

One avenue of future research involves the overheads associated with scheduling large-memory jobs. These include swapping jobs in and out of memory and changing a job's processor allocation. Since these overheads are likely to be large, the system should strive to execute short jobs quickly without interruption, and to only begin changing the processor allocation of a job after it is expected to continue running for much longer.

Another topic of importance is the extent to which approximate knowledge of service times and relative speedup properties can be used. Service-time knowledge can direct the scheduler to run the shortest jobs first, thereby minimizing mean response times. Speedup knowledge can help the scheduler obtain higher efficiency, thereby improving the limit on system throughput and reducing mean response times at high loads.

References

[AKK+95] Gail Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith. Scheduling on the Tera MTA. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 19–44. Springer-Verlag, 1995.

[BHMW94] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. In *Proceedings Supercomputing '94*, November 1994.

[CMV94] Su-Hui Chiang, Rajesh K. Mansharamani, and Mary K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 33–44, 1994.

[Cof67] Edward G. Coffman. Studying multiprogrammed systems. *Datamation*, pages 47–54, June 1967.

[EZL89] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[FN95] Dror G. Feitelson and Bill Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 337–360. Springer-Verlag, 1995.

[FR92] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[GST91] Dipak Ghosal, Guiseppe Serazzi, and Satish K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[GTU91] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, 1991.

[MZ94] Cathy McCann and John Zahorjan. Processor allocation policies for message-passing parallel computers.

In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 19–32, 1994.

[MZ95] Cathy McCann and John Zahorjan. Scheduling memory constrained jobs on distributed memory parallel computers. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modelling of Computer Systems*, pages 208–219, 1995.

[PS95a] Eric W. Parsons and Kenneth C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. Technical Report 336, Computer Systems Research Institute, University of Toronto, 1995.

[PS95b] Eric W. Parsons and Kenneth C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 127–145. Springer-Verlag, 1995.

[PSN94] 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 the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 5–18, 1994.

[Set95] Sanjeev Setia. The interaction between memory allocations and adaptive partitioning in message-passing multiprocessors. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 146–164. Springer-Verlag, 1995.

[Sev89] Kenneth C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of the 1988 ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems*, pages 171–180, May 1989.

[Sev94] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19:107–140, 1994.

[SN93] Xian-He Sun and L. Ni. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 19(1):27–37, Sept 1993.

[ST93] Sanjeev Setia and Satish Tripathi. A comparative analysis of static processor partitioning policies for parallel computers. In *Proceedings of the International Workshop on Modeling and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 283–286, January 1993.

[TG89] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, 1989.

[Wu93] Chee-Shong Wu. Processor scheduling in multiprogrammed shared memory NUMA multiprocessors. Technical Report 341, Computer Systems Research Institute, University of Toronto, 1993.

Appendix

Proof of Proposition 3: Let there be an optimal processor allocation with respect to memory occupancy for jobs in each memory class. Consider each job class c in turn. We construct system S' by transforming the jobs of class c, which have memory demand m_c, into jobs of either size m_L or of size m_U, without changing the values of $\overline{m}$ and $\overline{p}$.

Let a fraction $s_{L,c} = \frac{m_U - m_c}{m_U - m_L}$ of the jobs in memory class c in S be of size m_L in S' and the remaining fraction, $s_{U,c} = 1 - s_{L,c}$, be of size m_U in S'. Let these jobs execute with the same number of processors in S', thus leaving the average time-weighted processor allocation unchanged. The average work-weighted memory demand $\overline{m}$ of the jobs in S' also remains unchanged because the contribution of work-memory demand is the same as before, as is the total amount of work in the system. Since the fraction of jobs generated for memory class c is g_c, the fraction of work-memory demand in the original class is $m_c g_c$ and in the transformed class it is $s_{L,c} m_L g_c + s_{U,c} m_U g_c$, which are equal due to the definition of $s_{L,c}$. The average memory occupancy also remains the same by a similar argument as for the average work-memory demand.

Thus, S' has the same average time-weighted processor allocation and the same average work-weighted memory demand as S, but has an optimal processor allocation that is guaranteed to be no worse than that of S. From the transformation, the total fraction g_L of work associated with the m_L class is

$$g_L = \sum_c g_c \frac{m_U - m_c}{m_U - m_L} = \frac{m_U - \overline{m}}{m_U - m_L}$$

$\square$