FORTH-ICS / TR-094 August 1993

Locality-Based Scheduling for Shared-Memory Multiprocessors

Evangelos P. Markatos[*][†] *and Thomas J. LeBlanc*[*]

Abstract

The last decade has produced enormous improvements in microprocessor performance without a corresponding improvement in memory or interconnection network performance. As a result, the relative cost of communication in shared-memory multiprocessors has increased dramatically. Although many applications could ignore the cost of communication and still achieve good performance on the previous generations of shared-memory machines, good performance on modern machines requires that communication be reduced or eliminated. One way to reduce the need for communication is to use scheduling polices that exploit knowledge of the location of data when assigning processes to processors, improving locality of reference by co-locating a process with the data it will require. This chapter presents an overview of the tradeoffs to be made in process scheduling, and evaluates locality-based scheduling techniques at the level of the operating system kernel, thread package, and parallelizing compiler.

---

[*]Computer Science Department, University of Rochester, Rochester, NY 14627, USA
[†]Institute of Computer Science, F.O.R.T.H., Crete, Greece
E-mail {markatos,leblanc}@cs.rochester.edu
A version of this paper will appear in Zomaya (Ed.)
Current and Future Trends in Parallel and Distributed Computing,
World Scientific Publishing, 1994

1 Introduction

Shared-memory multiprocessors consist of a set of processors that access memory using an interconnection network or bus. In bus-based, cache-coherent (CC) machines like the Sequent Symmetry (also referred to as UMA, or uniform memory access, machines), there is a single global memory attached to the bus. Each processor has a local cache, which brings data from the global memory as needed; cache coherence is maintained in hardware. In distributed shared-memory (DSM) machines like the BBN Butterfly family of multiprocessors (also referred to as NUMA, or non-uniform memory access, machines), each processor has a local memory, but may access the local memory of another processor using the interconnection network. The most recent member of the Butterfly family, the TC-2000, incorporates local caches, but requires that cache coherence be maintained in software.

Despite these differences in the memory hierarchy, most shared-memory multiprocessors employ off-the-shelf microprocessors. Unfortunately, the dramatic improvements in microprocessor performance due to recent advances in VLSI and RISC technology have not produced a corresponding improvement in application performance on shared-memory machines. Just as increased integer performance does not produce a corresponding improvement in operating system performance [Anderson *et al.*, 1991b; Ousterhout, 1990], an increase in computational power in shared-memory multiprocessors does not guarantee a corresponding improvement in application performance because communication quickly becomes the bottleneck. As computational power continues to increase, application performance depends more and more on the extent to which communication costs can be reduced.

One way to reduce communication costs is to use scheduling polices that exploit knowledge of the location of data when assigning processes to processors, improving locality of reference by co-locating a process with the data it will require. This type of policy stands in stark contrast to the policies in use on many shared-memory machines, which attempt to maximize the utilization of processors, rather than minimize the need for communication. For example, many operating systems use a central work queue for scheduling processes; with a central work queue, no processor remains idle as long as there are processes on the queue. Under this type of scheduling policy, a process runs on the next available processor, even if this means that the data must move from one processor to another as a result of this decision.

Unfortunately, the need to reduce communication appears to be in conflict with the desire to keep all processors busy. For example, an application that executes on one processor incurs no communication overhead, but suffers from maximum load imbalance (because all processors are idle except one). On the other hand, if the same application executes on many processors, then the load may be more evenly spread among the processors, but the application may experience high communication overhead because processors need to cooperate in order to execute the parallel application. In general, there are three different dimensions of overhead that must be considered by any scheduling algorithm.

- *Load imbalance*: To maximize the utilization of the hardware (and hopefully improve execution time), the scheduler should strive to keep all processors busy.

- *Synchronization overhead*: The distribution of work to processors, and any redistribution of work needed to alleviate load imbalance, introduces synchronization overhead among processors. This overhead can be substantial on large-scale machines, and should be minimized by the scheduler.

- *Communication overhead*: Moving a process from one processor to another (or naively assigning a process to any available processor) may also result in moving data from one processor to another. The scheduler should attempt to minimize the amount of communication introduced as a by-product of scheduling decisions.

In this chapter we discuss the scheduling problem in shared-memory multiprocessors at three levels: the operating system kernel, the thread library, and the run-time system of a parallelizing compiler. We consider several scheduling alternatives at each level, and discuss how the relative advantages of different approaches to scheduling change with an increase in the cost of communication.

In section 2 we discuss the role of the operating system kernel scheduler in scheduling for locality, particularly with respect to the issue multiprogramming. Section 3 describes a scheduling technique for use in thread libraries that reduces the amount of communication introduced as a by-product of thread scheduling. Section 4 describes a loop scheduling algorithm for use in parallelizing compilers that accomplishes a similar goal. Section 5 summarizes our results and presents our conclusions.

2 Kernel Scheduling

The lowest level of scheduling in a multiprocessor system is implemented by the operating system kernel. The main goal of the kernel-level scheduler is to share the multiprocessor among applications, a goal which is usually achieved using multiprogramming. In order for multiprogramming to be effective however, any overhead directly or indirectly caused by multiprogramming must be minimized. There are several potential sources of overhead in a multiprogrammed multiprocessor environment, and each can significantly affect system performance.

Context switch overhead is introduced whenever processes share a processor. Although many multiprocessor thread packages provide an efficient user-level context switch mechanism, kernel intervention is still required if several kernel processes share a processor. The frequency of context switching through the kernel, and therefore the amount of context-switch overhead, depends on the quantum size (when processes share a processor using time-slicing) and the frequency of communication or synchronization (which may cause one process to block and another to run).

A second source of overhead is due to preemption in multiprogrammed systems that use time-slicing. If a process is preempted while inside a critical section or while computing some condition on which other processes depend, then processes may waste their quantum waiting for the preempted process to run. If processes spin while waiting, then processor cycles are wasted by spinning. Even if processes block during synchronization, they must perform a context switch and lose the remainder of their quantum.

A third source of overhead is the cost of cache reloading, remote memory references, and migration incurred when a process is moved from one processor to another. During execution, a process builds state on a processor, either in the cache of a cache-coherent multiprocessor, or in the local memory of a distributed shared-memory (NUMA) machine. If the process is then assigned to another processor, it must reload the cache on a cache-coherent machine, and issue remote references or migrate the contents of memory on a NUMA machine. Even if a process is not moved to a different processor, other applications can corrupt its cache while the process is preempted, forcing a cache reload. If the cache miss penalty is high, the associated overhead can seriously impact performance [Vaswani and Zahorjan, 1991].

A fourth source of multiprogramming overhead arises whenever parallel applications share processors. Every parallel program strikes a balance between the benefits of parallel execution and the overhead of parallelism *in the absence of processor sharing.* When a multiprogramming policy causes applications to share a processor, the overhead of parallelism remains, but the effective speed of the processors appears to decrease. As a result, the balance between the costs and benefits of parallelism embodied in a program can be upset by the multiprogramming policy, which results in inefficient execution. In particular, an application with nonlinear speedup will prefer a small number of dedicated processors to a larger number of shared processors, even when the aggregate processor time the application receives is the same in both cases.

It is extremely difficult to find a single multiprogramming policy that can maximize processor utilization, ensure fairness, and simultaneously address all of these sources of overhead. The costs associated with a particular policy depend on the underlying architecture: the cache miss penalty, the remote access penalty, and the cost of migration. These costs also depend on the structure of the parallel application, and in particular on the number of processes per application, the amount of state associated with a process, the frequency and type of synchronization, and the data reference pattern of the application.

Many different multiprogramming schemes have been proposed or implemented on multiprocessors, but most are derived from one of three basic approaches: unsynchronized time-sharing (time-slicing), synchronized time-sharing (coscheduling), and space-sharing (hardware partitions).

Multiprocessor time-slicing is a straightforward adaptation of uniprocessor time-slicing, and is frequently employed in operating systems derived from uniprocessor systems. Each application (or process) is given a fair share of the machine. At the end of a quantum, a processor selects the next process to run from the ready queue, which may or may not be shared with other processors. There is no coordination among the processes of an application with respect to when they run or even where they run. In particular, processes within an application might not all be assigned to different processors, and there is no guarantee that any two processes will ever run simultaneously.

Although a useful technique for balancing the load in a system, parallel applications can suffer severe performance penalties under multiprocessor time-sharing. Since there is no guarantee that an application's processes will run at the same time, processes may be blocked while waiting for a preempted process or may be required to context switch after every synchronization operation. Several studies have shown that this effect can lead to severe performance degradation [Leutenegger, 1990; Lo and Gligor, 1987a; Lo and Gligor, 1987b; Tucker and Gupta, 1989].

Many operating systems support a single centralized ready queue from which processes are dispatched according to their priority. This approach is very popular on small-scale UMA machines. However, Squillante and Lazowska [1990] have shown that by ignoring the affinity that may have been created between a process and a processor, a centralized ready queue can introduce a performance penalty of 99%, with 69% due to cache reload overhead and 30% due to increased bus traffic and contention. Their suggested solution, local ready queues for short-term scheduling and a global queue for longer-term load balancing, addresses the issue of affinity, but does not take other sources of overhead into account.

Coscheduling was originally proposed by Ousterhout [1982] to address the overhead related to synchronization. With coscheduling, the processes in an application all run at the same time. There are two important advantages to coscheduling: no process is forced to wait for another that has been preempted and processes may communicate without an intervening context switch. There are also disadvantages to coscheduling. If there are several applications in the system, the machine must cycle through each of them, during which time the caches can be expected to lose any contents related to an earlier execution [Tucker and Gupta, 1989]. Second, utilization may suffer if applications have a variable amount of parallelism, or if processes cannot be evenly assigned to time-slices of the machine.

When hardware partitions are used, no two applications share a processor. A set of processors may be dedicated to an application for a relatively long fixed interval [Black, 1990] or for the entire duration of the application. Within its own hardware partition, each application may choose to allocate one process per processor, thereby avoiding entirely the overhead attributed to multiprogramming. However, to ensure fairness and to efficiently utilize the processors, the number of processors assigned to an application might have to change when another application arrives or departs [Tucker and Gupta, 1989], or when the degree of parallelism changes within an application [Zahorjan and McCann, 1990]. Unless an application can easily adjust the number of processes it employs during execution, several processes from the same application may have to share a processor, introducing context switching and other related sources of overhead.

Tucker and Gupta [1989] proposed a combination of dedicated processor scheduling and a programming model that dynamically adjusts the number of processes in an application to equal the number of processors in the partition. Their model, which assumes the use of fine-grain threads in the application, can suspend a kernel process between the execution of two threads. Their experiments show that having one process per processor results in significant performance improvement when compared to a time-slicing policy. Subsequent work by Gupta *et al* [1991b] investigated the effects of different scheduling policies and synchronization primitives on an UMA multiprocessor using simulation. They showed that in the presence of multiprogramming, blocking primitives always outperform spinning primitives. They also showed that coscheduling and hardware partition policies are better than traditional round-robin prioritized polices due to their high cache hit ratio and low synchronization overhead. Moreover, hardware partitions along with process control [Tucker and Gupta, 1989] typically outperform coscheduling because hardware partitions usually achieve higher processor utilization.

Unfortunately, not all applications can easily adjust the number of running processes on demand. Some programming models encourage applications to create a static number of processes, so as to avoid unnecessary process creation, destruction, and context switching. Others use coarse-grain threads of control, which reduce the opportunities for dynamic adjustment. Although the programming model used by Tucker and Gupta is widely used, their work does not characterize the effects of multiprogramming on applications that do not adhere to the model.

Zahorjan and McCann [1990] simulated the performance of hardware partitions with a workload containing programs that change their parallelism frequently. They concluded that a dynamic hardware partition policy is the best choice, since such a policy can reallocate unused processors immediately. Subsequent experimental work by McCann, Vaswani, and Zahorjan [1993] on a Sequent Symmetry confirms this conclusion. The same argument may not be valid for a NUMA multiprocessor however, since processor reallocation may be too expensive to perform every time an application changes the amount of parallelism it employs. In addition, applications with a fixed amount of parallelism that synchronize very frequently may prefer coscheduling over hardware partitions, since a small hardware partition may force them to incur context switch overhead on every synchronization operation.

In this section we illustrate the differences between these three multiprogramming schemes using an implementation of each on the BBN Butterfly. We first describe the modifications to an existing operating system necessary to implement each scheme, and then present the results of experiments with these implementations running a sample application. Our experiments illustrate why coscheduling is often preferable to time-slicing, and why hardware partitions typically perform better than coscheduling. We conclude that under most circumstances, hardware partitioning is the best strategy for multiprogramming a multiprocessor, no matter how much parallelism applications employ or how frequently synchronization occurs.

2.1 Multiprogramming Implementations

Our multiprogramming experiments are based on a BBN Butterfly Plus multiprocessor running the Psyche operating system [Marsh *et al.*, 1991; Scott *et al.*, 1990]. We modified the Psyche kernel, which already supports time-slicing, to implement coscheduling and hardware partitions.

2.1.1 Time-Slicing

The Psyche kernel implements a straightforward extension of uniprocessor time-slicing. Users may create processes (represented by kernel processes) and bind them to physical processors. The kernel time-slices among the processes on a processor. Processes are never migrated.

Each processor has its own ready queue, which is sorted by process priority. Within a priority level, processes are served in a round-robin fashion. Each process gets a fair share of the processor; as in Unix, a user with many processes can get more cycles than a user with few processes.

2.1.2 Coscheduling

We implemented coscheduling in Psyche using an adaptation of Ousterhout's matrix algorithm [Ousterhout, 1982]. Coscheduling requires that process preemption be synchronized on all processors. In our implementation we use a quantum of 100 ms. To ensure that all processors begin a new quantum simultaneously, we embed a tree barrier [Mellor-Crummey and Scott, 1991] in the clock handler of each processor. The time required to synchronize 16 processors using this tree barrier is about 200 μs; the additional time required to make a scheduling decision using coscheduling is between 50 and 200 μs, depending on the number of applications. Without coscheduling the clock handler normally consumes about 200 μs each quantum, including the time to save state and make a scheduling decision. Our revised clock handler takes about 500 μs each quantum, or 0.5% of the quantum.

2.1.3 Hardware Partitions

Our implementation of dynamic hardware partitions requires cooperation between the operating system kernel and the thread library used to implement applications. The allocation of processors to applications is done by the kernel. Migration, which occurs when a partition grows or shrinks due to the departure or arrival of a new application, is implemented by the thread package.

When a new application arrives or departs the system, the kernel notifies each currently executing application about changes in its partition. If the partition shrinks, the thread library on that node may choose to either migrate the currently executing thread immediately, or finish executing the thread and then deallocate the processor. The latter option is used in conjunction with the task queue model; explicit migration is used in all other cases. If a thread is migrated, the memory object of the thread is moved to another processor in the application's partition, and the thread is placed on the ready queue of that processor.

Migration requires copying a minimum of one memory object (8K bytes). Each migration operation takes about 25 ms per memory object, which includes the cost of copying the memory object containing the state of the thread, unmapping the object in one address space, and mapping it into another. The cost of dynamically changing a system during execution from one 16-processor partition to two 8-processor partitions is about 700 ms, where each process contains 24KB of data.

2.2 Evaluation of Multiprogramming Policies

We use Gaussian elimination as a sample application to illustrate the effect of multiprogramming policy on application performance. We chose Gaussian elimination because it has several different parallel decompositions, which allows us to vary the degree of parallelism and the frequency of synchronization within a single application.

We implemented four versions of Gaussian elimination, representing different parallelizations of row elimination. The first implementation uses a very fine-grain decomposition and condition synchronization. The program creates a thread for each element in the matrix to be eliminated. Before eliminating an entry, the thread checks to see if the condition flags associated with the pivot row and the entry are set. If so, the pivot row and the row containing the entry are copied into the local memory, the computation is performed, and the result is copied back into the original matrix.

The second implementation is similar to the first, except that it uses barrier synchronization. The program creates a set of threads to eliminate the entries in a single column of the matrix. These threads synchronize using a barrier upon completion. The program then creates a new set of threads for the next column.

The third implementation uses a coarse-grain decomposition and condition synchronization. The program creates one thread per processor, and distributes the rows of the matrix among the threads in a round-robin fashion. Each thread eliminates all the entries in several rows. This implementation requires much less synchronization than the earlier implementation based on condition synchronization. In addition, there is unlikely to be much spinning, since the elimination of the pivot row is the first computation performed in each phase of execution. Most important, there are many fewer row copy operations performed with the coarse-grain decomposition: $O(N^2)$ instead of $O(N^3)$.

The fourth implementation is similar to the previous one, but uses barrier synchronization instead of condition synchronization. Each thread eliminates some elements of a single column of the matrix, synchronizes with the other threads using a barrier, and then proceeds to the next column.

For each multiprogramming policy, we ran the four implementations of Gaussian elimination under two different scenarios: (1) under ideal conditions where only one application is in the system, and (2) under multiprogramming, with an application in the background. Our multiprogramming experiments incorporate a compute-bound application in the background that consumes all the cycles it is given. Our experimental results focus on the execution time of the parallel portion of an application; the serial portion, consisting of program loading and creation of virtual processors, is not included in the timing figures.

2.2.1 Evaluation of Time-Slicing

Our main concern with time-slicing is the overhead introduced by preemption. In the absence of this overhead, we would expect a multiprogramming level of 2 (that is, two applications sharing the machine) to introduce a slowdown of 2.

Table 1 shows the running time of our four implementations of Gaussian elimination (512×512 matrix) on a 16 processor BBN Butterfly under time-slicing. The first column shows the running time of the application in stand-alone mode, the second column shows the running time when there is a single background application, and the third column shows the slowdown induced by the background application. The two implementations that use a fine-grain decomposition both take much longer to execute than the two implementations that use a coarse-grain decomposition because thread creation time dominates the fine-grain implementations, and the communication costs associated with the fine-grain implementations are much higher than those in the coarse-grain implementations.

	Stand-alone	Multiprogrammed	Slowdown
Coarse-grain w/ barriers	18.7	64.2	3.43
Coarse-grain w/ conditions	18.1	39.1	2.16
Fine-grain w/ barriers	38.4	97.3	2.53
Fine-grain w/ conditions	24.4	49.1	2.01

Table 1: Execution time (in seconds) of Gaussian elimination on 16-processor BBN Butterfly under time-slicing.

As seen in Table 1, the implementation that uses barrier synchronization and a coarse-grain decomposition suffers a slowdown of 3.43 under time-slicing when two applications share the machine. The reason for this unexpectedly high slowdown is that barrier programs are very sensitive to the effects of preemption, since the preemption of any one thread delays all threads.

The implementation that uses condition synchronization and a coarse-grain decomposition is not adversely affected by multiprogramming. With a multiprogramming level of two, this program experiences a slowdown of 2.16, which is very close to what was expected. The reason that preemption does not distort this execution is that a thread does not depend on every other thread making progress during a short interval of time, as is true with barriers. Only the thread computing the next pivot row can delay other threads when preempted.

The fine-grain decomposition with barriers experiences a slowdown of 2.53, which is an improvement on the slowdown of 3.43 suffered by the coarse-grain decomposition with barriers. The reason for this improvement is that both programs have the same number of barriers (and hence the same opportunities for problems with preemption), but the overall execution time of the fine-grain program is greater. As a result, there are three barriers per quantum in the coarse-grain program, and fewer than 1.5 barriers per quantum in the fine-grain program. The more frequent use of barriers in the coarse-grain program produces the difference in slowdown.

The fine-grain decomposition with condition synchronization experiences a slowdown of only 2.01, which is comparable to the slowdown experienced by the coarse-grain decomposition with condition synchronization. Once again, preemption does not distort this execution because a thread does not depend on every other thread making progress during a short interval of time, as is true with barriers. Condition synchronization is not frequent enough in this application for preemption to significantly affect the execution time.

None of these implementations of Gaussian elimination synchronize extremely often; even the finest-grain implementation must eliminate an element between synchronization points, and that takes several milliseconds. As a result, the implementation with barriers only executes a barrier about once every 50 *ms*. Much worse cases of slowdown are possible with smaller matrices. In particular, a

256×256 matrix problem slows down by a factor of 8 in the presence of one background application.

We can use odd-even sort to measure the effect of multiprogramming on programs that synchronize very frequently. Our odd-even sort program creates one thread on each processor. The array to be sorted is divided statically among the threads. Each thread performs N/P/2 comparisons in each phase, and then synchronizes with the other threads using a barrier. The length of a phase is a few milliseconds for an array of several thousand elements on 16 processors.

On a dedicated machine, sorting an array of 512 elements takes 286 ms. The same program run with a job in the background takes 103 seconds, a slowdown factor of 366! The problem is caused by a combination of barriers, frequent synchronization (every 500 μs), and preemption. If we modify the implementation of barriers to yield the processor to another application rather than spin, giving up the rest of the quantum but receiving the next quantum sooner, we see a slowdown of 540; yielding the processor ensures that almost no barrier is ever completed within a quantum.

2.2.2 Evaluation of Coscheduling

In order to measure the costs associated with coscheduling, we measured the running time of a coarse-grained Gaussian elimination program (256×256 matrix) under varying levels of processor sharing. The program was run on 4 processors, with 0, 1, 2, and 3 background applications. The results are shown in Table 2.

Number of Applications	Running Time	Slowdown
1	9.8	1.00
2	19.6	2.01
3	29.5	3.01
4	39.3	4.01

Table 2: Running time (in seconds) of coarse-grain Gaussian elimination program on 4-processor BBN Butterfly under coscheduling.

As seen in the table, the execution time of a single application rises linearly as the degree of multiprogramming rises. For this program, coscheduling does not appear to introduce any significant overhead.

Next we consider whether unused slots in the processor-time scheduling matrix are of much use. That is, when an application is given extra processor cycles for one of its threads at a time when the other threads in the application are not running, does this improve the execution time of the application? To answer this question, we ran the Gaussian elimination program with a background application that only uses half of the processors. This scenario creates a time-slice during which the Gaussian elimination program runs all its threads, followed by a time-slice in which half of its threads are given extra cycles. We measured the running time of the application under this scenario to be 19.4 seconds, which is a small improvement on the 19.6 seconds required by the application when no extra scheduling slots are given to it. This minor improvement in execution time suggests that unused slots do not contribute much to system throughput in the presence of synchronization. Ousterhout's simulations [Ousterhout, 1982] show that coscheduling is typically 80% effective (measured in terms of the percent of processor time spent coscheduled); this result suggests that the effectiveness can't be improved very much by utilization of empty slots in the scheduling matrix.

2.2.3 Evaluation of Hardware Partitions

Our main concern with hardware partitions is the overhead introduced when the number of threads used by an application exceeds the number of processors allocated to the application. As applications arrive and depart from the system, multiple threads from a single application may be forced to share a processor. To measure the overhead of multiplexing threads, we ran the Gaussian elimination program on a 512×512 matrix with 16 coarse-grain threads and barrier synchronization. We varied the number of processors, and observed the slowdown due to having fewer processors than threads; the results are shown in figure 1.

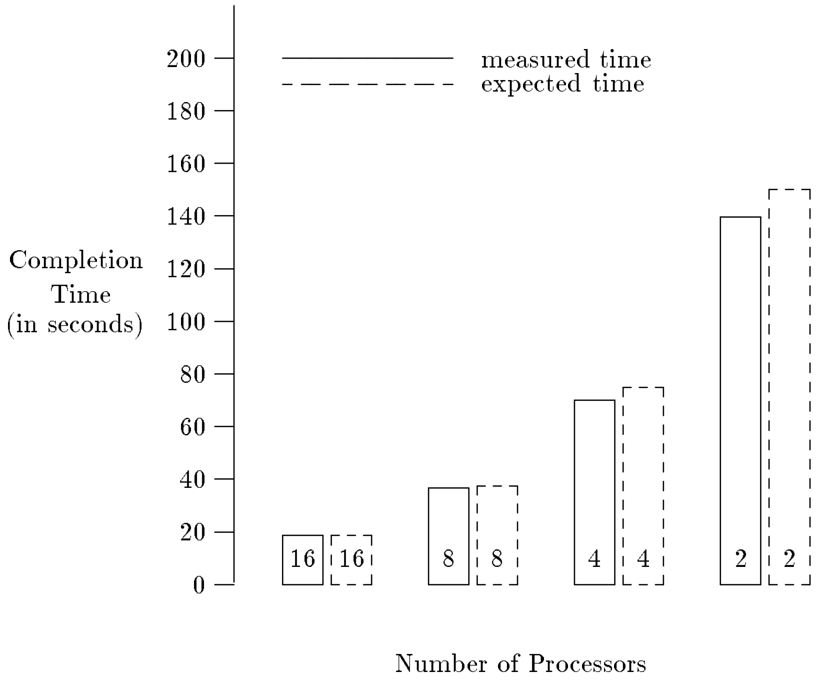

Figure 1: Running time of Gaussian elimination with 16 threads and barriers on hardware partitions.

We would expect the execution time of the program on 8 processors to be at least twice the execution time on 16 processors. The additional overhead of multiplexing threads should make the time on 8 processors even more than twice the time on 16 processors. Nonetheless, as shown in figure 1, the time required to execute the program with 16 threads on 8 processors is less than twice the time used on 16 processors. These same results were observed for the program that uses condition synchronization. One reason for the better than expected performance on 8 processors is that there is significant contention for the pivot row on 16 processors, and much less contention on 8 processors. In addition, there is a slight imbalance in the computation (due to tail effects in the division of work in the matrix), and therefore the application achieves higher processor utilization on 8 processors than on 16 processors. In general, applications can utilize 8 processors better than 16 processors because the speedup of an application is typically sublinear.

These experiments do not include the costs of migration. To measure the effects of dynamic hardware partitions, we started the Gaussian elimination program on 16 processors and then immediately introduced a background application. The arrival of the second application causes the kernel to divide the machine into two 8-processor partitions. The Gaussian elimination application migrates 8 threads from the larger partition into the new smaller partition. To isolate the costs of migration, no computation was performed by the Gaussian elimination program while holding 16 processors. The completion times of the various implementations of Gaussian elimination on a 16-processor partition and an 8-processor partition are shown in Table 3.

These results show that even with the one-time cost of migration, and the recurring cost of multiplexing threads on a virtual processor, a hardware partition of 8 processors takes less than twice as long as a 16 processor partition. The lack of linear speedup in the application, and the reduced communication and contention that results from using a smaller number of processors, argues for hardware partitions. Based on this observation, we would expect the benefits of using hardware partitions to exceed the costs in most cases.

	Stand-alone	Multiprogrammed	Slowdown
Coarse-grain w/ barriers	18.7	36.9	1.97
Coarse-grain w/ conditions	18.1	35.5	1.96
Fine-grain w/ barriers	38.4	56	1.47
Fine-grain w/ conditions	24.4	45	1.8

Table 3: Execution time (in seconds) of Gaussian elimination on 16-processor and 8-processor partitions of BBN Butterfly.

2.2.4 Comparison of Kernel Scheduling Policies

A comparison between the three scheduling policies for each version of the Gaussian elimination program is presented in figure 2. Each graph depicts the execution time of the program for each policy in the presence of a single background application.

As can be seen in Figure 2, under time-slicing a background application usually results in slowdown well above 2. Coscheduling's slowdown is only slightly higher than 2 in most cases. However, hardware partitions incur slowdown less than 2 in all cases, and in at least one case, the slowdown is substantially less than 2. Clearly, hardware partitions are preferable for this application, regardless of the number of threads used or the frequency of synchronization.

In summary, time-slicing introduces preemption, which can have enormous impact on a program, particularly programs that use barriers. Programs that don't use barriers, or synchronize infrequently, are immune to the effects of time-slicing. Coscheduling can remove the overhead due to preemption, but it has a built-in effectiveness of 80% or so, and performs poorly in cases where an application cannot effectively exploit the entire machine for all of its lifetime. Hardware partitions do not suffer from preemption, incur very little context switch overhead (assuming a user-level context switch mechanism), and reduce the need for communication (and therefore reduce the effects of contention). In addition, hardware partitions allow an application to optimize its implementation for the percentage of the machine allocated to it.

In general, there are several reasons why coscheduling can perform significantly worse than hardware partitions: (1) coscheduling results in cache corruption, whereas hardware partitions do not; (2) there are fewer remote references and less contention when fewer processors are used; (3) there is less imbalance in the computation when the total amount of work is divided among fewer processors. These factors are significant enough to more than compensate for the costs of (infrequent) migration and the additional overhead of blocking synchronization (rather than busy-waiting) required within a hardware partition when the number of threads exceeds the number of available processors. Based on these observations, we believe that in most cases the best choice of multiprogramming policy for multiprocessors is dynamic hardware partitions.

3 Thread Scheduling

Threads are a popular structuring method for parallel applications [Bershad *et al.*, 1988; Doeppner Jr., 1987; Sun Microsystems, Inc., 1990; Weiser *et al.*, 1989], and most manufacturers of multiprocessors provide a lightweight thread package as part of the standard programming environment. There are several reasons for the popularity of threads including:

- Many applications decompose naturally into fine-grain units of computation. Using a kernel process to represent each fine-grain unit of computation can be very expensive, both due to the overhead of process creation and destruction, and the fact that all scheduling must be done inside the kernel. Threads allow the implementation to reflect the natural decomposition of the algorithm without imposing enormous costs.

- Programs that use a fine-grain decomposition implemented with threads can usually run on any number of processors, and can easily adapt to a change in the number of processors (possibly

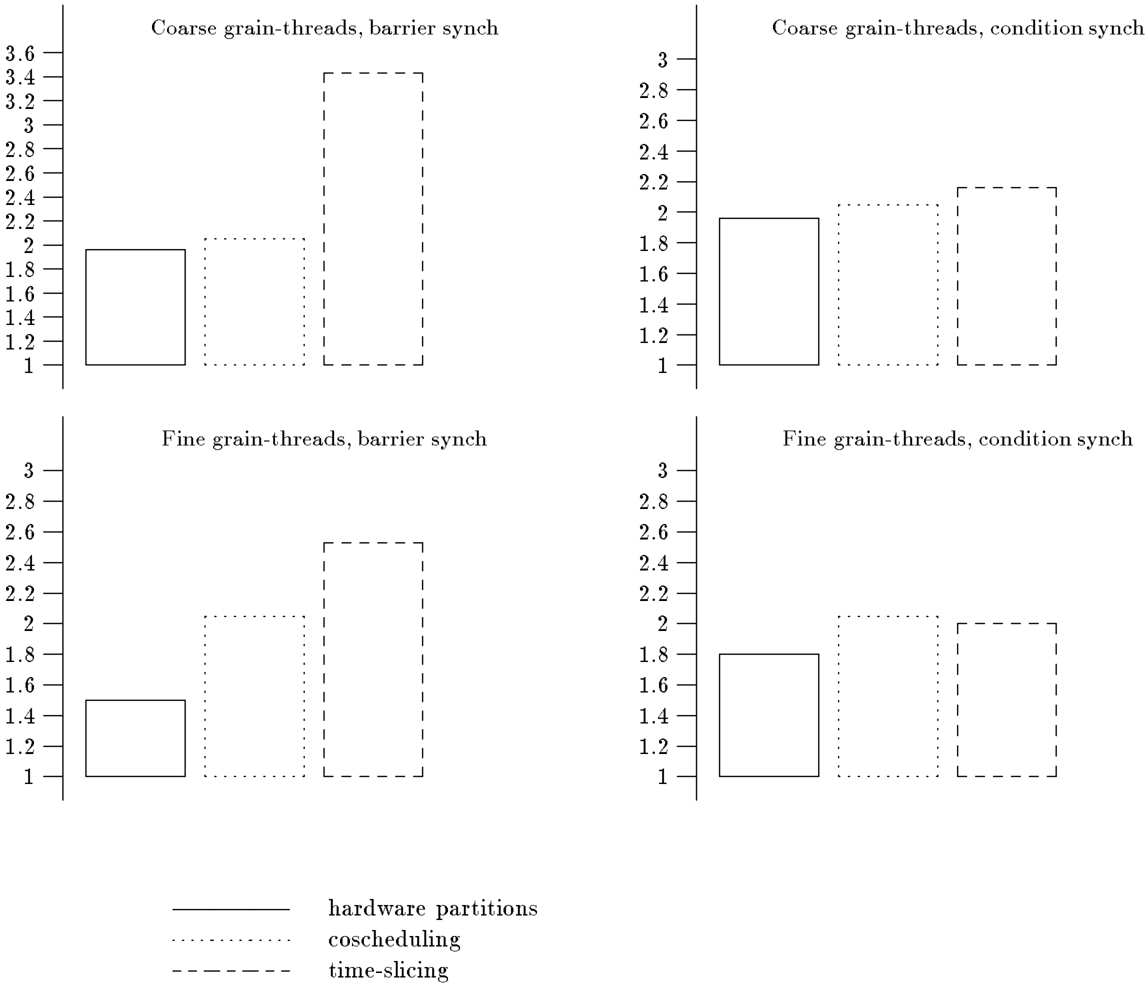

Figure 2: Relative slowdown introduced by a multiprogramming level of 2 for different scheduling policies and programming models.

due to a repartition of the machine when a new application arrives) [Tucker and Gupta, 1989].

- It is both easier and cheaper to migrate a lightweight thread, which shares it address space with other threads, than a kernel-level process with its own address space. The operating system can exploit this fact when preempting a thread that holds a critical resource, allowing the thread to complete the critical section on another processor [Anderson *et al.*, 1991a].

- Lightweight threads admit a fine-grain decomposition, which offers many opportunities to perform load balancing. In addition, load balancing can be implemented via thread placement rather than process migration.

Despite these advantages, the degree of parallelism that can be effectively exploited by an application is limited by the overhead of using threads. Historically, this overhead has been dominated by the cost of thread creation, destruction, and context switching. However, recent work has shown that the cost of these thread management operations can be drastically reduced, so that threads need only be an order of magnitude more expensive than a procedure call [Anderson *et al.*, 1989]. Under these circumstances, threads should be cheap enough to use for fine-grain parallelism.

Unfortunately, the overhead associated with lightweight threads is not limited to the cost of thread management. There is an additional cost to using fine-grain threads: the cost of bringing data into the local memory or cache where a thread executes. This cost, whether the result of explicit copy operations between local and remote memory on a distributed-memory machine, or the result of cache misses on a multiprocessor with coherent caches, can be substantial. On modern multiprocessors, which have extremely fast processors and relatively slow main memory, this overhead can dominate the execution time of a thread.

To illustrate the magnitude of this overhead, we implemented both a coarse-grain and fine-grain decomposition of Gaussian elimination on a Silicon Graphics 4D/480GTX Iris multiprocessor workstation (a member of the Power series). Using 6 processors, the coarse-grain decomposition (one thread per processor) requires 11.7 seconds to process a matrix with 400,000 elements; the fine-grain decomposition (one thread per element to be eliminated) takes 31.15 seconds on 6 processors. The factor of 3 difference in performance cannot be explained by the cost of creating threads, since it only takes about one second to create, schedule, and destroy all 200,000 threads used in the fine-grain decomposition. In addition, since threads run to completion, there is no extra context-switch overhead generated by the use of fine-grain threads. The difference between the performance of the fine-grain and coarse-grain decompositions is mainly attributed to the time required by each thread to load its data into the local cache. In the fine-grain decomposition, each thread loads an entire row of the matrix into the local cache, and references each element of the row only once or twice. The time spent loading data into the cache in the coarse-grain decomposition is small, because each processor loads a portion of the matrix into its local cache, and then references only that portion.

This example illustrates a general problem with lightweight threads: they do not execute long enough to amortize the cost of establishing their state in the local memory or cache. Even though thread operations may be very cheap, an implementation that uses fine-grain threads will typically perform much worse than an analogous coarse-grain implementation. As a result, programmers avoid using fine-grain threads, despite the many benefits of doing so.

Previous work on thread scheduling has focussed on the goal of load balancing. For example, in the process control scheme [Tucker and Gupta, 1989], Uniform System [Thomas and Crowther, 1988], Brown Threads [Doeppner Jr., 1987], and Presto [Bershad *et al.*, 1988], all threads of the same application are placed in a FIFO central work queue. Processors take threads from this queue and run them to completion. The load is evenly balanced in that no processor remains idle as long as there is work to be done.

Anderson *et. al.* [1989] argued for the use of per-processor ready queues within a thread package to improve scalability (reducing contention for the single ready queue) and to preserve processor affinity. Under this scheme, a newly created thread is placed on the ready queue of the processor on which it was created. Idle processors scan their own ready queues first, looking for threads to execute. If there are no threads in the local ready queue, a processor looks in the the ready queues of other processors.

All of these user-level policies execute a thread on the processor on which the thread was created, or on whichever processor happens to be idle when the thread reaches the front of the ready queue. Once a thread begins execution, it typically runs to completion on that processor, thus preserving cache affinity. Only in rare cases does a thread establish state on a processor and then migrate to another. However, even in cases where a thread executes on only one processor, it may spend a substantial percentage of its lifetime bringing the data it needs into the local cache. Our goal is to find an appropriate initial placement of the thread so as to reduce the time spent loading data into the cache.

This thread placement problem has been studied from the compiler's point of view. Bokhari [1987] and Polychronopoulos [1988] considered the problem of assigning a graph of communicating processes to a set of processors so that the total completion time is minimized. Bokhari showed that this problem is NP-complete, and described algorithms for special cases where the problem is polynomial. Polychronopoulos gave an optimal algorithm for a specific set of graphs, and a heuristic algorithm that finds a suboptimal solution for any graph. The key idea in the heuristic is to merge two communicating processes into one, as long as the parallelism of the program is not affected.

We propose a new thread scheduling technique, called memory-conscious scheduling (MCS), that reduces the overhead associated with loading data into local memory or cache. The distinguishing feature of this technique is the priority placed on maintaining locality of reference when scheduling threads. The basic idea is to schedule a set of threads that reference the same data on the same processor. By doing so we guarantee that only the first thread to run on a processor will have to bring a significant amount of data into the local memory or cache; other threads will be able to use the data left behind in the local memory or cache. Of course, a static assignment of threads to processors could result in load imbalance, and therefore we migrate threads when load imbalance occurs. Our experiments on the Iris and BBN Butterfly Plus multiprocessors confirm that this scheduling technique results in significant performance improvements for applications using fine-grain threads. In fact, the execution time of an application using fine-grain threads under memory-conscious scheduling is often comparable to that of the corresponding coarse-grain implementation of the same application.

3.1 Implementation Issues

Under memory-conscious scheduling, a thread executes on a processor whose local memory or cache already contains some of the data the thread will access. Since we do not expect an optimal solution to the thread and data placement problem, we do not require extremely accurate or extensive information about a program. The more information that is available, the better the thread placement decision is likely to be. However, memory-conscious scheduling can still be used even in cases where information about threads and data location is incomplete. Under those circumstances, memory-conscious scheduling might make imperfect decisions, but will almost always perform better than a scheme that does not preserve locality at all.

Even though we do not require a complete and accurate precedence graph for the program, we do require some knowledge of the distribution of data and the data access patterns of threads. This knowledge can be provided by the programmer, the thread package, or a compiler.

On a distributed shared-memory machine like the BBN Butterfly Plus, data and thread placement is often under user control. Since locality management is performed by the application programmer, through the explicit allocation and copying of data, the programmer knows the initial location of all data, and is aware of any movement of data. As a result, the programmer can explicitly assign threads to processors based on the location of data. This is the approach we used in our implementation of memory-conscious scheduling within an existing thread package running under the Psyche multiprocessor operating system [Marsh *et al.*, 1991] on the Butterfly.

BBN's Uniform System library [Thomas and Crowther, 1988] suggests an alternative approach to implementing memory-conscious scheduling on a distributed shared-memory machine. The Uniform System is a shared-memory, data-parallel programming environment. Within a Uniform System program, task generators are used to create parallel tasks (threads). Each task operates on some portion of a large, shared address space distributed evenly throughout the machine. Task descriptors are placed on a global work queue, and are removed by processors looking for work. Locality management is performed in software by the application. The programmer can use a bulk data transfer mechanism to copy data from the shared address space into local memory, where it can be modified and then copied back.

The Uniform System library knows the location of shared data, since it provides the routines to allocate memory in the shared address space. In many cases it also knows the data a thread will access; the descriptor that defines a task is usually an index into a shared data structure, and the data accessed by the task is determined by this index. Given this knowledge, memory-conscious scheduling can be implemented by the thread scheduler in the Uniform System. Although a library package cannot in general know which data will be accessed by a thread, we have found that this information is readily available for most Uniform System programs.

A machine with coherent caches, like the SGI Iris, introduces two additional complications:

- Locality management, including the placement of data, is no longer under programmer control. The hardware coherency protocol causes data to migrate or to be replicated when it is accessed.
- There is no local memory per se associated with each processor in the Iris; there is only main memory and local caches. As a result, no data is local to any processor at startup, and the initial assignment of data to caches depends on the initial assignment of threads to processors.

Due to these complications, an implementation of memory-conscious scheduling on the Iris cannot be based on the initial location of data, as is the case with the Butterfly Plus. Instead, thread placement decisions must be based on the precedence relations among threads. Since the current location of data depends on the most recent access to it, a thread should run on the same processor as its predecessor in the precedence graph.* Presumably, the data used by a thread's predecessor will still be in the cache when the thread runs.†

If a thread has more than one predecessor in the precedence graph, choosing the processor that last ran any one of them will reduce the need to load data into the cache. If the majority of the data needed by a thread was last accessed by one other thread, then the processor that executed that thread should be chosen.

3.2 Performance Implications

In this section we examine the performance implications of memory-conscious scheduling on two different multiprocessor architectures: a distributed shared-memory machine without caches (BBN Butterfly Plus), and a bus-based, cache-coherent multiprocessor (SGI Iris). For our experimental evaluation, we chose application programs whose communication patterns are representative of a large class of fine-grain parallel applications. These application programs are:

- *Gaussian elimination*: This well-known algorithm for solving a system of N simultaneous linear equations is representative of a large class of scientific applications that use vector operations. In the fine-grain decomposition, a thread is created for each element to be eliminated. Each thread adds a multiple of the current pivot row to the row of the element to be eliminated.
- *Merge sort*: This standard sorting algorithm is representative of a large class of *divide-and-conquer* problems, including convex hull, FFT, factorial, fibonacci, and the planar closest-neighbor problem. In the fine-grain decomposition, a thread is created for each recursive subdivision of the input. Each thread merges two sorted lists.

---

*If data is accessed by at most one thread, no scheduling policy can reduce the overhead of loading the data into the cache, since it has to be loaded at least once. Knowing the initial placement of data would help in this case on the Butterfly Plus, but the performance benefits would likely be small, since such a program would either entail coarse-grain threads (amortizing the cost of loading data over a long period of execution), or would terminate quickly under any reasonable policy.

†If the machine is multiprogrammed, we assume a space-sharing policy is used, in which a subset of processors is devoted to a single application.

- *Grassfire*: This nearest-neighbor algorithm to compute the depth of objects in an image represented by a binary input matrix is representative of many other parallel algorithms for successive over-relaxation, convolution, edge detection, feature enhancement, and smoothing. During each iteration, the fine-grain decomposition creates a new thread for each row in the image.

We implemented three versions of each application: a coarse-grain decomposition, a fine-grain decomposition with a load balancing policy, and a fine-grain decomposition under memory-conscious scheduling. The coarse-grain decomposition creates as many threads as processors, and assigns a part of the data to each thread. Both fine-grain decompositions create one thread for each natural unit of parallelism in the application (e.g. one thread for each element to be computed in a matrix). Under the load balancing policy, a thread is assigned to the least loaded processor. Under memory-conscious scheduling, a thread is assigned to a processor containing some of the data it will access.

3.2.1 Performance on the BBN Butterfly Plus

To quantify the benefits of memory-conscious scheduling on the Butterfly Plus, we measured the execution time of the fine-grain implementations of our applications with and without memory-conscious scheduling. We also measured the execution time of the fine-grain implementation under the load balancing policy exclusive of inter-processor communication, so as to place a bound on the benefits of any placement policy. The results of our experiments on 8 processors appear in Table 4.

Application	Load balancing	MCS	Lower Bound
Gauss elimination	101	71.5	67.3
Merge sort	0.95	0.75	0.6
Grassfire	93	67	59

Table 4: Execution time (in seconds) of fine-grain parallel applications.

As can be seen in Table 4, memory-conscious scheduling improves the performance of Gaussian elimination and Grassfire by about 28%, and merge sort by 21%, when compared against the traditional load balancing policy. Moreover, no other thread placement policy is likely to do much better, since memory-conscious scheduling is within 5-20% of the unrealizable lower bound, where communication is free. Given that communication is not free in practice, and also that parallel programs require some communication, these results suggest that memory-conscious scheduling provides nearly optimal thread placement for these fine-grain parallel programs.

3.2.2 Performance on the SGI Iris

In the previous section we showed that memory-conscious scheduling improves application performance by 20-30% on the Butterfly Plus, a distributed shared-memory machine. This result may not be surprising, given that nonlocal memory accesses are substantially more expensive than local memory accesses on the Butterfly, and software-based locality management is essential to good performance. Software-based locality management has not received much attention in bus-based cache-coherent multiprocessors like the Iris however, since the existence of coherent caches creates an illusion of uniform memory access. In this section we quantify the benefits of using memory-conscious scheduling on the Iris, and show that in general these benefits depend not on the presence or lack of coherent caches, but rather on the cost of a remote memory access (or cache miss) relative to the speed of the processor.

To quantify the benefits of using memory-conscious scheduling on the Iris, we measured the execution time of the three different versions of each of our application programs: coarse-grain threads, fine-grain threads with load balancing (LB), and fine-grain threads with memory-conscious scheduling (MCS). The results appear in Figures 3-5.

Figure 3 shows that the fine-grain decomposition of Gaussian elimination under the load balancing policy is 3 times slower than the coarse-grain decomposition on 7 processors. In addition, the fine-grain decomposition with load balancing is unable to exploit more than three processors; the extensive bus traffic generated by having every thread load a row of the matrix into the local cache during its short lifetime limits the number of processors that can be used effectively. Memory-conscious scheduling eliminates most of this bus traffic however, so much so that the performance of the fine-grain decomposition under memory-conscious scheduling is comparable to that of the coarse-grain decomposition. The only difference between the two is the cost of creating 200,000 threads, or about one second.

Figure 4 plots the results for Grassfire. Once again, fine-grain threads under memory-conscious scheduling are comparable to a coarse-grain decomposition. The improvement over the load balancing policy is not quite as dramatic as in the earlier example, but is still substantial. Similar results for merge sort are shown in Figure 5.

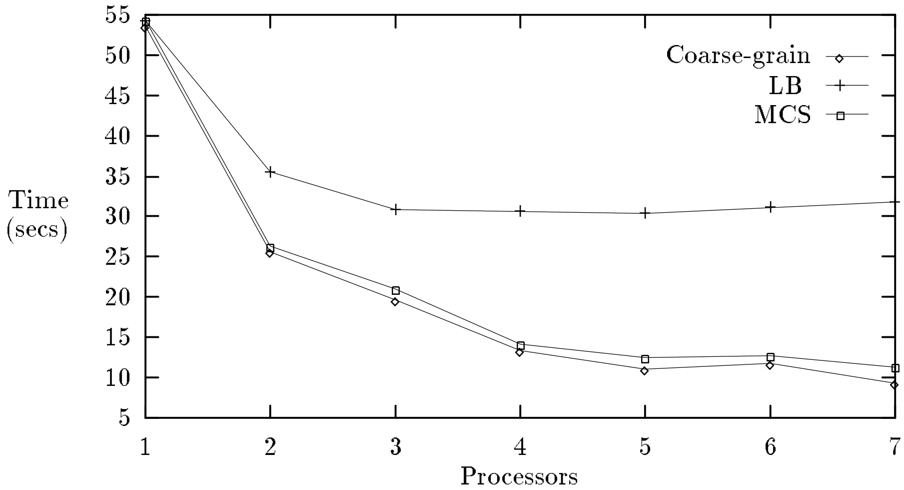

Figure 3: Gaussian elimination of a 640 by 640 matrix

We can draw several conclusions from these examples. First, fine-grain threads under traditional scheduling policies perform much worse than coarse-grain threads, not because of the high cost of thread management, but rather because of the cost of repeatedly loading data into the local cache. Second, memory-conscious scheduling alleviates most of this performance disparity by scheduling a thread on the processor containing the data it needs. Third, the benefits of memory-conscious scheduling depend on the application, and in some cases, on the number of processors.

The performance benefits of memory-conscious scheduling vary across our applications because each of the applications exhibits a different degree of data sharing among threads. In Gaussian elimination, each thread modifies a single row of the matrix based on the contents of the pivot row, which need only be loaded into each cache once. In merge sort, each thread processes two sorted lists produced by two other threads. In Grassfire, each thread modifies a single row of the image matrix based on the contents of two boundary rows. By scheduling a thread on the processor containing the data to be modified, memory-conscious scheduling eliminates a third of the memory traffic in Grassfire, half the memory traffic in merge sort, and nearly all the memory traffic in Gaussian elimination. The performance results in Figures 3-5 are consistent with these observations.

Figure 3 suggests that the relative benefits of memory-conscious scheduling depend in part on the number of processors used during execution. There are several reasons for this:

- With a small number of processors, there is a good chance that a random placement of threads produces the desired result of having threads run on the processors containing their data. For

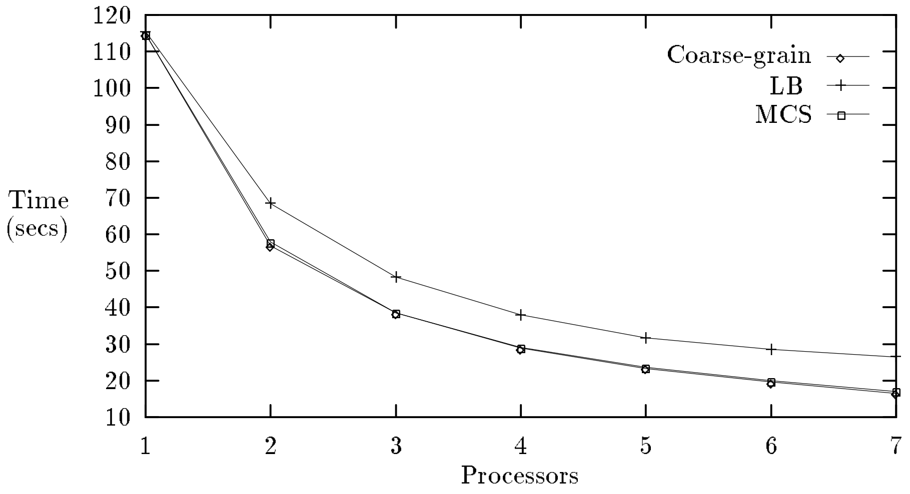

Figure 4: Grassfire on a 512 by 512 matrix

example, with 2 processors there is a 50% chance that a random placement policy produces the best placement for a particular thread.

- As the number of processors increases, so does bus contention, which slows down every main memory access. By reducing the need for main memory accesses, memory-conscious scheduling reduces contention, which improves the speed of any remaining accesses. This effect is particularly important on bus-based multiprocessors such as the Iris, where bus contention is often a problem.
- As the number of processors increases, so does the total amount of available cache (or local memory) space. Extra cache space decreases the likelihood that data will be ejected from a cache, which means that a thread's data will almost always reside in some cache.

To illustrate these points, we plotted the relative performance improvement of memory-conscious scheduling over the traditional load balancing policy for each of our applications as a function of the number of processors. The results appear in Figure 6.

As can be seen in Figure 6, the performance benefits of memory-conscious scheduling increase with the number of processors, although the precise improvement depends on the application. For example, the percentage improvement of Grassfire rises slowly, but steadily, with an increase in the number of processors. The improvement of merge sort rises very quickly up to 4 processors, but then remains constant. Gaussian elimination exhibits dramatic improvements up to 4 processors, and then slow, steady improvements thereafter.

Gaussian elimination exhibits a jump in improvement between 3 and 4 processors because the matrix used in our experiments doesn't fit in three caches, but does fit in four.[‡] Thus, there are no cache evictions on 4 processors; once the caches contain the entire matrix, memory-conscious scheduling reduces the need for any main memory accesses other than those caused by write-sharing.

Performance improvements are still possible even when the matrix does not fit in the local caches. As long as parts of the matrix reside in the caches long enough to be used by more than one thread, some main memory accesses are avoided. In Gaussian elimination, the portion of the matrix that

[‡]The processors on the Iris have a 64KB first-level cache and a 1MB second-level cache.

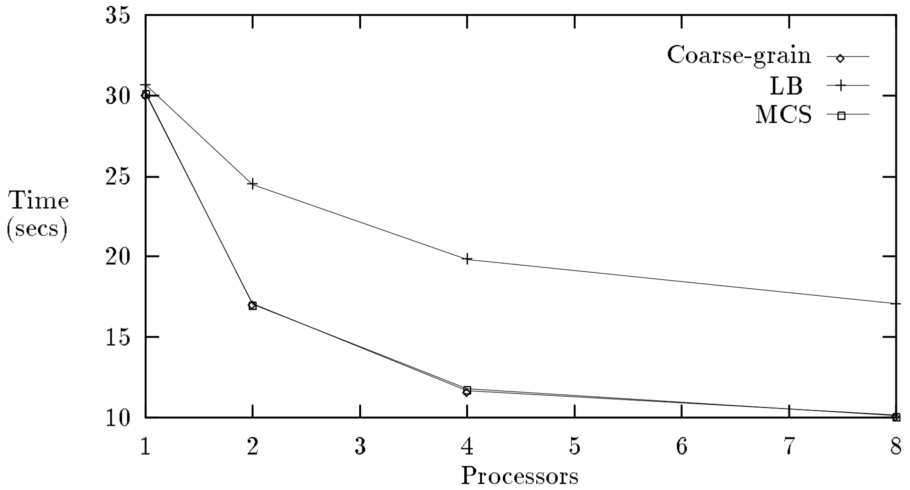

Figure 5: Merge sort of 2 million integers

needs to be stored in the caches shrinks as the matrix becomes triangular, and the computation is centered on higher numbered rows. During the latter stages of the execution, the data required by the threads will fit in the local caches, even if the original matrix did not.

The image matrix used in Grassfire fits in two caches, so we do not see much improvement as we increase the number of processors from 2 to 4. Even though the input to merge sort fits in eight caches (but not four), we do not see an improvement in the relative benefits of memory-conscious scheduling when we move from 4 to 8 processors; the form of the divide-and-conquer algorithm is such that most of the benefits of locality management come from the low levels in the tree, which do not require all the data to be resident in the caches.

3.2.3 Comparison of Butterfly Plus and Iris Results

It is somewhat surprising that memory-conscious scheduling improves performance by 20-30% on the Butterfly Plus, and by 41-64% on the Iris. We would expect any locality management technique, including memory-conscious scheduling, to be more effective on distributed share-memory machines like the Butterfly Plus than on bus-based, cache-coherent multiprocessors like the Iris. It is obvious from our experiments that the lack of hardware coherency does not by itself dictate the need for locality management.

The processors on the Iris are about 12 times faster than the processors on the Butterfly Plus. However, a cache miss on the Iris is only about 9 times faster than a remote memory access on the Butterfly Plus (in the absence of contention). Therefore, all other things being equal, cache misses are a greater percentage of the execution time of a program on the Iris than are remote memory references on the Butterfly. Since memory-conscious scheduling reduces the number of cache misses and remote memory references by roughly the same amount, it has a greater effect on the Iris than on the Butterfly. As processor speeds continue to improve relative to memory speeds, locality management techniques such as memory-conscious scheduling will provide even greater benefits on future machines.

There is another reason why memory-conscious scheduling is so effective on the Iris. The Iris suffers from bus contention when more than a few processors make frequent references to main memory. Any reduction in main memory references on the Iris due to memory-conscious scheduling will reduce bus contention for the remaining references, further improving performance. The Butterfly's communication network, on the other hand, is relatively immune to contention, so a reduction in the

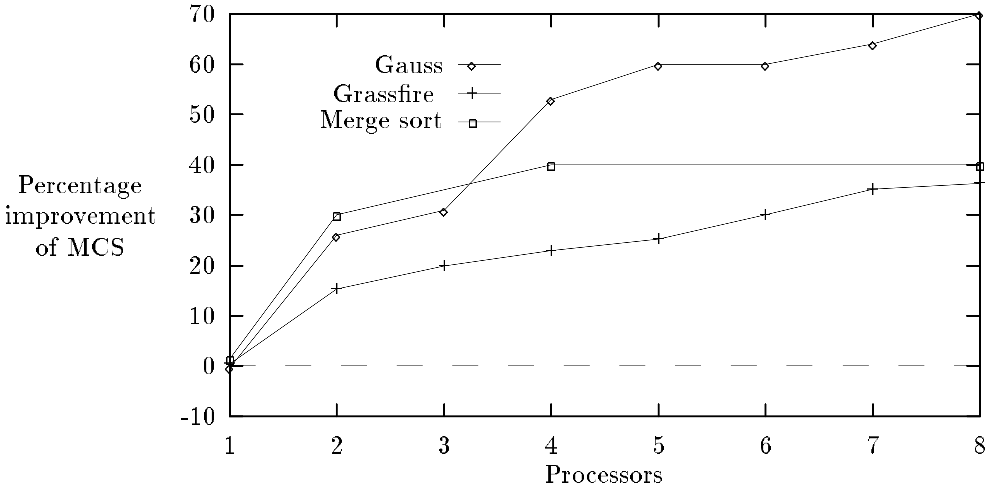

Figure 6: The effect of number of processors on memory-conscious scheduling.

number of remote references isn't likely to have much effect on the remaining remote references.

Based on our experiments, we conclude that modern multiprocessors cannot efficiently support lightweight threads unless memory-conscious scheduling is used. If the thread scheduler does not preserve affinity, a coarse-grain decomposition will almost always out-perform a fine-grain decomposition, in some cases by a factor of 3 or more. This discrepancy cannot be attributed to thread management operations, but is instead due to the excessive bus or network traffic associated with the fine-grain decomposition. Memory-conscious scheduling removes much of the network traffic associated with the placement of fine-grain threads; the remaining minor difference in performance is due to the cost of creating and managing a large number of threads.

4 Loop Scheduling

Although thread libraries provide a simple way to program parallel applications, many users prefer to leave the work of decomposition, parallelization, and scheduling to the compiler. This is especially true in the case of large sequential applications that have already been programmed in a sequential language. Parallelizing these applications by hand is a tedious and time-consuming process. Thus, parallelizing compilers are left with the task of parallelizing the application. The most prominent source of parallelism in most sequential applications are loops. The unit of work that can be executed in parallel is usually a loop iteration. The task of the compiler is to find which iterations can be executed in parallel. One task of the run-time environment of a parallelizing compiler is to schedule parallel iterations on processors. This task is called loop scheduling.

The simple *static scheduling* algorithm divides the number of loop iterations among the available processors as evenly as possible, in the hope that each processor receives about the same amount of work. This algorithm minimizes run-time synchronization overhead, but does not balance the load dynamically.

The simplest dynamic algorithm for scheduling loop iterations is called *self-scheduling* [Smith, 1981; Tang and Yew, 1986]. In this algorithm, each processor repeatedly executes one iteration of the loop until all iterations are executed. The algorithm relies on a central work queue of iterations, where each idle processor gets one iteration, executes it, and repeats the same cycle until there are no more iterations to execute. Self-scheduling achieves almost perfect load balancing, since all processors finish within one iteration of each other. Unfortunately, this algorithm incurs significant synchronization overhead; each iteration requires atomic access to the central work queue.

Uniform-sized chunking [Kruskal and Weiss, 1985] reduces synchronization overhead by having each processor take K iterations, instead of one. This algorithm amortizes the cost of each synchronization operation over the execution time of K iterations, resulting in less synchronization overhead. Uniform-sized chunking has a greater potential for imbalance than self-scheduling however, as processors finish within K iterations of each other in the worst case.

Guided self-scheduling [Polychronopoulos and Kuck, 1987] is a dynamic algorithm that changes the size of chunks at run-time, allocating large chunks of iterations at the beginning of a loop so as to reduce synchronization overhead, while allocating small chunks towards the end of the loop to balance the workload. Under guided-self scheduling each processor is allocated $1/P_{th}$ of the remaining loop iterations, where P is the number of processors. Assuming all loop iterations take the same amount of time to complete, guided-self scheduling ensures that all processors finish within one iteration of each other and use the minimal number of synchronization operations.

In some cases guided-self scheduling might assign too much work to the first few processors, so that the remaining iterations are not sufficiently time-consuming to balance the workload. This situation arises when the initial iterations of a loop are much more time-consuming than later iterations. The *factoring* algorithm [Hummel *et al.*, 1992] addresses this problem. Under factoring, allocation of loop iterations to processors proceeds in phases. During each phase, only a subset of the remaining loop iterations (usually half) is divided equally among the available processors. Because factoring allocates a subset of the remaining iterations in each phase, it balances load better than guided-self scheduling when the computation times of loop iterations vary substantially. In addition, the synchronization overhead of factoring is not significantly greater than that of guided-self scheduling.

Although guided-self scheduling minimizes the number of synchronization operations needed to achieve perfect load balancing, the overhead of synchronization can become significant in large-scale systems with very expensive synchronization primitives. *Trapezoid self-scheduling* [Tzen and Ni, 1993] tries to reduce the need for synchronization, while still maintaining a reasonable balance in load. This algorithm allocates large chunks of iterations to the first few processors, and successively smaller chunks to the last few processors. The first chunk is of size $\frac{N}{2P}$, and consecutive chunks differ in size $\frac{N}{8P^2}$ iterations. The difference in the size of successive chunks is always a constant in trapezoid self-scheduling, whereas it is a decreasing function both in guided-self scheduling and in factoring.

All of these loop scheduling algorithms attempt to balance the workload among the processors without incurring substantial synchronization overhead. Each of the algorithms assumes that an individual iteration takes the same amount of time to execute on every processor. This assumption is not valid however on many shared-memory multiprocessors. The existence of memory that is not equidistant from all processors (such as local memory or a processor cache) implies that some processors are closer to the data required by an iteration than others. Loop iterations frequently have an *affinity* [Squillante and Lazowska, 1990] for a particular processor — the one whose local memory or cache contains the required data. By exploiting processor affinity, we can reduce the amount of communication required to execute a parallel loop, and thereby improve performance.

4.1 Affinity Loop Scheduling

Affinity scheduling is based on the assumption that, in many cases, loop iterations have an affinity for a particular processor. In order for this assumption to hold, it must be the case that: (1) the same data is used over and over by an iteration, and (2) the data is not removed from the local memory (or cache) before it can be reused.

Data reuse is common in many applications, particularly those that employ iterative algorithms wherein a parallel loop is nested within a sequential loop. In such cases, each iteration of the parallel loop accesses the same (or nearby) data on successive iterations of the enclosing sequential loop. During the first iteration of the sequential loop, each iteration of the nested parallel loop loads the required data into the local memory or cache, where it may remain during subsequent iterations of the enclosing sequential loop.

Data reuse may also occur in programs produced by a parallelizing compiler. Earlier work has suggested that nested loops be interchanged in such a way as to reduce synchronization and communication overhead [Gupta, 1989]. The resulting loop structure nests a parallel loop within a sequential loop, again producing the desired form. If necessary, several parallel loops can be coalesced into one [Polychronopolous, 1988].

We consider the loop scheduling problem to have three dimensions: load imbalance, synchronization overhead, and communication overhead due to non-local memory accesses. Our algorithm for affinity scheduling builds on previous work in loop scheduling, while also attempting to exploit processor affinity. The main ideas underlying the algorithm are:

- As with many known algorithms, we assign large chunks of iterations at the start of loop execution, so as to reduce the need for synchronization, and assign progressively smaller chunks to balance the load.

- We use a deterministic assignment policy to ensure that an iteration is always assigned to the same processor. After the first execution of the iteration, that processor will contain the required data, so subsequent executions of the iteration will not need to load the data into local storage.

- We reassign a chunk to another processor (which also involves moving the required data) only if necessary to balance the load. An idle processor removes chunks from another's queue, and executes them indivisibly, so an iteration is never reassigned more than once.

We assume that the underlying hardware or software implements a coherent memory, so that data is copied into local storage when first accessed. This copy is implemented in hardware on machines with coherent caches, such as the Symmetry, the Silicon Graphics machine, and the Kendall Square Research multiprocessor, and may be implemented in the operating system on machines lacking coherent caches, like the Butterfly [Bolosky *et al.*, 1989; Cox and Fowler, 1989; LaRowe, Jr. and Ellis, 1991].

Our affinity scheduling algorithm divides the iterations of a loop into chunks of size $\lceil N/P \rceil$, where N is the number of iterations in the loop, and P is the number of available processors. The i_{th} chunk of iterations is always placed on the local work queue of processor i. When a processor is idle, it removes $1/k$ of the iterations in its local work queue and executes them. [§] If a processor's work queue is empty, it finds the most loaded processor, removes $\lceil 1/P \rceil$ of the iterations in that processor's work queue, and executes them. [¶]

Note that we distinguish between assigning a loop iteration to a processor's work queue, and executing the iteration on that processor. Initially, loop iterations are assigned to a processor's work queue in chunks of size $1/P$, so as to balance the load statically. Processors execute $1/k$ of the remaining iterations on their local work queue at a time, which corresponds to at most N/kP iterations. Processors execute $1/P$ of the remaining iterations from a remote work queue, which corresponds to at most N/P^2 iterations.

Figure 7 contains a pseudocode description for the affinity scheduling algorithm. Although we implemented this algorithm by hand for our experiments, it could easily be incorporated into a parallelizing compiler.

There are two important differences between affinity scheduling and previous dynamic loop scheduling algorithms. First, the initial assignment of chunks to processors in affinity scheduling is deterministic. That is, processor i is always assigned the i_{th} chunk of iterations to execute. For many programs, this assignment ensures that repeated executions of the loop will access data that is already stored in the local memory or cache. Second, affinity scheduling initially assumes that load imbalance will not occur, and therefore assigns the same number of iterations to each processor's work queue. Each processor gets iterations from its own local work queue; accesses to different work queues can proceed in parallel, and each access is local, and therefore cheap. If load imbalance arises, the algorithm migrates iterations from loaded processors to idle ones. Migrating iterations causes the

[§] The constant k is a parameter of our algorithm. In most of our experiments we assume k equals P.

[¶] Synchronization is required to remove iterations from a work queue, but not to check the load on a processor.

associated data to move twice in most cases; the data must first move to an idle processor to alleviate load imbalance, and then move back to its original location to restore processor affinity. However, under affinity scheduling this overhead is introduced only when load imbalance arises, whereas other algorithms incur this overhead on every scheduling decision.

```
loop_initialization(N,P)
// N is the number of loop iterations, P is the number of processors
{
    for(i = 0 ; i < P ; i++) {
        // assign iterations ceil(i*N/P) to min(N,ceil((i+1)*N/P))
        // to processor i
        assign_iterations(i)
    }
}

loop // executed by each processor
    // get 1/k of the local iterations to execute
    range = get_iterations_from_local_queue(1/k) ;
    if (range == empty)
        max_load = find_most_loaded-processor()  ;
        if (max_load == nil) break ;
        // get 1/P of the iterations from the most loaded processor
        range = get_iterations_from_nonlocal_queue(max_load,1/P);
        if (range == nil) break ;
    execute(range) ;
forever
```

Figure 7: Pseudocode for Affinity Scheduling

Despite these differences, affinity scheduling has all the advantages of the best dynamic loop scheduling algorithms. That is, it balances the load dynamically, minimizes synchronization, and is immune to the arrival and departure of processors in the system.

4.2 Experimental Evaluation

We implemented the following loop scheduling algorithms by hand on the Silicon Graphics Iris: static scheduling (STATIC), self-scheduling (SS), guided-self scheduling (GSS), factoring (FACTORING), trapezoid self-scheduling (TRAPEZOID), affinity scheduling with $k = P$ (AFS), and a hand-optimized algorithm (BEST-STATIC). BEST-STATIC represents our attempt at the best static assignment possible, given complete knowledge of the application and its input. We implemented this assignment by hand, after examining the application and the input, so as to maximize locality of reference and minimize load imbalance. While not generally realizable, since it requires programmer intervention and assumes knowledge of the application's input, BEST-STATIC is a useful base-line for evaluating other loop scheduling algorithms.

We selected five application programs that present loop scheduling algorithms a range of opportunities for addressing load imbalance, synchronization overhead, and communication overhead. Our application suite contains the following programs:

- Successive Over-Relaxation (SOR):

```
DO SEQUENTIAL 19 I = 1,MAXITERATIONS
    DO PARALLEL 29 J = 1,N
        DO SEQUENTIAL 39 K = 1,N
           A(J,K) = UPDATE(A,J,K)
```

```
        39 CONTINUE
    29 CONTINUE
19 CONTINUE
```

All iterations of the parallel loop take about the same time to execute, so better load balancing algorithms are not likely to produce much better performance. However, the i_{th} iteration of the parallel loop always accesses the i_{th} row of the matrix, so scheduling algorithms that exploit processor affinity are likely to produce better performance.

- Gaussian Elimination:

```
DO SEQUENTIAL 19 K = 2,N
    DO PARALLEL 29 I = K,N
        DO SEQUENTIAL 39 J = K-1,N+1
            A[I][J] -= A[K-1][J] * A[i][K-1]/A[K-1][K-1]
        39 CONTINUE
    29 CONTINUE
19 CONTINUE
```

This application exhibits some load imbalance across iterations, and offers some opportunities for exploiting processor affinity. Although successive executions of an iteration of the parallel loop do not access exactly the same matrix elements each time, there is significant overlap in the elements referenced by successive executions of an iteration.

- Transitive Closure:

```
DO SEQUENTIAL 19 K = 1,N
    DO PARALLEL 29 J = 1,N
        IF (A(J,K) .EQ. TRUE)  THEN
            DO SEQUENTIAL 39 I = 1,N
                IF (A(K,I) .EQ. TRUE) A(J,I) = TRUE
            39 CONTINUE
    29 CONTINUE
19 CONTINUE
```

The distinguishing characteristic of this application is that each iteration of the parallel loop may take time $O(1)$ or $O(N)$ (where the input matrix is of size $N \times N$), depending on the input data. Since the input values determine the variation in iteration execution time, this application serves to evaluate the effectiveness of load balancing for each scheduling algorithm. This application also benefits from some form of affinity scheduling, since the i_{th} iteration of the parallel loop always accesses the i_{th} row of the matrix.

- Adjoint Convolution:

```
DO PARALLEL 19 I = 1,N*N
    DO SEQUENTIAL 29 K = I,N*N
        A(I) = A(I) + X*B(K)*C(I-K)
    29  CONTINUE
19  CONTINUE
```

This application exhibits significant load imbalance; the i_{th} iteration of the parallel loop takes time proportional to $O(n^2 - i)$. There is no affinity to exploit however, so this application serves to evaluate the effectiveness of load balancing in the absence of affinity.

Table 5 summarizes the properties of our application suite with respect to load imbalance and affinity. If an application exhibits load imbalance, the iterations of the loop may take varying amounts of computation time, so a static scheduling algorithm may not be appropriate. If an application exhibits affinity, we can improve performance by scheduling iterations appropriately.

Application	Load imbalance	Affinity
SOR	none	yes
Gauss elimination	little	yes
Transitive closure	input dependent	yes
Adjoint convolution	large	no

Table 5: Load imbalance and affinity characteristics of the application suite.

4.3 Comparison of Loop Scheduling Algorithms

In this section we compare the performance of the various loop scheduling algorithms on the SGI Iris.

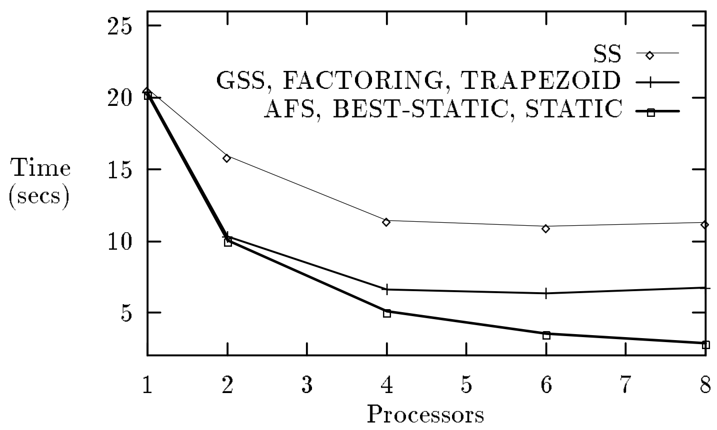

Figure 8: Performance of loop scheduling algorithms for SOR.

Figure 8 presents the execution time (in seconds) of SOR ($N = 512$) running on 1 to 8 processors. As can be seen in the figure, self-scheduling (SS) performs the worst of all, due to its high synchronization overhead. Other algorithms with lower synchronization overhead, such as GSS, FACTORING, and TRAPEZOID, perform much better than SS because the small chunk size used by SS is of no benefit for an application in which there is no significant difference in the execution time of iterations. All of these algorithms perform worse than the algorithms that exploit affinity. Both STATIC and AFS are comparable to the best possible static algorithm. These results confirm that affinity scheduling can improve the performance of loop scheduling algorithms.

Figure 9 plots the execution time of Gaussian elimination ($N = 768$) under the different scheduling algorithms. It is surprising to see that none of the scheduling algorithms that ignore processor affinity can effectively utilize more than two processors. There is simply too much contention for the shared bus under these algorithms, since every iteration must load data into the local cache. SS performs worst of all, because of its high synchronization overhead, but the performance difference narrows quickly as the communication costs of GSS, FACTORING, and TRAPEZOID start to dominate synchronization

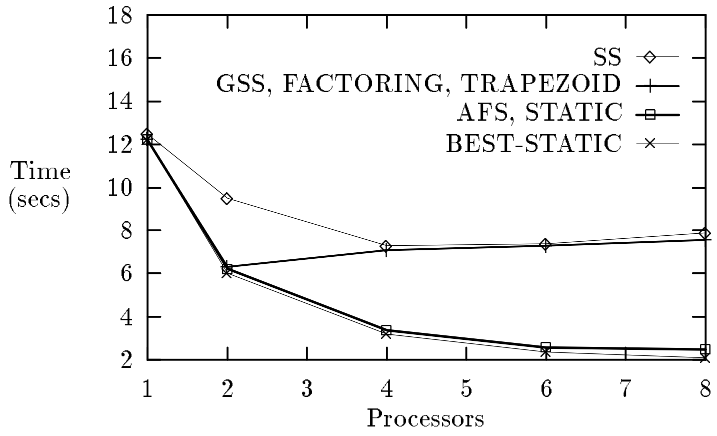

Figure 9: Performance of loop scheduling algorithms for Gaussian elimination.

costs. Once again, AFS and STATIC perform the best; they are very close to BEST-STATIC in the worst case, and a factor of 3 better than the other dynamic loop scheduling algorithms. AFS and STATIC can effectively use all 8 processors.

This application is a perfect example of the fact that the dominant source of overhead in many applications is communication (caused by cache misses), not synchronization. Loop scheduling algorithms that focus on synchronization overhead alone perform poorly when compared to algorithms that reduce communication overhead by exploiting processor affinity.

Figure 10 presents the completion time of the transitive closure application when given a skewed input graph of 640 nodes containing a clique of 320 nodes, and no other edges. This is the first example

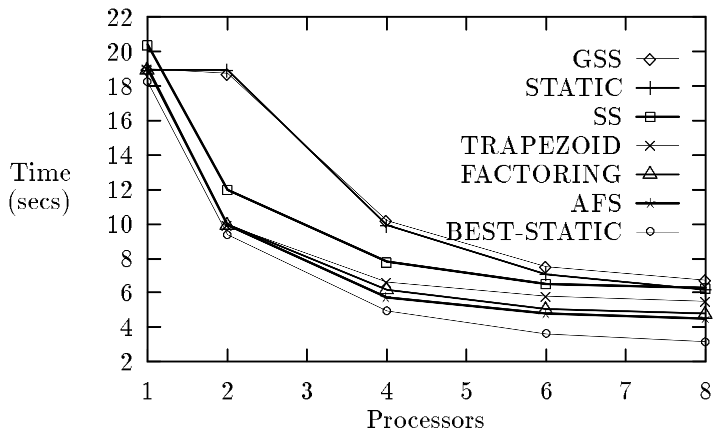

Figure 10: Performance of loop scheduling algorithms for transitive closure (skewed input).

where there is significant imbalance in the computation across iterations, which explains why STATIC performs poorly. Although SS manages to balance the load, it still suffers from high synchronization overhead. The surprising result in Figure 10 is that GSS performs worst of all. Although GSS assigns only $1/P_{th}$ of the iterations to the first processor, those iterations contain $2/P_{th}$ of the total work; the remaining iterations do not contain enough work to balance the load. Both FACTORING and TRAPEZOID start with a smaller initial chunk of iterations, and therefore balance the load better. AFS has the same load balancing properties as FACTORING and TRAPEZOID, but exploits affinity as well.

Although AFS performs the best, the improvement over FACTORING and TRAPEZOID is at most 15%. The existence of significant load imbalance forces an affinity scheduler to override the initial assignment of iterations to processors and instead execute iterations on any available idle processor. Each time an iteration moves to another processor, the data must be loaded into a different cache. This is also why AFS does not perform as well as BEST-STATIC, which has knowledge of the input, and is therefore able to distribute the clique nodes evenly among the processors, while maintaining processor affinity.

Figure 11 presents the performance of the scheduling algorithms for adjoint convolution with $N = 75$. Iterations have no affinity for a particular processor in this application, since the parallel

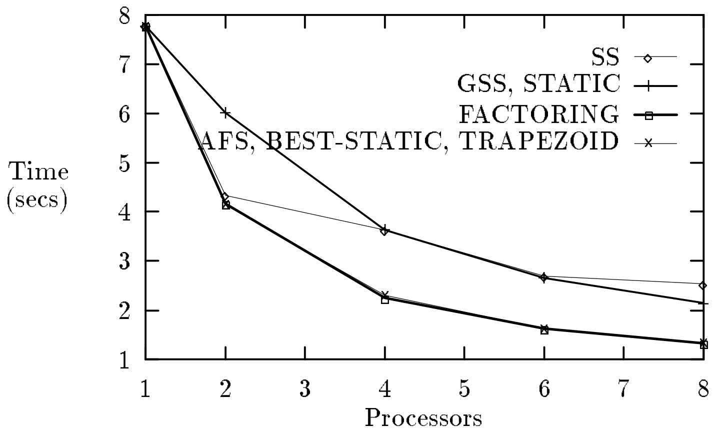

Figure 11: Performance of loop scheduling algorithms for adjoint convolution.

loop is not embedded within a sequential loop. There is significant load imbalance across iterations however, since the first iteration takes time proportional to $O(N^2)$, while the last iteration takes time proportional to $O(1)$. As expected, loop scheduling algorithms that emphasize load balancing, such as FACTORING, TRAPEZOID and AFS, perform the best. GSS and the static methods assign too much work to the first few processors and suffer load imbalance as a result. SS again suffers from high synchronization overhead. These results are consistent with those reported in [Hummel *et al.*, 1992].

We should note that a trivial change to our implementation of GSS would improve its performance to be comparable to FACTORING, although not as good as AFS for these examples. Instead of taking $\lceil N/P \rceil$ iterations, each processor could take $\lceil N/(kP) \rceil$ iterations, where k is an appropriate constant. With this change, GSS could start with smaller chunks, leaving more opportunities to balance the load without introducing significant synchronization overhead.

4.4 Increasing the Cost of Communication

Our experiments on the Iris confirm that communication overhead is a dominant factor in application performance on modern shared-memory multiprocessors. Why then do so many loop scheduling algorithms ignore communication overhead? The answer lies in the changes in hardware that have occurred over the last few years. RISC technology and floating point co-processors have increased the speed of computation dramatically, while memory and interconnection network speeds have improved only modestly.

These trends in multiprocessor architecture shift the emphasis from computation costs to communication costs, and suggest a much greater need for scheduling algorithms that reduce communication. In order to demonstrate this trend, we executed our Gaussian elimination program on a Sequent Symmetry S81 multiprocessor, a bus-based, cache-coherent machine that predates the Iris. The processors on the Iris are about 30 times faster than the processors on the Symmetry, but the peak bandwidth of the Symmetry bus is 80 MB/sec, while the peak bandwidth of the Iris bus is only 64 MB/sec. Figure 12 plots the execution time of Gaussian elimination on a 256 by 256 matrix under three dynamic loop scheduling algorithms on the Symmetry. From this figure we can see that AFS and GSS are comparable in performance on the Symmetry, while our earlier results showed that AFS clearly dominates GSS on the Iris. We can conclude that the ability of AFS to exploit processor affinity in Gaussian elimination is of little value on the Symmetry, since communication is cheap relative to computation.

We also see in figure 12 that TRAPEZOID performs 10-15% worse than both AFS and GSS on this application. The reason for this can be traced to the load balancing properties of TRAPEZOID. When all iterations take the same time to execute, processors finish within one iteration of each other under guided-self scheduling [Polychronopoulos and Kuck, 1987]. Under TRAPEZOID, processors finish within several iterations of each other [Tzen and Ni, 1993]. When an iteration takes a long time to complete, the imbalance introduced by the trapezoid algorithm can be noticeable. Although the trapezoid algorithm requires fewer accesses to the work queue, the Sequent does not employ a large number of processors, and therefore the low synchronization overhead of TRAPEZOID does not outweigh the load imbalance it causes.

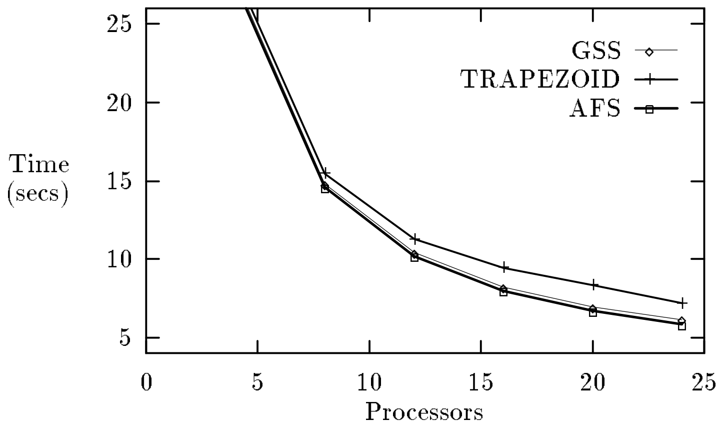

Figure 12: Gaussian elimination on the Sequent Symmetry.

These results suggest that communication was a relatively minor source of overhead on the previous generation of shared-memory multiprocessors, and that both load imbalance and synchronization overhead were dominant. Our results on the Iris suggest that the situation has changed dramatically, so much so that communication is now the dominant factor in performance. As processor speeds continue to improve at a higher rate than either memory or interconnection network speeds, the overhead of communication will likely increase even more. In fact, the discrepancy in processor and memory speeds is already high enough that a breakthrough in memory and interconnection network technology *without continuing improvements in processor technology* would be required to reduce the significance of communication in parallel applications. In short, communication is a dominant factor in modern multiprocessors and there is no indication that the situation will change in the foreseeable future. Any scheme designed to reduce communication overhead, such as affinity scheduling, will produce ever-greater returns as long as current trends continue.

4.5 Scaling the Number of Processors

To demonstrate the importance of affinity scheduling on recent large-scale multiprocessors, we performed several experiments on the Kendall Square KSR-1, a large-scale, cache-coherent multiprocessor released in 1992. These experiments used only those applications that exhibit locality, i.e., Gaussian elimination, SOR, and transitive closure.

Figure 13 presents the completion time of Gaussian elimination (using a 1024 by 1024 matrix) under various loop scheduling algorithms on the KSR-1. In this figure we see that, once again, AFS performs best. It improves the completion time of the application by a factor of 3.7 when compared to FACTORING and GSS, and by a factor of 2.8 compared to TRAPEZOID. The reason that TRAPEZOID performs better than FACTORING and GSS is that TRAPEZOID has the fewest number of synchronization operations, and synchronization is relatively expensive on the KSR.

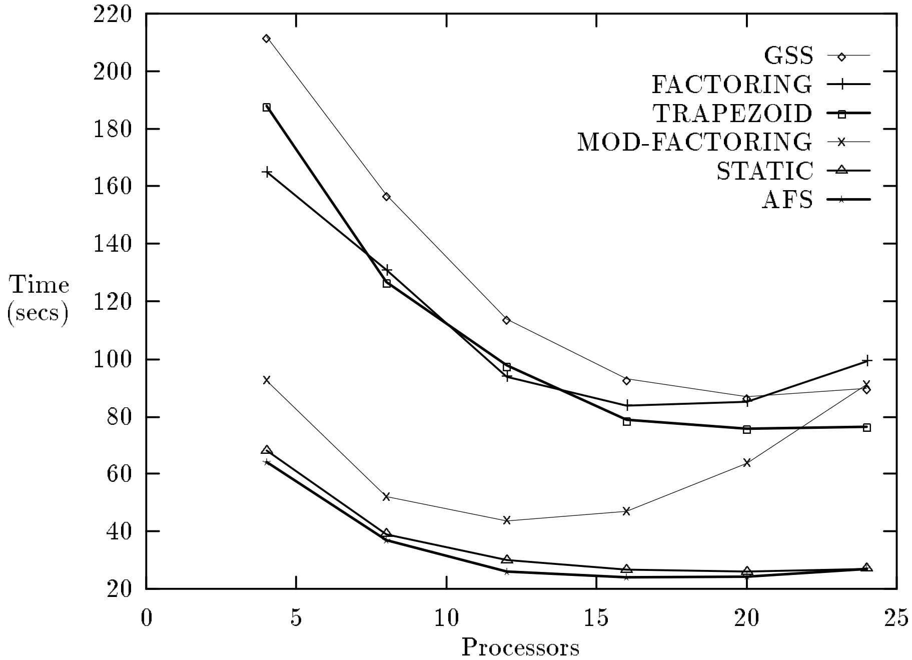

Figure 13: Gaussian elimination on the KSR-1.

Figure 14 shows the completion time of transitive closure (1024 node graph, where 40% of the nodes form a clique) under the different loop scheduling algorithms on the KSR-1. From this figure we can see the importance of affinity scheduling; the other dynamic scheduling algorithms cannot exploit more than 12 processors. After AFS, the next best algorithm is TRAPEZOID, which has the smallest number of synchronization operations, and therefore manages to degrade more gracefully than the other algorithms. Although AFS performs best, the improvement over the other algorithms is not as great as it was for Gaussian elimination. There is almost no load imbalance in Gaussian elimination, and hence no need to destroy any affinity, whereas transitive closure does have load imbalance and therefore affinity scheduling must reassign iterations fairly frequently.

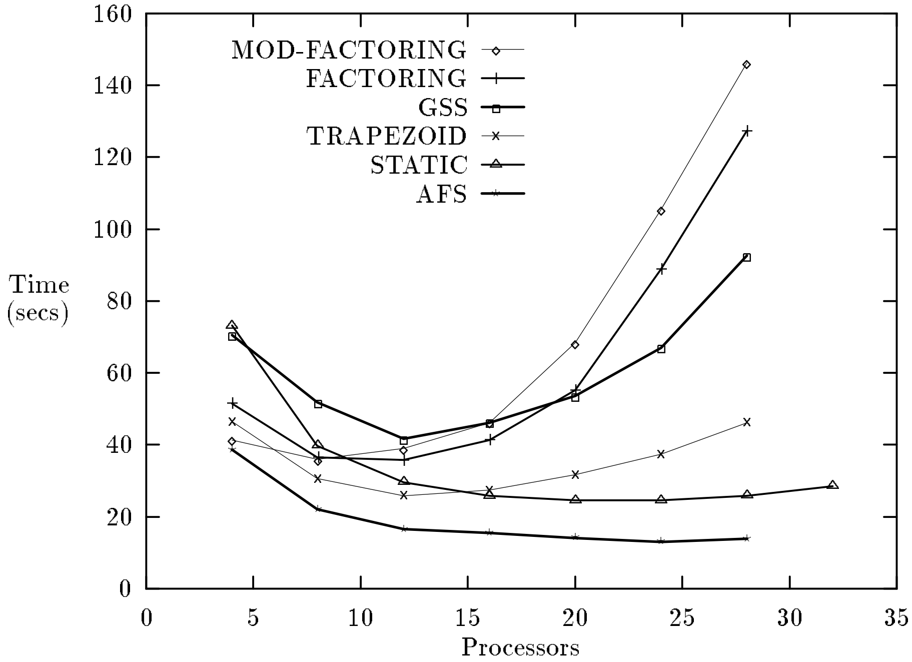

Figure 14: Transitive closure on the KSR-1.

Figure 15 presents the completion time of SOR (1024 by 1024 matrix and 128 iterations) on the KSR-1. Although AFS and STATIC perform the best in this case, they are not much better than the other algorithms, even though SOR has a lot of affinity to preserve, and there is almost no load imbalance to hinder affinity. So why isn't AFS much better than the other algorithms? The reason for this anomaly is that SOR performs a few floating point additions and one floating point division within the inner loop, and floating point division is implemented in software on the KSR-1. Thus, computation in SOR is expensive on the KSR-1, and the benefits of preserving affinity are not significant in comparison.

5 Conclusions

Most scheduling schemes in use today focus on the role of synchronization and load imbalance in application performance. The increasing cost of communication in shared-memory multiprocessors argues for new scheduling policies that reduce or eliminate communication. We have shown that the operating system kernel, the thread library, and the compiler all have a role to play in reducing the need for communication. Based on our experiments with a variety of shared-memory machines and applications we conclude:

- *In most cases, multiprogramming via hardware partitioning offers the best application performance.* In comparison to hardware partitions, time-sharing across the entire machine destroys the affinity a process may have built up with a processor, and significantly increases overhead due to synchronization. Coscheduling alleviates the overhead due to synchronization, but results in increased cache corruption and more remote references when compared with hardware partitions. In addition, coscheduling is rarely 100% effective, as most parallel programs are unable to fully utilize all processors for the lifetime of the program.

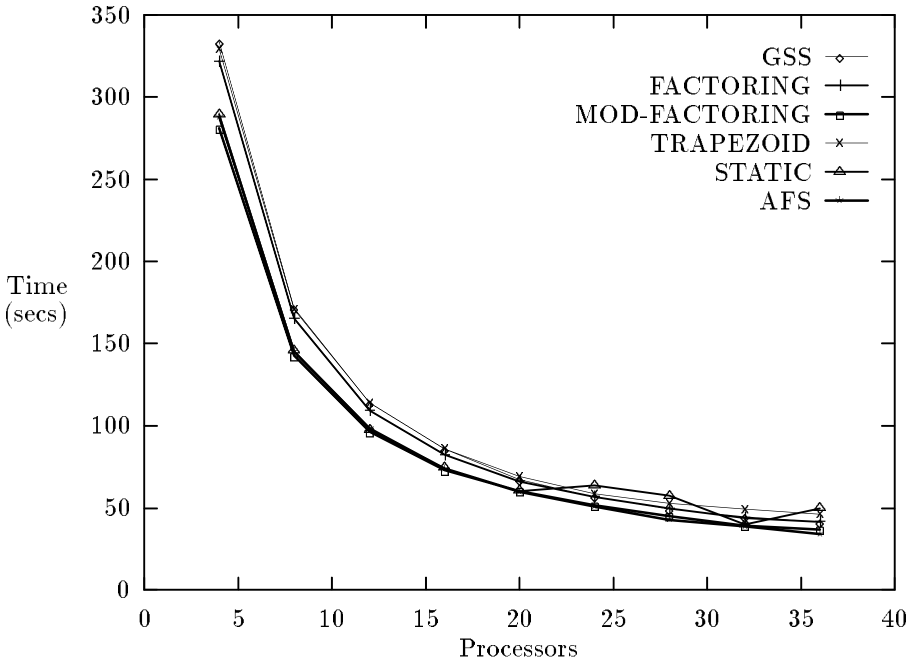

Figure 15: SOR on the KSR-1.

- *At any level in the system, central work queues are an inappropriate scheduling mechanism, even for small-scale multiprocessors.* Central work queues require the frequent movement of data among processors, since every process must load the data it needs into the local cache. The resulting communication overhead degrades performance even for a very small number of processors.
- *Both thread and loop scheduling algorithms must consider communication as an important source of overhead.* Algorithms that ignore communication overhead incur a significant performance penalty on modern multiprocessors. If processor speeds continue to improve more quickly than memory or interconnection speeds, communication will be an increasing percentage of an application's execution time; scheduling methods that reduce both communication and synchronization overhead are going to have an even greater impact in the future.

If current trends continue, it will be increasingly difficult for parallel applications to utilize large-scale multiprocessors effectively. One way to address these problems is to recognize the dominant role of communication in current systems, and to adopt techniques for reducing communication in parallel programs. Cache architecture sensitive parallel application restructuring (CASPAR) [Cheriton *et al.*, 1991], latency-tolerant techniques [Agarwal *et al.*, 1990; Gupta *et al.*, 1991a], and the scheduling schemes discussed here are all steps in the right direction. These techniques will be even more important in the future if shared-memory machines are to be used efficiently for parallel programming.

Acknowledgements

The authors would like to thank Prakash Das, Mark Crovella and Cezary Dubnicki for providing many of the experimental results in section 2. We would like to thank Argonne National Laboratory for allowing us to use their Sequent Symmetry, and Donna Bergmark and the Cornell Theory Center for assistance with and use of their KSR-1. This work was supported by the National Science Foundation under grants CDA-8822724 and CCR-9005633, and the Office of Naval Research Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930).

References

[Agarwal *et al.*, 1990] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 104–114, May 1990.

[Anderson *et al.*, 1991a] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 53–79, October 1991.

[Anderson *et al.*, 1989] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors," *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[Anderson *et al.*, 1991b] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska, "The Interaction of Architecture and Operating System Design," In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[Bershad *et al.*, 1988] B.N. Bershad, E.D. Lazowska, H.M. Levy, and D.B. Wagner, "An Open Environment for Building Parallel Programming Systems," In *Proceedings of the ACM/SIGPLAN PPEALS 1988 Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 1–9, July 1988.

[Black, 1990] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *IEEE Computer*, 23(5):35–43, May 1990.

[Bokhari, 1987] S. H. Bokhari, *Assignment problems in parallel and distributed computing*, Kluwer Academic Publishers, Boston, 1987.

[Bolosky *et al.*, 1989] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott, "Simple But Effective Techniques for NUMA Memory Management," In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[Cheriton *et al.*, 1991] D. R. Cheriton, H. A. Goosen, and P. Machanick, "Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience," In *Proceedings of the International Symposium on Shared-Memory Multiprocessing*, pages 109–118, 1991.

[Cox and Fowler, 1989] A.L. Cox and R.J. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[Doeppner Jr., 1987] T. W. Doeppner Jr., "Threads: A System for the Support of Concurrent Programming," Technical Report CS-87-11, Department of Computer Science, Brown University, 1987.

[Gupta *et al.*, 1991a] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber, "Comparative Evaluation of Latency Reducing and Tolerating Techniques," In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254–263, May 1991.

[Gupta *et al.*, 1991b] A. Gupta, A. Tucker, and S. 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, May 1991.

[Gupta, 1989] R. Gupta, "Synchronization and Communication Costs of Loop Partitioning on Shared-Memory Multiprocessor Systems," In *Proceedings of the International Conference on Parallel Processing*, pages II:23–30, August 1989.

[Hummel *et al.*, 1992] S.F. Hummel, E. Schonberg, and L.E. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," *Communications of the ACM*, 35(8):90–101, August 1992.

[Kruskal and Weiss, 1985] C.P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.

[LaRowe, Jr. and Ellis, 1991] R. P. LaRowe, Jr. and C. S. Ellis, "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[Leutenegger, 1990] S. T. Leutenegger, *Issues in Multiprogrammed Multiprocessor Scheduling*, PhD thesis, University of Wisconsin-Madison, August 1990.

[Lo and Gligor, 1987b] S.-P. Lo and V.D. Gligor, "Properties of Multiprocessor Scheduling Algorithms," In *Proceedings of the International Conference on Parallel Processing*, August 1987.

[Lo and Gligor, 1987a] S.-P. Lo and V.D. Gligor, "A Comparative Analysis of Multiprocessor Scheduling Algorithms," In *Proceedings 7th International Conference on Distributed Computing Systems*, pages 205–222, September 1987.

[Marsh *et al.*, 1991] B.D. Marsh, M.L. Scott, T.J. LeBlanc, and E.P. Markatos, "First Class User-Level Threads," In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 110–121, October 1991.

[McCann *et al.*, 1993] C. McCann, R. Vaswani, and J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors," *ACM Transactions on Computer Systems*, 11(2), May 1993, Also published as Technical Report 90-03-02, University of Washington, March 1990 (Revised February 1991).

[Mellor-Crummey and Scott, 1991] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[Ousterhout, 1982] J. K. Ousterhout, "Scheduling Techniques for Concurrent Systems," In *Proceedings of Distributed Computing Systems*, pages 22–30, October 1982.

[Ousterhout, 1990] John Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?," *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[Polychronopolous, 1988] C. D. Polychronopolous, *Parallel Programming and Compilers*, Kluwer Academic Publishers, Boston, MA, 1988.

[Polychronopoulos and Kuck, 1987] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transactions on Computers*, C-36(12), December 1987.

[Scott *et al.*, 1990] M.L. Scott, T.J. LeBlanc, and B.D. Marsh, "Multi-Model Parallel Programming in Psyche," In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 70–78, March 1990.

[Smith, 1981] B. Smith, "Architecture and Applications of the HEP Computer System," In *Proceedings of the SPIE, Real-Time Signal Processing IV*, 1981.

[Squillante and Lazowska, 1990] M. S. Squillante and E.D. Lazowska, "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling," Technical Report 89-06-01, Computer Science Department, University of Washington, February 1990.

[Sun Microsystems, Inc., 1990] Sun Microsystems, Inc., "Lightweight Processes," In *SunOS Programming Utilities and Libraries*, March 1990, Sun Part Number 800-3847-10.

[Tang and Yew, 1986] P. Tang and P.-C. Yew, "Processor Self-Scheduling for Multiple Nested Parallel Loops," In *Proceedings 1986 International Conference on Parallel Processing*, pages 528–535, August 1986.

[Thomas and Crowther, 1988] R.H. Thomas and W. Crowther, "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors," In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 245–254, August 1988.

[Tucker and Gupta, 1989] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[Tzen and Ni, 1993] T.H. Tzen and L.M. Ni, "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers," *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[Vaswani and Zahorjan, 1991] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors," In *Proceedings of the Thirteenth Sumposium on Operating Systems Principles*, pages 26–40, October 1991.

[Weiser *et al.*, 1989] M. Weiser, A. Demers, and C. Hauser, "The Portable Common Runtime Approach to Interoperability," In *Proceedings of the 12th Sumposium on Operating Systems Principles*, pages 114–122, December 1989.

[Zahorjan and McCann, 1990] J. Zahorjan and C. McCann, "Processor Scheduling in Shared Memory Multiprocessors," In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–225, May 1990.