Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors

Songnian Zhou and Timothy Brecht
Computer Systems Research Institute
University of Toronto, Toronto, Ontario, M5S 1A4 Canada
zhou/brecht@sys.toronto.edu

Abstract

Large-scale Non-Uniform Memory Access (NUMA) multiprocessors are gaining increased attention due to their potential for achieving high performance through the replication of relatively simple components. Because of the complexity of such systems, scheduling algorithms for parallel applications are crucial in realizing the performance potential of these systems. In particular, scheduling methods must consider the scale of the system, with the increased likelihood of creating bottlenecks, along with the NUMA characteristics of the system, and the benefits to be gained by placing threads close to their code and data.

We propose a class of scheduling algorithms based on *processor pools*. A processor pool is a software construct for organizing and managing a large number of processors by dividing them into groups called pools. The parallel threads of a job are run in a single processor pool, unless there are performance advantages for a job to span multiple pools. Several jobs may share one pool. Our simulation experiments show that processor pool-based scheduling may effectively reduce the average job response time. The performance improvements attained by using processor pools increase with the average parallelism of the jobs, the load level of the system, the differentials in memory access costs, and the likelihood of having system bottlenecks. As the system size increases, while maintaining the workload composition and intensity, we observed that processor pools can be used to provide significant performance improvements. We therefore conclude that processor pool-based scheduling may be an effective and efficient technique for scalable systems.

Appears in: Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May (1991), pp. 133-142

1 Introduction

In the quest for greater processing power to support increasingly demanding applications, parallel computer systems have the potential of meeting computing needs by combining many relatively inexpensive, easy-to-obtain computing components to form a system. An important class of parallel systems is shared-memory multiprocessors, in which all processors have hardware access to a single physical address space, although the memory units used to implement the address space may be distributed throughout the system. The main advantage of a shared-memory multiprocessor, as opposed to a non-shared-memory multiprocessor such as a hypercube, is the simple programming model it presents to the application programmer. Executable programs and data stored in memory are directly shared among the processors, and communication and synchronization among the processing activities can be realized through the shared memory.

A number of small-scale shared-memory multiprocessors of up to 30 processors have been successfully built, such as those developed by Alliant, DEC, Encore, Sequent and SGI. They are typically based on a shared bus connecting several processors with local caches to global memory. Such systems have provided us with increased processing power and valuable experience in the use of parallel systems. However, their performance potential is rather limited. Large-scale multiprocessors with over 100 processors offer much greater potential in their capacity for supporting parallel applications, for two main reasons: First, applications that possess high degrees of parallelism may have their performance potential realized in such systems. Second, multiple parallel applications can be efficiently executed in such systems concurrently. The second reason may be as important as the first since not all (or even many) applications can use 100 processors effectively. A large-scale multiprocessor represents an important computing facility, as well as a substantial investment. It is therefore desirable to maintain high system utilization. As in the uniprocessor case, multiprogramming is an important method for achieving high system utilization in multiprocessors. In addition to sharing the CPU time of the individual processors among multiple concurrent jobs (time multiplexing), with multiprocessors we also have the opportunity to divide the processors among several applications (space multiplexing).

Very few general purpose, large-scale shared-memory multiprocessors have been built to date (example systems include BBN's Butterfly [1], IBM's RP3 [18], Illinois' Cedar [10], Myrias' SPS-2 [2], and Toronto's Hector [23]). The bus-based, global-memory architecture of small systems does not scale because the shared bus quickly becomes a system bottleneck as more and more processors are added. Although the bus can be replaced by a large switch, the cost of the switch grows rapidly with system size, and such a Uniform Memory Access (UMA) architecture is likely to make all the memory accesses uniformly slow. From a performance point of view, Non-Uniform Memory Access (NUMA), by which the physical memory of the system is distributed and placed closer to the processors, is inevitable for large-scale shared-memory multiprocessors.

Scheduling parallel applications on shared-memory multiprocessors presents problems not encountered in traditional sequential systems. Although a substantial amount of research has been done on scheduling small UMA multiprocessors, very little research has been done on large-scale, NUMA systems (see Section 5 for a brief survey). Due to the substantial differences in scale and architecture, desirable scheduling algorithms for large multiprocessors may be quite different from those for small systems. In particular, the scheduling algorithms should explore the increased opportunity of sharing the system among several applications, and because of the NUMA characteristics of such systems, should take advantage of locality in order to realize the performance potential of such systems.

In this paper, we study the desirable characteristics of scheduling algorithms for multiprogrammed use of general purpose, large-scale NUMA multiprocessors. The study of large scale shared-memory multiprocessors has received little attention in the literature. In order to advance the understanding of the problems in this area we first identify the non-goals of this paper:

1. We are not trying to design and evaluate optimal scheduling algorithms for large systems. Such algorithms are likely to be highly dependent on the workload, for which little information is currently available.
2. We are also not trying to study multiprocessor scheduling in general; instead, we choose to focus on the unique aspects of large NUMA systems.

Our goal is to identify and evaluate the essential properties of scheduling algorithms that take into consideration the NUMA architecture of large multiprocessors, and to assess the potential performance gains of such considerations. Previous results obtained for small systems are used wherever applicable. As an exploratory study, we design a range of simple scheduling algorithms that explicitly consider, to varying degrees, the scale and NUMA characteristics of the system. We then simulate the algorithms under parallel workloads to assess the desirability of their various features. The performance index we use in evaluating various algorithms is the mean response time (Mean RT) of all jobs executed by the system.

A large shared-memory system is prone to many hardware and software bottlenecks; as the number of processors increases, shared resources, such as the interconnection network, the memory, the global run queue, and the software servers, may all become limiting factors of system performance. We believe that effective scheduling algorithms, like scalable architectures, are a key to alleviating bottlenecks in a large-scale system. One known architectural technique for large systems is to organize the processors and memory units into clusters.[1] If memory sharing and communication occur mainly within the clusters, then the interconnection network among the clusters is less likely to become a bottleneck. Analogously, we hypothesize that *a key feature of scheduling multiprogrammed parallel applications in large NUMA systems is the use of* **processor pools**. In contrast to hardware clusters of processors, processor pools are used as an operating system construct for scheduling applications, and are applicable even in systems with no apparent processor clusters. The processors in a system are partitioned into a number of pools, in which threads from several jobs are scheduled to run.

[1] A large number of such systems have been designed, a few are being built. Examples include Michigan's HM^2P, MIT's Alewife, Wisconsin's Multicube, Illinois' Cedar, Stanford's VMP-MC, and Toronto's Hector [23].

The parallel threads of a job are run in a single processor pool, unless there are performance advantages for a job to span multiple pools.

Although the concept of a processor pool is based on the large scale and NUMA characteristics of a system, and is not tied to any particular architecture, in actually implementing processor pools on a specific system, its NUMA characteristics should be fully exploited. There are often natural clusters of processors that are good candidates for pools. For example, in the Wisconsin Multicube architecture [11], processors are interconnected by vertical and horizontal buses to form a grid. Cache consistency is achieved by snoopy caches sharing the buses. The rows or columns of processors are therefore natural candidates for pools. As another example, consider the Toronto Hector system [23], which is organized as multiple levels of parallel rings connecting stations of a small number of processors sharing a bus. The stations or the local rings are therefore good candidates for processor pools.

Besides easing system bottlenecks, processor pools, coupled with possible hardware processor clusters, have two more potential advantages: First, parallel processing within pools of processors is likely to be more efficient when processor clusters are considered in forming pools in particular machines. For example, while a shared bus is undesirable for a large system, it may be used within processor clusters on which pools are based. Cache consistency may thus be achieved efficiently using snoopy caches. Second, accessing a memory location associated with the local pool is often faster than accessing a memory location outside the pool. If the memory management component of the operating system cooperates with the scheduler to increase the percentage of memory accesses that are local to the pool, then the processors' utilization will improve, and the applications will run faster. On the other hand, partitioning the system into processor pools introduces the problem of load balancing, which is usually avoided in small systems by simply having a global run queue. Therefore, the choice between system-wide scheduling and pool-based scheduling may be viewed as a tradeoff between localization of parallel jobs and load balancing.

Our idea of processor pools can be motivated by Figure 1, in which the average response times of a simulated parallel workload, running on hypothetical UMA systems with various numbers of processors, are shown. The workload and system model are described in detail in Section 2. The jobs have various degrees of parallelism, with averages of 10 and 20 shown in the figure (an average parallelism of 10 means that on average jobs create 10 parallel threads). The performance results are derived with an average processor utilization of 60% under the unrealistic assumption that there is no contention for the interconnection network or memory. The key observation in Figure 1 is that, for a given level of job parallelism, the performance improves with the size of the system, as the probability of finding idle processors in the whole system increases; however, the law of diminishing returns is observed in large systems. For example, a system with 50 processors yields performance quite close to that of 120 processors. When the various overheads associated with large NUMA systems are taken into account, the performance of larger systems is likely to degrade, suggesting that it may be advantageous to schedule jobs in pools of processors to reduce bottlenecks and improve efficiency.

We study the merits of employing processor pools using simulation. The factors that may affect the performance of processor pool-based scheduling, such as the average parallelism of

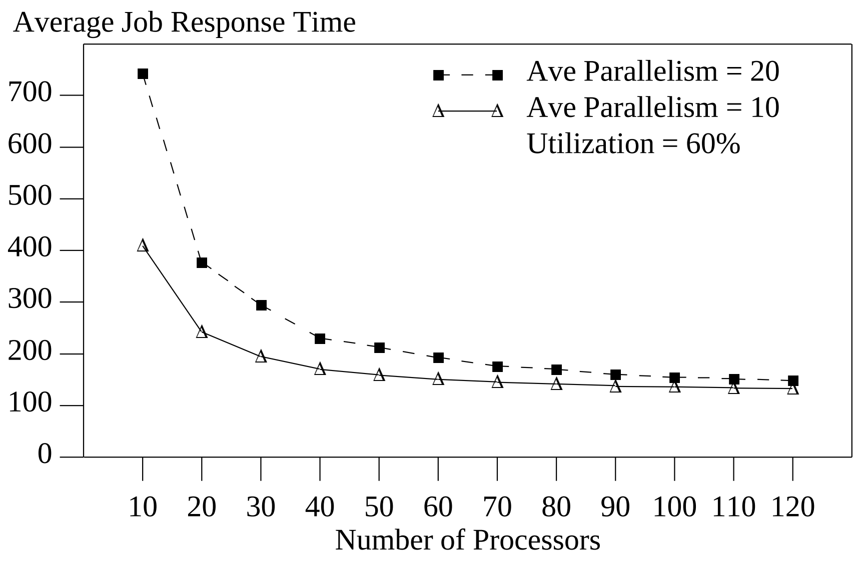

Figure 1: Performance of hypothetical UMA systems.

the jobs, the load level of the system, the scale of the system, the differentials in memory access costs, and the likelihood of having system bottlenecks are identified and studied. The rest of this paper is organized as follows. In Section 2, we present our models for the system and the workload. The algorithms we study are described in Section 3, along with our overhead assumptions. We present a sequence of simulation experiments in Section 4 in order to assess the desirability of the various features of the proposed scheduling algorithms. Related work is briefly discussed in Section 5, followed by concluding remarks in Section 6.

2 System and Workload Models

Our model of the system is simple and general. The system consists of P homogeneous processors, where P is in the range of several tens to several hundreds. All of the memory forms a physical address space accessible in hardware from any of the processors. Memory is distributed such that some memory locations are accessed more quickly than others. We model the differentials in memory access costs at a high level, without modeling the memory architecture of a particular system (see Section 3.3).

The CPU service demand and response time of a job, as well as some of the overhead costs considered in this paper, are measured in an abstract *time unit*, which, with current processor and network speeds, may be considered to be in the range of 20-100 milliseconds. A job consists of three consecutive phases, as shown in Figure 2. The INIT and FINAL phases have fixed CPU service demand of 10 time units. The PARA phase has a *desired parallelism, d* (i.e., given an unlimited number of idle processors, d threads would be created). Each of the threads has a CPU service demand (in the absence of any overhead) following a uniform

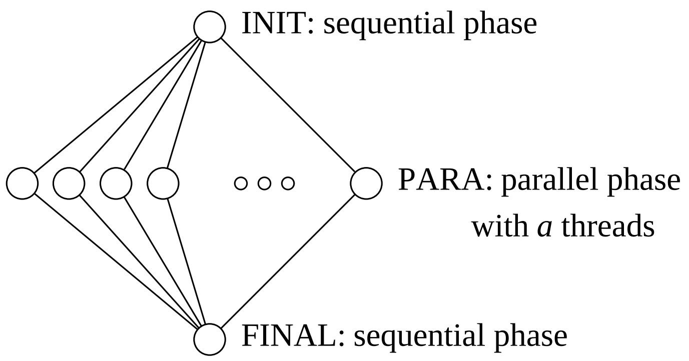

Figure 2: Structure of a parallel job.

distribution U(90,110). A scheduling algorithm may decide to create a number of threads, a, less than or equal to d.
The corresponding parallelism is called the *actual parallelism*. To make the workload comparable across all scheduling algorithms, however, the total CPU demand of the PARA phase is considered a characteristic of the job, and is independent of the value of a. Moreover, the CPU demand of each of the a (actual) threads has a uniform distribution of $\mathrm{U}(100d/a - 10, 100d/a + 10)$. Such a job model assumes some form of user-level scheduling that interacts with the system scheduler at the start of the PARA phase to decide how many threads to create, and distributes the work roughly evenly among the parallel threads. Typically, the application programmer specifies the logical tasks in the job that can be executed in parallel, and a library package keeps track of them and assigns them to the system threads for execution. If the granularities of the tasks are relatively small, then the unchanged spread in the thread execution time distribution is reasonable. Task switching does not involve the operating system, and incurs much lower overhead than for switching system threads. Recent work on UMA scheduling indicates that such application participation in scheduling is both feasible and desirable [22, 12, 21].

The fork-join job model described above is quite typical of many parallel applications. However, it does not model any synchronization and task dependency relationships among the threads except for the barrier synchronization at the end of the PARA phase. While we could generalize the model to include multiple phases, or more complex dependency structures, we do not expect such changes to affect our results concerning pool-based scheduling significantly (see Section 4.6).

The job model described above is of the correlated type in that the CPU demand of a job is positively correlated with its degree of parallelism [14]. In addition to this workload model, we also briefly consider a comparable workload in which the total service demand of a job is not correlated with the number of desired threads (see Section 4.6).

Our workload model is based on the assumption that there exists a broad mixture of parallel jobs, as well as some sequential jobs. The desired parallelism of a job, d, follows a bounded-geometric distribution with mean D and upper bound P. A geometric distribution with a mean somewhat greater than D is used to generate a value for d; if a value greater than P is obtained, another random value is generated until the value obtained is less than or equal to P. The resulting distribution has a true mean of D. Thus, the parallelism of the jobs is characterized by a single parameter, the *average desired parallelism, D*.[2] No more than P threads will be requested by a job, since, in the presence of user-level scheduling, there is no point in requesting more threads than the number of processors in the system. The bounded-geometric distribution of parallelism results in a wide range of parallel jobs. There is only a small percentage of sequential or very small parallel jobs (e.g., for a system with $P = 60$ processors and a workload with $D = 20$, only 15.7% of the jobs have parallelism ≤ 4), which seems to be desirable, as we are primarily concerned with parallel job scheduling. A greater number of sequential and small parallel jobs will improve application performance, as indicated by our results in Section 4.1. Similarly, the bounded-geometric distribution generates a small percentage of jobs with parallelism much larger than the mean (e.g., for $P = 60$ and $D = 20$, 6% of the jobs have parallelism between 50 and 60). Such jobs may do well in a system with light loads; however, with multiprogrammed use of the system and the moderate to heavy loads we are interested in, the average job response time would rise sharply if individual jobs were allowed to use most of the processors, while other jobs used the remaining processors. Therefore, the relatively small number of very large parallel jobs generated with the bounded-geometric distribution seems to be realistic. The actual parallelism of a job, a, is determined by the particular scheduling algorithm, to be discussed in the next section.

The system workload consists of a Poisson stream of jobs arrivals. The *load level*, L, of the system is defined as the average utilization of the processors in the absence of any overhead. Thus, for a 60-processor system serving jobs with an average desired parallelism of 20, a default load level of 60% corresponds to a job arrival rate of $60 \times 60\% \ / \ (10 + 100 \times 20 + 10) = 0.0178$ jobs/unit time. When the various overheads to be discussed in Section 3.3 are considered, the actual utilization of the processors, U, will be higher.

3 Scheduling Algorithms and Overheads

3.1 General Description

The system is partitioned into a number of processor pools for the purpose of job scheduling. Each processor in the system belongs to one and only one pool. In this paper, we assume that all of the pools have an equal number of processors, p. Each pool has a single run queue shared by all the processors in the pool. When a job arrives, a master thread is created and assigned to a pool with the minimum load, and the INIT phase is started. The current load index for a pool is its number of *active* threads (if there are large variations in thread service demands other load indices could be used). An active thread is a thread either running or

[2] Note that our notion of average parallelism is different from that proposed by Eager, Lazowska and Zahorjan [8]; theirs represents the average parallelism that a job exhibits during its execution, whereas ours refers to the average of the desired parallelism in the PARA phase of all the jobs in a workload.

waiting in a ready queue.[3] The processor pool used for the master thread is called the *home pool* for the job. When the INIT phase completes, a threads are created for the PARA phase, and are assigned to one or more pools. The particular pool(s) used for these threads and the value of a are determined by the algorithms to be described in Section 3.2. When all of the parallel threads of a job complete, the corresponding master thread is placed back into the ready queue of the home pool for the FINAL execution phase.

Threads executing in a pool are scheduled using a round-robin policy. All processors use the same time slice, T. When a time slice expires, the local pool's run queue is inspected. If it is empty, then the thread continues without a context switch, thus avoiding the associated overhead. Otherwise, the running thread is added to the tail of the queue, and the first thread in the queue is executed.

3.2 Alternatives for the Algorithms

There are two areas in which the algorithms we consider may differ:

Assigning threads to pools. In this paper, we will only consider the static assignment of threads to pools (i.e., once a thread is assigned to a pool, it executes within that pool until completion). The parallel threads of a job are considered for placement as a group. We always consider the job's home pool first, and then other candidate pools in increasing order of their load, until either all of the d threads have been assigned to idle processors (indirectly through their run queues), or until we have used as many pools as a particular algorithm will allow. For simplicity, we assume that all pools are equal, so the only factor in choosing pools is their current load. This may not be true in some architectures, where more intelligent decisions may be made to further improve locality (among the pools used). The processor pools that a job can use are called its *candidate pools*. There is a range of possibilities for the number of candidate pools a job considers:

1. *No-spanning:* All of the parallel threads of the job are assigned to the job's home pool. This method ensures maximum localization of the application, but may limit the number of processors an application can use, and hence its speedup.
2. *Unlimited-spanning:* The parallel threads can be assigned to any pool.
3. *Limited-spanning:* This is an intermediate choice between the above two extremes. Let

$$x = round(F \times d \;/\; p)$$

where F is a control parameter for the possible number of pools to span. The number of candidate pools is then determined by the following function:

$$f(d) = \begin{cases} 1 & \text{if } x < 1 \\ x & \text{if } 1 \leq x \leq P/p \\ P/p & \text{if } x > P/p \end{cases}$$

[3] In particular, After the INIT phase, the master thread is put to sleep until all the parallel threads have completed, so it is not counted in the pool load during the PARA phase.

The above functions are used to restrict the number of pools that a job can use based on its desired parallelism. For example, suppose $d = 20$, $p = 10$, and $F = 0.8$, then the job can only consider $x = 2$ pools in a 60-processor system. By varying the value of F, we can control the degree of pool spanning and study its desirability. When $F = 0$, $f(d) = 1$ and we get the no-spanning case. When F is very large, $f(d) = P/p$ and we get the unlimited-spanning case.

The above choices represent a range of tradeoffs between the localization of a job within one or few pools, thus achieving efficient execution, and load balancing among the pools.

Limiting the degree of parallelism. If there are few idle processors in the candidate pools when a job starts its PARA phase, it may be desirable to limit its number of parallel threads, through user-level scheduling, so that it does not claim too large a share of the processing resources. By limiting the number of PARA threads, contention for the processors may also be reduced, so less context switching overhead is incurred. Again, there is a range of choices:

1. *Unrestricted-parallelism:* $a = d$.
2. *Restricted-parallelism:* $a = min(d, max(p_{idle}, G \times d, 1))$.
   where p_{idle} is the total number of idle processors in the candidate pools, and the parameter G ($0 \leq G \leq 1$) controls the minimum level of actual parallelism. If the number of idle processors in the candidate pools exceeds the desirable parallelism, then we simply create d parallel threads; otherwise, we ensure that at least one or $G \times d$ threads are created, whichever is greater. Notice that this minimum actual parallelism is directly proportional to the desired parallelism.

Once the actual parallelism of a job has been determined, the total CPU demand of the PARA phase is roughly evenly distributed among the a threads, as described at the beginning of Section 2. If $a > p_{idle}$ then the threads without idle processors are distributed among the candidate pools to balance their resulting load.

Continuing the example we used for determining the number of pools to span, suppose $F = 0.5$, and there are only 2 and 6 idle processors in the two 10-processor candidate pools, respectively, then $20 \times 0.5 = 10$ threads will be created, with 3 and 7 of them allocated to each of the two pools. As a result the number of threads in each of the two pools is 11 and the load is balanced.

With restricted-parallelism, if $G = 1$, then we get unrestricted-parallelism as an extreme case. If $G = 0$, then no more threads will be assigned than the total number of idle processors in the candidate pools, with a minimum of one. With such a policy, contention for the processors among the threads will be minimized; however, a large parallel job may be squeezed into a small number of parallel threads, drastically inflating its response time, as well as the average job response time. This problem is particularly severe in our simplified version of user-level scheduling, since the number of parallel threads a job may have cannot be dynamically adjusted. As a result of limiting the jobs' parallelism ($a \leq d$), the *average actual parallelism*, A, is usually somewhat less than the average desired parallelism, D.

Although pool-based scheduling weakens system-wide load balancing, a certain degree of load balancing is achieved in the above algorithms in two ways: First, the pools with the lowest loads are selected for the jobs. Second, the loads of the candidate pools are balanced at parallel thread placement time.

3.3 Overhead Assumptions

We consider four types of overhead in our simulation experiments.

In-pool overhead. As pointed out in the introduction, the execution of a parallel job is likely to be more efficient if its threads run on processors close to their shared data and to each other. As the number of processors in a pool increases, the threads of a parallel job spread over a wider range of processors, incurring greater overhead. We model this by the *in-pool overhead*, which represents the penalty for ignoring the NUMA characteristics of the system. Four sources of in-pool overhead may be identified as the pool size increases:

1. The percentage of remote memory or shared cache accesses may increase, causing longer delays.
2. As a result of increased remote memory traffic, the shared interconnection network and memory or shared caches may become congested, causing additional delays.
3. The overhead in keeping the processors' caches consistent may increase, as software or directory-based methods may have to be employed [5]. Alternatively, if no cache consistency is maintained, cache misses increase, so local and remote memory traffic increases.
4. Software servers shared by the threads (such as a thread scheduler and memory manager) may become congested.

Clearly, the impact of some of these overheads depends on the particular architecture concerned. One possible classification of large-scale shared memory multiprocessor architectures is the following:

- *hierarchy*: processors are connected by multiple levels of buses and/or parallel rings or switching networks. Examples include Hector [23] and Cedar [10].
- *hierarchy of shared caches*: Another type of hierarchical architecture, though not strictly NUMA, consists of processors connected by multiple levels of buses with shared caches, and global memory at the bottom level. An example is VMP-MC [6].
- *multi-grid of buses*: processors are connected by buses running in two or more dimensions which are used to maintain cache consistency. An example is Multicube [11].
- *general network*: Each processor has a limited number of links through which processors are connected, either directly or indirectly through routing switches, to form a network. Remote memory requests may have to be routed through intermediate processors and/or small routing switches. For example, CMU's PLUS [3].

- *switching-network*: A global switching network interconnects all of the processors; hence, all processors are equidistant from each other. If the processors are associated with local memory, memory accesses will be non-uniform (being either local or remote). Examples include the BBN Butterfly [1] and RP3 [18].

Architecture classes	Natural clusters suitable for pools	Overhead Sources			
		1	2	3	4
hierarchy	subhierarchies	S	S	S	S
hierarchy of shared caches	subhierarchies	W	S	S	S
multi-grid of buses	processor rows or columns on same bus	S	S	S	S
general network	subareas of processors	S	S	S	S
switching network	none	W	W	S	S

Table 1: Classification of NUMA architectures.

In each of the architectural classes presented, memory may or may not be associated with each processor (although processor-memory pairs are used in many of the example systems). The NUMA architectural classes and the severity of the in-pool overheads are summarized in Table 1. It is apparent that the in-pool overhead is present in all of them, but to varying degrees. The degree to which we feel each of the overheads is applicable is indicated by using an 'S' for strongly applicable and an 'W' for weakly applicable.

In addition to the architecture, the operating system and the applications (in particular, their memory sharing and task dependency patterns) all affect the level of in-pool overhead in a particular system. To remain independent of particular systems (beyond assuming NUMA) and job characteristics, we consider all the potential sources of overhead related to the pool size together, by modeling the in-pool overhead with a linear function that is used as an inflation factor on the parallel thread execution time:

$$w = \begin{cases} 0 & \text{if } p \leq p_0 \\ W \times (p - p_0) & \text{if } p > p_0 \end{cases}$$

where W is a parameter that decides the severity of the overhead, and p_0 determines a pool size below which no in-pool overhead is incurred. Based on the success of small-scale multiprocessors, small pools are not likely to introduce significant amounts of in-pool overhead. A default value of 5-10 for p_0 seems to be reasonable for most of the NUMA architectural classes identified. For some architectures and jobs the actual function may well be superlinear or sublinear, and may have sudden jumps as the pool size goes beyond the size of a natural processor cluster in the system (i.e., forming a step function). Nonetheless, we still feel that a linear function with an offset of p_0 and varying slope W can capture the essence of the in-pool overhead.

Given a value of w, each of the parallel threads of a job running in a pool with more than p_0 processors requires $(1 + w)$ times as much CPU time as their base values. In the extreme case of a single 60-processor pool, default values of $W = 0.5\%$ and $p_0 = 10$ result in an in-pool overhead of 25%. In other words, if a parallel application randomly spread its threads over a 60-processor system, it would be slowed down by 25% compared to the hypothetical case of its running in a system with a single, fast bus connecting 60 processors with no contention. This degree of overhead is consistent with the measurement results reported in [19]. Note that the in-pool overhead is based on the pool size, rather than the number of threads in the pool, the reason being that the threads are executed on idle processors and may move around in the pool, depending on the in-pool scheduling algorithm, thus much of the overhead remains. For example, over time, a single thread of an application may execute on several processors within a pool while continuing to reference the same memory locations. The time to access this memory will be a function of the distance from the thread's current processor to the memory (and thus the size of the pool).

Pool-spanning overhead. While in-pool overhead reflects the overhead incurred by running a job within that pool, similar overheads will result from running a job in more than one pool, for similar reasons. That is, a larger number of processors are involved and the localization of execution may be decreased. These overheads have not been charged in the in-pool overhead and must therefore be reflected in the pool-spanning overhead. Note that these are not overheads that are being "double-charged" but rather overheads for which a portion is charged because of in-pool overhead and a portion is charged to pool-spanning overhead.

Besides the four main sources already outlined for in-pool overhead, a job that executes in more than one pool will also be subject to other types of overhead. These overheads include the maintenance of the pools and their loads, as well as accessing that information. However, the main source of overhead will be the coordination required between pools. For example, if the threads of a job execute on processors belonging to more than one pool, multiple scheduling and memory management modules of the operating system may be involved. It is therefore reasonable to expect that the overhead of spanning k pools of p processors each would incur greater overhead than executing in a single pool containing kp processors. The overhead incurred by spanning pools is also positively correlated with the pool size for two reasons. First, it is believed that (as in the in-pool case) the amount of overhead incurred will be greater in larger pools because the possibilities and costs of accessing memories or caches over greater distances will increase with the size of the pools as will contention for the interconnect and software servers and the amount of effort involved in maintaining cache coherency. Second, if coordination between pools is required, the cost associated with that coordination may depend on the size of the pools (larger pools may require more information).

As is the case of in-pool overhead, we assume a very simple form for *pool-spanning overhead*:

$$s = S \times (k-1)p,$$

where S is a parameter controlling the severity of spanning overhead. Like in-pool overhead, this overhead is represented as an inflation factor on the threads' CPU demand. For example, suppose a job spans six pools of five processors each, then, with a default value of $S = 1.0\%$, the job incurs a 25% pool-spanning overhead. Note that the values of W and S should be chosen such that the combined overhead of a job running on a fixed number of processors increases with the number of pools into which these processors are divided. The factor $k-1$ is used instead of k because there is no spanning overhead when $k = 1$ (i.e., a job uses a single pool).

The in-pool and pool-spanning overheads have a direct impact on the performance of pool-based scheduling algorithms, because these algorithms are based on the success of using smaller pools to reduce the in-pool overhead, and spanning pools only if there are performance advantages.

Run queue contention overhead. When the executing thread on a processor changes, it is necessary to exclusively access the shared run queue. This may result in contention overhead, which grows with the number of processors sharing the queue [16, 15]. We do not model such contention explicitly, but rather approximate it using the following formula:

$$\textit{contention cost} = \begin{cases} 0 & \text{if } p \leq p_0 \\ r \times (p - p_0) & \text{if } p > p_0 \end{cases}$$

where r is the slope of the line that reflects the severity of run queue contention, and p_0 is as used for in-pool overhead. Such a function tends to underestimate the amount of overhead for larger pool sizes, since contention for a shared resource tends to grow superlinearly. Unlike the percentage overheads associated with processor pools, this and the cache loading overhead described below are absolute (in units of time), and are incurred at context switching time.

Cache loading overhead. Modern multiprocessor systems employ large caches in order to keep the processor busy doing useful computation. When a thread is started/resumed, its cache context needs to be loaded. This may slow down applications significantly. Although the performance degradation due to cache loading is dependent on the particular jobs and threads involved, we choose to model this overhead very crudely by charging a constant cost of c time units for each context switch if the new thread belongs to a different job from that of the previously running thread. Otherwise, the overhead is a fraction t of the above, as part of the cache context of one thread may be usable by another thread of the same job. Also, threads of the same job execute within the same address space and therefore do not incur the additional overhead of switching address spaces as would a thread from a different job.

Other overheads incurred during a context switch (in addition to cache loading), are the basic cost of saving and loading the registers and swapping other process context information, as well as actually making the scheduling decision. Although we expect such costs to be low compared to the cache loading cost, we simply include them in what we call the cache loading overhead. The default c value used in simulation is 0.005 time units.

Par	Meaning	Def
P	number of processors in the system	60
L	system load level	60%
D	average desired parallelism	20
T	time slice (units of time)	10
F	level control for degree of pool spanning	0.8
G	level control for actual parallelism	0.5
p_0	pool size threshold for overheads	10
t	fraction of cache load cost for thread	0.25
W	factor for in-pool overhead	0.5%
S	factor for pool-spanning overhead	1.0%
r	factor for run queue contention overhead	0.003
c	factor for cache loading overhead	0.005

Table 2: Simulation parameters and their default values.

In addition to the above four overheads, accessing the pool load information and executing the job placement algorithm also incur certain overhead. Given the relatively low frequency of parallel thread placement, and the fact that most accesses to the pool load information are reads, we expect such overheads to be low compared to the ones we have discussed, hence, we chose to not model them.

The parameters used in the experiments, described in the next section, are listed in Table 2 together with their default values. The default values for F and G are chosen to produce optimal performance. Our experiments are typically conducted by using all of the default values and varying a few of the parameters being studied.

4 Simulation Experiments

4.1 The Benefits of Pool-Based Scheduling

To assess the benefits of pool-based scheduling, we simulated the operation of a 60-processor system employing the limited-spanning algorithm with user-level parallelism restriction activated.[4] All of the default values for the parameters shown in Table 2 are used, except that the value of the average desired parallelism, D, is varied. The results are shown in Figure 3.

[4]The system size of 60 is chosen to be large enough to expose the potentials of processor pools, yet small enough to enable us to perform a large variety of experiments and obtain reasonably tight confidence intervals. Later in the paper, we examine a 120-processor system.

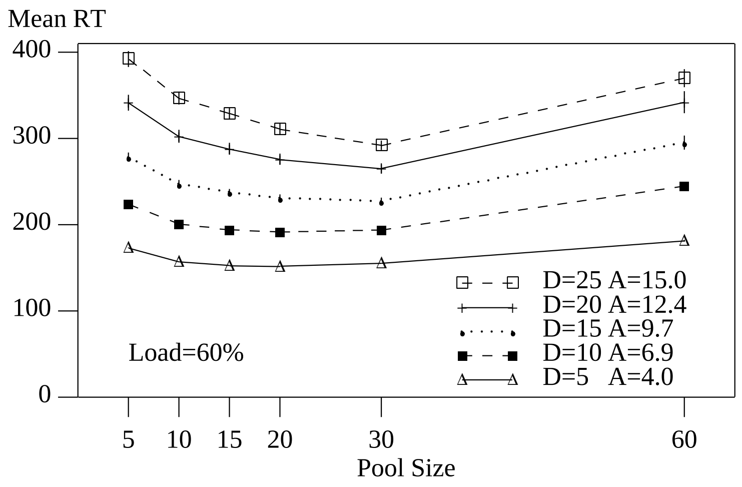

Figure 3: Benefits of pool-based scheduling.

In this and all of the subsequent figures, the vertical line segments at the data points show the 90% confidence intervals.[5]

With $D = 20$ and at a moderate load level of 60%, the average job response time is reduced from 342 (time units) when one pool is used to 265 when two pools of 30 processors are used, whereas the absolute minimum (non-achievable) response time is about 127 (which is the sum of the execution times of the three phases in the absence of overheads). The significant performance improvements when processor pools are used substantiate our hypothesis that pool-based scheduling can improve the average job response time. At the large-pool-size extreme, where the entire system is treated as a single pool, performance degrades because the parallel threads of the jobs are scheduled on the processors irrespective of their relative positions in the system. The poor locality, low cache affinity and high contention for the ready queue cause high in-pool overhead, resulting in poor performance. At the small pool extreme, jobs are forced to span pools to take advantage of the idle processors in a number of pools. Consequently, the pool-spanning overhead increases, again resulting in poor performance. Using intermediate pool sizes and limiting the number of pools a job is allowed to span according to its desired parallelism and the current load, allows most jobs (with the exception of the very large ones) to run efficiently within a single pool. Such execution only incurs moderate in-pool overhead.

Comparing the shapes of the curves, we notice that the performance advantage gained by using processor pools increases with the average parallelism, suggesting that pool-based scheduling is increasingly desirable for workloads consisting of larger parallel jobs. Comparing the positions of the curves, we observe that, with a constant load level (at 60% in Figure 3), performance degrades as the parallelism in the jobs increases. This is intuitive for several reasons:

[5]Some of the 90% confidence intervals we obtained are very narrow, and are invisible in the figures.

1. The workload becomes more bursty as a larger number of parallel threads are created at the start of a job's PARA phase and the probability of pool spanning also increases.
2. The difference between the number of desired threads and the number of actual threads increases as the load increases. So fewer threads are executing a larger amount of work.
3. The response time of a job is dependent upon the completion of all of the parallel threads. The probability of one of the threads being delayed increases with the degree of parallelism.

The pool size resulting in the best performance is positively correlated with the mean parallelism of the jobs. The results indicate that it is desirable to use larger pools for larger parallel jobs.

From Figure 3, we can also observe that, at a load level of 60%, the average actual parallelism, A, is significantly smaller than the average desired parallelism, D, especially for the larger values of D.[6] Such effective restriction of parallelism is desirable under moderate to heavy load conditions since it reduces the run queue contention and cache overheads, thus improving performance.

It may be argued that, when relatively large pools are used, the parallel threads of a small job should be placed onto processors close to each other to exploit locality within a pool and reduce in-pool overhead. This complicates the scheduling of the threads, however, since the threads would be restricted to a subset of the pool's processors. Load imbalance could also become a significant problem. Using smaller pools would achieve similar effects without these complications. Another possibility is to divide the system into pools of non-equal sizes and select pool(s) for a job's execution based on its parallelism as well as the current load. We intend to explore this possibility in our future work.

4.2 Overhead Sensitivity

While the results obtained are encouraging, we hasten to add that they depend on the overhead assumptions made. The values of the overheads, in turn, depend on the system and the workload. We therefore conducted a sequence of sensitivity studies.

Figure 4 shows the impact of the in-pool and pool-spanning overheads on performance. It is clear that the desirability of processor pools increases with these overheads, as might be expected. Significant performance improvements are seen over a relatively wide range of pool overheads. Only results for a few pairs of values of W and S are shown in Figure 4; however, since the performance with very small and very large pool sizes is largely influenced by pool-spanning and in-pool overheads, respectively, the performance curve with any combination of the W and S values shown in Figure 4 may be approximated by combining the respective halves of the relevant curves shown.

We also examined the effects of the time slice as well as the run queue contention and cache loading overheads on our results by running the same experiments as shown in Figure 3, except we varied the quantum sizes instead of the load (varying the time slice produces the

[6]Larger pool sizes yield slightly higher values of A; the values of A shown in Figure 3 are the averages over all the pool sizes simulated.

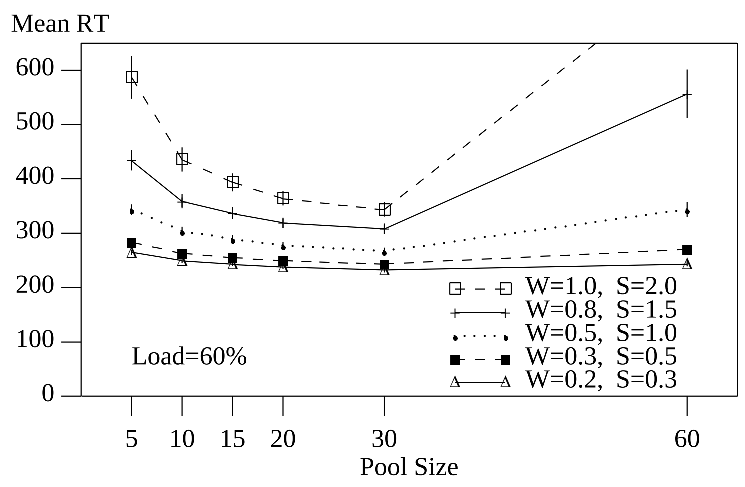

Figure 4: Performance sensitivity to pool overheads.

same effects on the mean job response time as varying the run queue contention and cache loading overhead since there is a direct correlation between the size of a time slice and the number of context switches). Figure 5 shows that, with relatively large quantum sizes, the impact of the run queue contention and cache loading overheads is not significant, thus the desirability of processor pools is prominent. With very small quantum sizes (e.g., $T = 1$), performance degrades due to the run queue contention contention. One could argue, however, that it is undesirable to use very small quantum sizes and pay such high run queue contention and cache overheads.

4.3 Degree of Pool Spanning

In Section 3, we pointed out that the degree to which jobs are allowed to span processor pools represents a tradeoff between localization of parallel jobs and load balancing. The parameter F controls the degree of pool spanning (see Section 3.2). All of the previous experiments were conducted using the limited-spanning algorithm with the default value of $F = 0.8$. Figure 6 shows the performance of the system with varying values of F.

We can see that the default value of $F = 0.8$ yields near-optimal performance for all pool sizes. Figure 6 also highlights the importance of an appropriate level of pool spanning. The unlimited-spanning algorithm performs poorly with intermediate pool sizes because of too much spanning. Although idle processors in the entire system can be used, resulting in larger actual parallelism and good load balancing, the decreased locality in the parallel threads more than offsets any performance gain from spanning. In the other extreme, the no-spanning algorithm with intermediate pool sizes performs very well on a workload with an average desired parallelism of 20. Although a few large parallel jobs are forced to execute within a single pool, they do not seem to have a substantial impact on the average response time. Such a strict limitation on large parallel jobs, however, may be unacceptable, as one

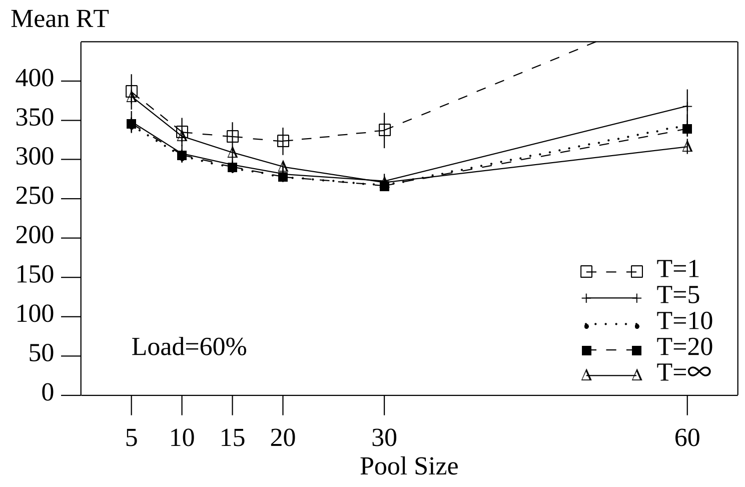

Figure 5: Performance sensitivity to CPU quantum size.

advantage of a large parallel system (i.e., being able to run large parallel jobs) is completely sacrificed to improve overall system performance. With very small pools (e.g., five-processor pools), the performance of the no-spanning algorithm degrades rapidly.

4.4 Performance with Various Load Levels

Figure 7 shows that the benefits of pool-based scheduling increase with the system load, as does the importance of the pool size. Under heavier loads the pool overheads have a stronger impact on performance. Pool-based scheduling is most desirable under moderate to heavy system load. The values for U in the figure are the measured average processor utilizations, and are higher than the load levels, L.

4.5 System Size

Figure 8 shows the performance of processor pools in a system of 120 processors. Our observations in Section 4.1 for a 60-processor system also hold for this larger system. Pool-based scheduling becomes even more desirable, since larger systems make the consideration of locality imperative; scheduling the jobs in the entire system with no regard for locality may result in intolerable performance. We again observe a positive correlation between the average parallelism and desirable pool sizes.

To allow comparison, Figure 8 also includes two curves taken from Figure 3 for a 60-processor system with an average desired parallelism of 10 and 20 (shown with dotted lines). Comparing these two curves to their corresponding curves for a 120-processor system, we notice that, using pool-based scheduling, the system performance, with a workload of fixed average desired parallelism and load level, improves as the system size increases. Thus, the capacity of the system in supporting a particular class of workload is truly scalable. With

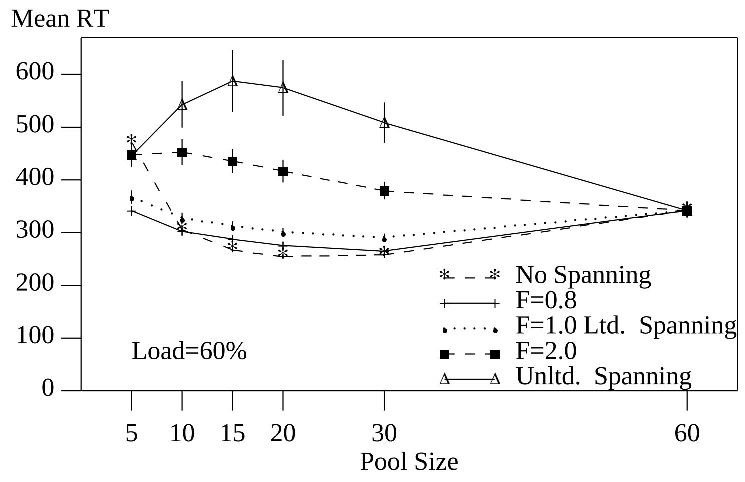

Figure 6: Performance sensitivity to pool spanning.

increasing system size, but constant pool size, the number of pools increases, providing greater opportunities for load balancing. Thus, an incoming job is more likely to be able to find a pool with a relatively large number of idle processors, and is able to use a larger number of actual parallel threads. This reasoning is supported by the increased average actual parallelism, A, for the 120-processor system shown in Figure 8 (as compared to that for the 60-processor system). For instance, with $D = 20$, A increases from 12.4 in a 60-processor system to 14.9 in an 120-processor system.

The improved performance with intermediate pool sizes in a larger system is in sharp contrast with the much poorer performance of the same type of workload (e.g., $D = 20$ and $L = 60\%$) in the same system not using processor pools. Although our linear assumption about the in-pool overhead may be too high in a very large system (because applications that are able to employ a large number of processors effectively must be able to exploit locality to a greater extent than we have modeled), it is clear that scheduling parallel threads in a large system with no regard for locality is likely to produce high overhead and hence poor performance. We therefore conclude that pool-based scheduling is not only a technique for achieving better performance in a system of a particular size, but also of importance in building scalable systems.

4.6 Revisit: Model, Algorithms, and Overheads

To focus on the key issue of pool-based scheduling, we used a simple system model, a set of simple workloads, and a class of scheduling algorithms that are tailored to large-scale NUMA systems but lack many features that one would expect to find in an optimal scheduling algorithm. Our overhead assumptions are also simplistic. It is therefore necessary, after presenting our experimental results, to assess the potential influences of these simplifications upon our conclusions.

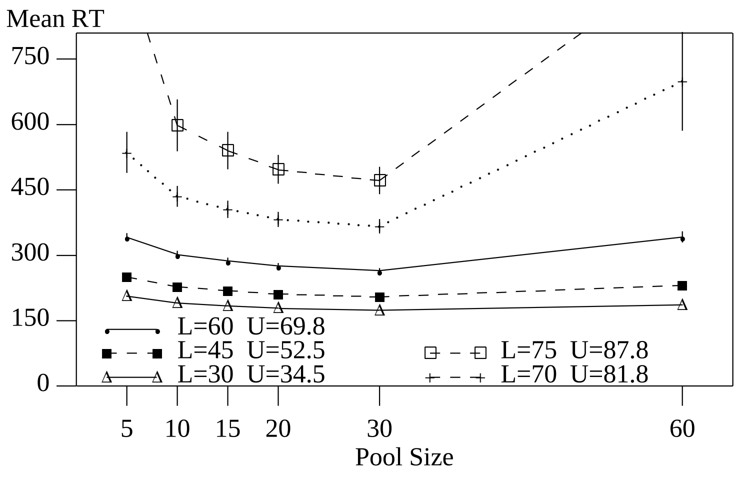

Figure 7: Performance sensitivity to load levels.

The scheduling policy for each pool uses a single run queue, and treats the threads from various jobs independently. While better performance may result if features such as coscheduling [17] (for those jobs executing within a single pool) and per-processor run queues [7] are included, we feel that the scheduling policy used within the pools are largely orthogonal to the concept of pool-based scheduling. Had a better in-pool policy been adopted in our simulation, the curves in Figure 3 would have decreased uniformly, but their shapes and relative positions would likely be similar. We tested this hypothesis by running the experiments shown in Figure 3 with the cache loading and thread scheduling overheads turned off (not shown); without these overheads, the policy based on a single run queue per pool achieves maximum load balancing. The results are as expected. However, if coscheduling proves to be desirable for jobs that span pools, then the use of processor pools would results in extra overhead in coscheduling.

User-level scheduling has been shown to be effective in reducing cache and scheduling overheads [22, 12, 21]. We included a simplified version of user-level scheduling in our algorithms in the form of limiting the number of parallel threads created depending on the desired parallelism and the system load. However, no dynamic adjustments to the number of threads a job may have are allowed (the adjustment would typically result from a change in the load). Since we observed that cache and scheduling overheads are insignificant with large quantum size, we do not expect that further refinement of user-level scheduling would change our conclusions.

We assume that the pool and run queue contention overheads are present only when the pool size is greater than p_0. All the preceding experiments are based on a default value of 10 for p_0. Our experiments using values between 5 and 10 showed little variation in the effectiveness of processor pools.

Our simple job model does not include I/O and paging, and only a token representation

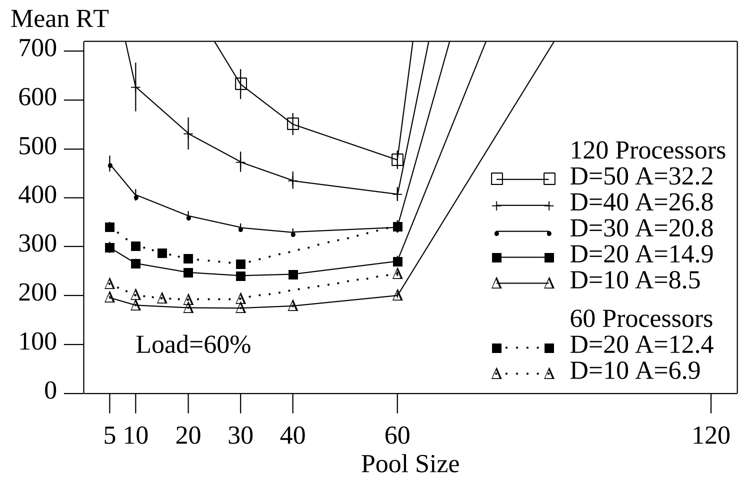

Figure 8: Performance of processor pools in a larger system.

of synchronization. We feel that the I/O and paging activities are relatively independent of the placement of the threads, unless the locations of the I/O devices or memory contention are considered in the placement and therefore expect that our results would not be changed significantly. However, if such activities cause very frequent context switching, thus making the cache and scheduling overheads more important, then the benefits of pool-based scheduling may become obscured and insignificant.

For most of our experiments we used a simple correlated workload. To examine the workload model's impact on our algorithms we also ran experiments using a workload consisting of jobs whose total CPU demand and degree of parallelism are uncorrelated. Such a workload contains much more variability in that there are jobs with a few parallel threads that require substantial CPU time, and highly parallel jobs with little CPU demand. The simulation results based on such a workload also determined that processor pools significantly improve performance.

5 Related Work

The multiprogrammed use of multiprocessors started to attract attention of researchers only a few years ago, but has since become a very active area of research. Ousterhout proposed the idea of coscheduling in his 1982 paper [17] as a way to coordinate the scheduling of a group of communicating processes. Majumdar, Eager and Bunt first examined dynamic, static, and semi-static approaches to global scheduling in multiprogrammed multiprocessor systems [14]. Eager, Lazowska, and Zahorjan proposed the use of average parallelism as a job characterization that is important for scheduling purposes [8]. Sevcik pointed out that a small number of improved job characterization parameters may substantially improve the quality of scheduling decisions [20]. In a recent simulation study, Zahorjan and McCann concluded that dynamically changing the number of processors allocated to a job in response to its changes in parallelism performs better than static policies over a range of workloads [24]. Leutenegger and Vernon undertook a comparison study of scheduling algorithms to identify the most significant characteristics of scheduling algorithms [13].

The idea of user-level, application-initiated methods for limiting the number of a job's threads in order to reduce scheduling and context switching overheads was introduced by Vandevoorde and Roberts [22], and refined by Junkin and Wortman [12]. Tucker and Gupta studied the dynamic adjustment of the number of threads each job may have in order to distribute the processing resources evenly among the jobs, as well as to reduce overhead [21].

The idea of partitioning the processors of a system among jobs is not a new one. Tucker and Gupta suggested implementing a number of *processor groups* primarily for the support of individual parallel and sequential jobs [21]. Black also proposed the use of *processor sets* for individual applications [4]. Our concept of processor pool, however, is different from the above in that they are not used to partition the processors among the jobs, but rather, are used as a scheduling unit, with possibly multiple jobs running in a single pool, and jobs spanning multiple pools (thus the use of the word "pool").

Almost all of the previous research has been concerned with small to medium UMA multiprocessors. Little research has been performed on scheduling of large-scale NUMA multiprocessors for multiprogrammed use. Ni and Wu addressed the issue of run queue contention by proposing the use of multiple run queues in a system [16]. Nelson and Squillante studied the same problem, and suggested methods that allow a processor to dequeue multiple threads or schedule threads onto other processors [15]. In a recent study, Feitelson and Rudolph study a scalable distributed hierarchical control structure for gang scheduling in a large-scale multiprogrammed environment [9].

6 Conclusion

In this paper, we proposed a class of scheduling algorithms based on *processor pools* for large-scale NUMA multiprocessors. Our simulation experiments use a set of simple workloads and a simple system model to show that parallel application scheduling based on processor pools may effectively reduce the average job response time. The performance improvements attained by using processor pools increase with the average parallelism of the jobs, the load level of the system, the scale of the system, the differentials in memory access costs, and the likelihood of having system bottlenecks. While allowing large parallel jobs to span multiple pools may improve the overall performance, such pool spanning should be carefully controlled. As the system size increases while maintaining workload composition and intensity we observed significant performance improvements when processor pools are used. We therefore conclude that processor pool-based scheduling may form an essential component of a scalable system.

It is important to observe that we have treated processor pools as an abstract concept based on the large scale and NUMA characteristics of a system, and not on any particular architecture. However, in actually implementing processor pools on a specific system, its NUMA characteristics should be fully exploited, as illustrated with example systems in the discussion of in-pool overhead (see Section 3.3). The values of the pool overhead factors, W and S, capture the benefits of localization, or, in other words, the penalty for ignoring the NUMA characteristics of the system. In the tradeoff between localization and load balancing, the pool-based scheduling algorithms capitalize on localization, while apparently sacrificing little load balancing.

In future work, we plan to look into the performance potential of processor pools of unequal sizes, of pools that dynamically split and merge in response to workload changes and of using pools for running different classes of jobs. We also plan to incorporate I/O activities into our model and study their impact. In coordination with our simulation and modeling work, we will use our prototype large-scale NUMA system, Hector [23], to obtain a better understanding of pool-based scheduling, and better characterizations of the overhead costs.

Acknowledgments

Partial support for this work was generously provided by a grant and a Post Graduate Scholarship from the Natural Science and Engineering Research Council of Canada and by an external research program sponsored by Digital Equipment of Canada. This work has been influenced by discussions with members of the Hurricane operating system project for the Hector multiprocessor and the Scheduling Discussion Group at the University of Toronto. Michael Stumm and the anonymous referees provided valuable comments on the content and presentation of this paper.

References

[1] Butterfly parallel processor overview. Technical Report No. 6148, BBN Laboratories, Cambridge, Mass., 1988.

[2] M. Beltrametti, K. Bobey, R. Manson, M. Walker, and D. Wilson. PAMS/SPS-2 system overview. Technical report, Myrias Research Corporation, June 1989.

[3] Roberto Bisiani and Mosur Ravishankar. Plus: a distributed shared-memory system. In *Proc. IEEE Intl. Conf. on Computer Architecture*, 1990.

[4] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[5] Hoichi Cheong and Alexander V. Veidenbaum. Directory-based cache management in multiprocessors. *IEEE Computer*, 23(6):39–48, June 1990.

[6] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Multi-level shared caching techniques for scalability in VMP-MC. In *Proc. ACM Intl. Conf. on Computer Architecture*, 1989.

[7] Stephen W. Curran and Michael Stumm. Scheduling a unix workload on small-scale, shared memory multiprocessors. *Computer Systems*, 3(4), Oct 1990.

[8] 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.

[9] D. G. Feitelson and L. Rudolph. Mapping and scheduling in a shared parallel environment using distributed hierarchical control. In *1990 International Conference on Parallel Processing*, pages I1–I8, 1990.

[10] D. Gajski, D. L. Kuck, D. Lawrie, and A. Sameh. Cedar - a large scale multiprocessor. In *Proc. IEEE Intl. Conf. on Par. Proc.*, 1983.

[11] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: a new large-scale cache-coherent multiprocessor. In *Proc. IEEE Intl. Conf. on Computer Architecture*, 1988.

[12] M. Junkin and D. Wortman. Supervisor - an approach to controlling concurrency. Technical report, Computer Systems Research Institute, University of Toronto, March 1989.

[13] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the ACM SIGMETRICS Conference*, pages 226–236, 1990.

[14] Shikharesh Majumdar, Derek Eager, and Richard B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of the ACM SIGMETRICS Conference*, pages 104–113, 1988.

[15] Randolph D. Nelson and Mark S. Squillante. Analysis of contention in multiprocessor scheduling. Technical Report 89-11-06, University of Washington, November 1989.

[16] Lionel M. Ni and Ching-Farn E. Wu. Design tradeoffs for process scheduling in shared memory multiprocessor systems. *IEEE Transactions on Software Engineering*, 15(3):327–334, March 1989.

[17] John K. Ousterhout. Scheduling techniques for concurrent systems. *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, October 1982.

[18] G. F. Pfister and et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture. In *Proc. IEEE Intl. Conf. on Par. Proc.*, pages 764–771, 1985.

[19] Jr. Richard P. LaRowe and Carla Schlatter Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. Technical Report CS-1990-10, Duke University, April 1990.

[20] Kenneth C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. *Proceedings of ACM SIGMETRICS Conference*, pages 171–180, 1989.

[21] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *ACM Symp. on Operating Systems Principles*, pages 159–166, 1989.

[22] Mark T. Vandevoorde and Eric. S. Roberts. Workcrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[23] Z. G. Vranesic, M. Stumm, D. M. Lewis, and R. White. Hector - a hierarchically structured shared memory multiprocessor. *IEEE Computer*, 24(1):72–79, January 1991.

[24] John Zahorjan and Cathy McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of the ACM SIGMETRICS Conference*, pages 214–225, 1990.