The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications

Anoop Gupta, **Andrew Tucker**, and **Shigeru Urushibara**
Computer Systems Laboratory
Stanford University, CA 94305

May 11, 1995

Abstract

Shared-memory multiprocessors are frequently used as compute servers with multiple parallel applications executing at the same time. In such environments, the efficiency of a parallel application can be significantly affected by the operating system scheduling policy. In this paper, we use detailed simulation studies to evaluate the performance of several possible scheduling strategies. These include regular priority scheduling with a single run-queue, coscheduling or gang scheduling, process control with processor partitioning, handoff scheduling, and affinity-based scheduling. We also explore the tradeoffs between the use of busy-waiting and blocking synchronization primitives and their interactions with the scheduling strategies. Since effective use of caches is essential to achieving high performance, a key focus in the paper is on the impact of the scheduling strategies on the caching behavior of the applications.

Our results show that in situations where the number of processes exceeds the number of processors, regular priority-based scheduling in conjunction with busy-waiting synchronization primitives results in extremely poor processor utilization. In such situations, use of blocking synchronization primitives can significantly improve performance. Process control and gang scheduling strategies offer the highest performance, and their performance is relatively independent of the synchronization method used. However, for applications that have sizable working sets that fit into the cache, process control performs better than gang scheduling. For the applications considered, the performance gains due to handoff scheduling and processor affinity are shown to be minor.

1 Introduction

Parallel application programmers often implicitly assume a virtual machine where all processors are dedicated to their single application. When run on a multiprogrammed machine, where many applications may be executing simultaneously, this is frequently not the case and each processor must be shared among multiple processes. In such environments, the scheduling strategy used by the operating system can have a significant impact on the performance of the parallel applications. In addition to the overhead of context switching between the multiple processes, there are two other factors that can substantially degrade the performance.

First, many parallel applications use synchronization primitives that require spin-waiting on a variable. If the process that is to set the variable is preempted, the processes that are running but waiting for the variable to be set will waste processor cycles. Scheduling strategies that ensure this happens only rarely will thus result in higher performance. Second, the frequent context switching itself can affect process cache behavior. After a context switch a process may be rescheduled on another processor, without the cache data it had loaded into the cache of the previous processor. Even if the process is rescheduled onto the same processor, intervening processes may have overwritten some or all of the cache data. Since high cache-hit rates are essential to achieving good processor utilization in modern multiprocessors [2, 12, 17], scheduling strategies that ignore cache effects may severely degrade performance.

To address the above issues, a number of different solutions have been proposed. For example, to address the synchronization problem, the gang scheduling strategy [10, 16] ensures that all runnable processes from the same application execute at the same time. Another solution is to use blocking locks instead of busy-waiting

locks to implement synchronization. To improve cache behavior, scheduling strategies that use information about the amount of data a process has on each processor's cache (its "affinity" for that processor) have been proposed [18]. Another approach uses partitioned scheduling, where processes from an application are always scheduled onto the same subset of processors, ensuring that the data shared between the processes is always found in the cache. Finally, the process control strategy [21] addresses both synchronization and cache problems, by dynamically ensuring that the number of runnable processes matches the available number of processors.

While many scheduling strategies have been proposed, as listed above, detailed evaluations using parallel applications have not been carried out—most of the previous studies [10, 18, 13, 22, 23] have used synthetic workloads and analytical models or high-level simulations. The advantage, of course, is that a much wider range of options can be studied, but many subtle but important factors are overlooked. Especially lacking has been data regarding the impact of the scheduling strategies on the caching behavior of applications. One work [21] presents results from an implementation of process control on an Encore Multimax, but there also it was not possible to separate out the effects due to synchronization and caching behavior.

In this paper, we use detailed simulation studies to explore the interaction between scheduling strategies and synchronization methods, and their impact on the overall and cache performance of applications. Our study uses the Tango multiprocessor tracing system [6], enhanced to allow multiple applications to be run at the same time with a user supplied scheduling strategy. The memory reference generator is tightly coupled to an architecture simulator that models a simple bus-based cache-coherent shared-memory multiprocessor. For each time slice, the simulation environment provides data for each process about the fraction of time spent executing useful instructions, waiting for cache misses, and waiting due to synchronization. The applications used in the study are: MP3D, a particle-based simulator used in aeronautics; LU, a program that computes the LU-decomposition of a matrix; Maxflow, a program that determines the maximum flow through a network; and Matmul, a matrix multiply program that uses a blocking algorithm.

Our results show that in situations where the number of processes exceeds the number of processors, the way synchronization is performed has a large effect on performance. With a regular priority-based scheduler, the applications achieve an average processor utilization of only 28% when using busy-waiting synchronization primitives. Almost 60% of the processor cycles are wasted in the busy loop, and the remaining 12% are spent servicing cache misses. The use of blocking synchronization primitives, especially with limited spinning before suspending, significantly improves the performance of the applications—the average processor utilization goes up to 64%. The addition of processor affinity to the scheduling strategy results in a small but noticeable improvement in the cache behavior of applications. The use of handoff scheduling results in negligible performance benefits. Gang scheduling and process control techniques are shown to offer the best performance, providing processor utilizations of about 71%. They impose no constraints on the kinds of synchronization primitives used and the processor utilization is only 3% below that achieved by a batch scheduler.

The paper is structured as follows. In Section 2 we discuss the simulation environment used in this study, and in Section 3 we describe the benchmark applications and their performance under a batch scheduler. Section 4 discusses the performance problems that arise when a regular priority scheduler is used along with busy-waiting synchronization primitives. In Section 5 we evaluate the use of blocking locks to solve the busy-waiting problem. We also explore the effectiveness of handoff and affinity scheduling. In Section 6 we evaluate the performance of gang scheduling, and in Section 7 we consider the performance of two-level schedulers with particular focus on use of process control. Finally, related work is presented in Section 8 and we conclude in Section 9.

2 Simulated Multiprocessor Environment

We use a simulated multiprocessor environment to study the behavior of applications under different scheduling strategies. The simulation environment consists of two parts: (i) a functional simulator that executes the parallel applications and implements the scheduling strategy and (ii) an architectural simulator that models the multiprocessor.

The functional simulator used for this study is an extension of the Tango multiprocessor reference generator [6]. The Tango system takes a parallel application program and interleaves the execution of its processes on a uniprocessor to simulate a multiprocessor. This is achieved by associating a virtual timer with each process of the application and by always running the process with the lowest virtual time first. Processes are run an instruction at a time so that they do not miss synchronizations with the other processes; the virtual timer is updated after every instruction. The amount by which the timer is incremented is determined by the architectural simulator. For example, if the instruction executed by a process is a memory reference, the virtual timer is

incremented by the cache hit cost if the architectural simulator determines that it is a cache hit, and by the cache miss cost otherwise.

The standard Tango system provides facilities to run only one parallel application at a time, with each process permanently scheduled onto its own processor. We have extended it to run multiple parallel applications at the same time and linked it with a scheduling module to implement different scheduling policies and blocking synchronization primitives. When a running process exhausts its time quantum or is preempted, the scheduling module is invoked and performs a context switch, if necessary, as prescribed by the scheduling policy.

The architecture simulator used for this study assumes a simple bus-based shared-memory multiprocessor.[1] Each CPU is assumed to be a RISC processor with a 64 Kbyte cache. The cache line size is assumed to be 16 bytes. The processor executes each instruction, including floating point instructions, in a single cycle (aside from memory references). Memory reference instructions that hit in the cache cost 1 cycle, and those that miss in the cache cost 20 cycles. Bus contention is not modeled. The machine is assumed to have an invalidation-based cache coherence protocol [1]. There is no overhead associated with cache coherence, in that a remote CPU is not stalled if a line has to be invalidated in its cache.

Throughout this study we assume a machine with 12 processors. For the purposes of the scheduling strategies, the default time slice length is assumed to be 250,000 cycles. On a 25MHz machine (for example, the SGI 4D/240 multiprocessor workstation uses a 25MHz MIPS R3000 processor), this would correspond to a time slice of 10ms, which is typical. We also assume a default context-switching cost of 500 cycles. This includes the cost of saving the context of the process being swapped out, that of bringing in the context of the new process to be run, and any associated run-queue/delay-queue manipulations.

3 Benchmark Applications and Their Performance

In this section, we describe the primary data structures and the computational behavior of the four applications used in this study. This information is essential for understanding the performance of the applications under different scheduling strategies. The selected applications exhibit considerable variation along several important dimensions, such as the granularity at which parallelism is exploited, the frequency of synchronization operations, and the spatial and temporal locality exhibited by the processes. All of the applications are written in C and use the synchronization and sharing primitives provided by the Argonne National Laboratory parallel macro package [14]. The four applications we studied are MP3D, LU, Maxflow, and Matmul. Table 1 provides some basic statistics about the programs when each application is run individually with twelve processes on twelve processors with the parameters used for this study. As can be seen, all applications achieve excellent speedups. The reason for LU's superlinear speedup will be discussed later.

Table 1: General statistics for the benchmarks.

Application	Total Cycles *(x 1000)*	Total Refs *(x 1000)*	Shared Refs *(x 1000)*	Cache Misses *(x 1000)*	Speedup	Shared Data Size *(Kbytes)*
MP3D	160377	37613	30165 (80.2%)	2704 (7.2%)	11.1	992
LU	92937	17381	17222 (99.0%)	738 (4.2%)	18.1	532
Maxflow	193609	48403	29004 (59.9%)	3099 (6.4%)	10.0	162
Matmul	599709	107325	54145 (50.4%)	4038 (3.8%)	12.0	10838

Figure 1 shows the breakdown of the processor execution time when each of the applications is run on one and twelve processors with one and twelve processes, respectively. Starting from the bottom, each bar shows the percentage of time spent doing useful work (or processor utilization), the percentage spent waiting for data to be fetched into the cache, and the percentage spent idling or busy-waiting due to synchronization operations (this fraction also includes overhead of context switches). In this paper we use processor utilization as our primary indicator of performance. Although utilization is mainly an indicator of throughput, not response time,

[1] We considered simulating a scalable multiprocessor with clusters and a hierarchical memory system, such as the Stanford DASH multiprocessor [12], but eventually decided against it. Since scheduling strategies for simple uniform-memory access multiprocessors are not well understood so far (the subject of this paper), we decided that making the architecture more complex would only confuse results and would be counterproductive.

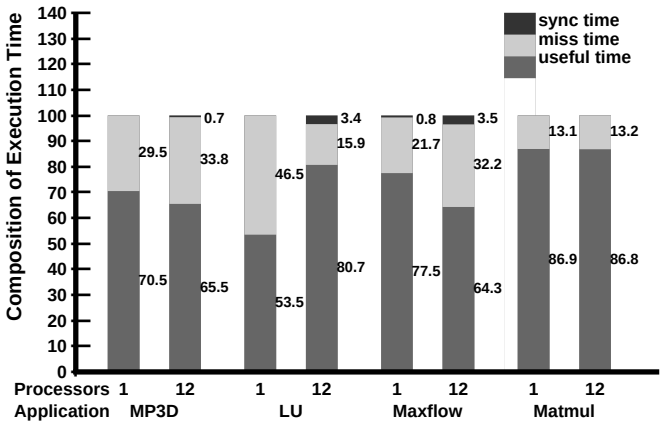

Figure 1: Application performance with 1 and twelve processors.

we do not expect this to be a problem as almost all of the scheduling strategies we consider allocate processor time fairly. Where relevant, we provide additional measures of performance.

3.1 MP3D

MP3D [15] is a 3-dimensional particle simulator. It is used to study the pressure and temperature profiles created as an object flies at high speed through the upper atmosphere. The primary data objects in MP3D are particles (representing the air molecules) and space cells (representing the physical space, the boundary conditions, and the flying object). The overall computation of MP3D consists of evaluating the positions and velocities of particles over a sequence of time steps. During each time step, the particles are picked up one at a time and moved according to their velocity vectors. If two particles end up in the same space cell they potentially collide based on a probabilistic model. Collisions with the object and the boundaries are also modeled. The simulator is well suited to parallelization because each particle can be treated independently at each time step. The particles can be divided among the processors either statically in equal numbers, or dynamically in medium-sized chunks to reduce overhead. The main synchronization consists of barriers between each time step.

In real life, the MP3D application is designed to run with enough particles to fill the complete main memory of a machine. Since at each time step in the application all particles are moved (i.e., the complete memory is swept through) the caches are expected to miss on each particle. The space cells are also expected to bounce around between the caches of the processors and not result in very good hit rates. For our experiments we ran MP3D with 25000 particles, a $64 \times 8 \times 8$ space cell array, and simulated several time steps. The number of particles handled by each processor (25000 divided by 12) is large enough that they do not fit into the 64 Kbyte cache associated with each processor. The space cell data structure is 192 Kbytes.

For MP3D, Figure 1 shows that the processor utilization for the uniprocessor version is only about 70% because of the high cache miss rate (6%). The processor utilization drops even further to 65% for the parallel version, because of an increase in the miss rate for the space cells. (The miss rate for the particles remains the same in uniprocessor and parallel versions, as references to the particles always miss.) The overall speedup for the application is quite good though, approximately $12 \cdot (0.65/0.70)$, or more than 11-fold.

3.2 LU

LU performs LU-decomposition on dense matrices. The primary data structure in LU is the matrix being decomposed. Working from left to right, columns are used to modify all columns to their right. However, a column can be used to modify others only when it has itself been modified by all columns to its left. Thus the columns get ready to modify others one by one. For the parallel implementation, columns are statically assigned to the processors in an interleaved fashion. Each processor waits until a column becomes ready, and then uses

that *pivot* column to modify all columns owned by it to the right of the pivot. Once a processor completes a column, it releases any processors waiting for that column. As computation proceeds in LU, the pivot column moves to the right and the number of columns that remain to its right decreases. As a result, the amount of data accessed and the work done per pivot column decreases, while the synchronization rate goes up.

For our experiments we performed LU-decomposition on a 256×256 matrix. Interestingly, the processor utilization for the uniprocessor is much worse than that for the 12 processor run (53% versus 80%). The uniprocessor performance is poor because the 256×256 matrix (512 Kbytes in size) does not fit into the 64 Kbyte cache, resulting in a poor cache hit rate. For the 12 processor case, however, each processor owns only 1/12 of the matrix, which does fit into the cache. The result is much higher processor utilization and superlinear speedups. Some of the benefits of the reduced cache miss time in the 12 processor run are lost, however, due to synchronization overhead—about 4% of the cycles.

3.3 Maxflow

Maxflow [4] finds the maximum flow in a directed graph. This is a common problem in operations research and many other fields. The program is a parallel implementation of an algorithm proposed by Goldberg and Tarjan [11]. The bulk of the execution time is spent picking off nodes from a task queue, adjusting the flow along each node's incoming and outgoing edges, and then placing its successor nodes onto a task queue. Maxflow does not assign the nodes of the graph to processors statically. Instead, task queues are used to distribute the work. Each processor has its own local task queue and needs to go to the single global task queue only when its local queue is empty. Tasks are put on to the global queue only when processes are waiting there, and on to the local queue otherwise. The task queues are made up of the nodes themselves, linked together with appropriate pointers. Maxflow exploits parallelism at a fine grain.

In Figure 1 we see that the processor utilization of Maxflow goes down from 78% with 1 processor to 64% with 12 processors. The primary reason is a significant increase in the miss rate; miss time goes up from 21% to 32% as nodes and edges bounce between the processors' caches. As the number of processors increases, there is a greater chance in the parallel version that multiple activations of the same node get evaluated on different processors. The time spent performing synchronization also increases by 3.5% in the parallel version. The application speedup for 12 processors is about 10-fold.

3.4 Matmul

Matmul is a highly optimized block-based parallel algorithm for multiplying two matrices. To multiply two matrices A and B to produce C, the computation proceeds by first copying a block of B into the cache,[2] and then repeatedly using it to generate data for the matrix C. When all computation associated with the currently loaded block of B is finished, the next block is loaded into the cache and used similarly. The parallelism exploited is very coarse grain and negligible synchronization is involved.

For our experiments we used matrices of 672×672 elements (approximately 3.5 Mbytes in size) and a block size of 56×56 elements (about 25 Kbytes in size). Since the resulting computation is very long, only a small part of the computation is shown in the graphs. The processor utilization stays very high, about 87%, for both uniprocessor and 12 processor runs and perfect speed-up is achieved.

4 Performance of a Simple Priority-Based Scheduling Strategy

In the previous section, we discussed the performance of the applications when run with one and twelve processors. The applications were assumed to run in isolation, one at a time, and consequently scheduling did not matter.[3] The runs in the previous section, in some sense, represent the best case performance for the applications. We now consider situations where multiple applications are running at the same time on the machine.

[2]The block of B is first copied into contiguous local memory before being used in computation. This is to avoid interference between the block's rows when they are loaded into the processor cache.

[3]By scheduling, we mean operating system level scheduling and not the allocation of work to processes within the application. Internally, each application assigns work to processes in a manner to enhance locality.

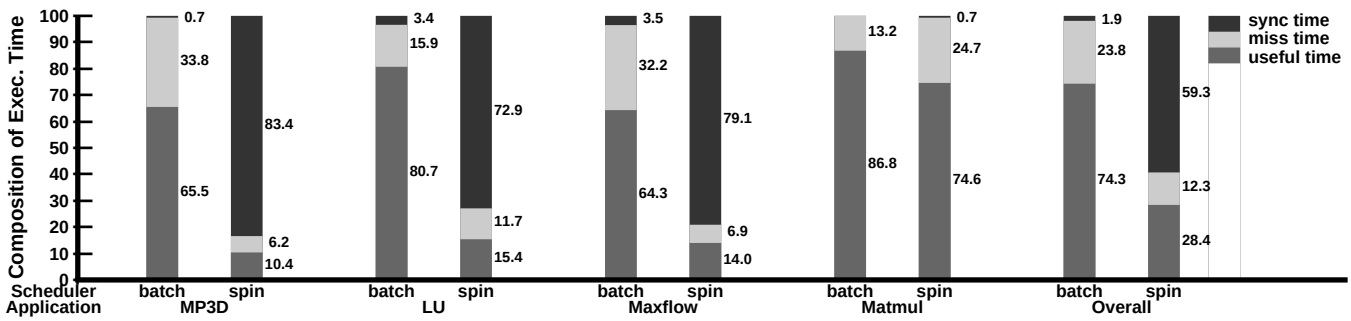

Figure 2: Comparison of batch mode and regular priority scheduling.

In this section, we focus on the performance of a simple priority-based scheduling strategy, similar to what is used in many existing multiprocessors [3]. We assume that there is a single run-queue that holds all runnable processes. The run-queue is ordered by priority, where the priority is simply the inverse of the accumulated running time of a process. Whenever a processor finishes a time slice, it performs a context switch and picks the highest priority process from the run-queue to execute. If there are multiple processes with the same priority, then one is randomly chosen.[4] As stated in Section 2, the time slice duration is assumed to be 250,000 cycles and we assume the context switch overhead to be 500 cycles. No I/O effects are taken into account. We further assume, for the present, that all synchronization primitives in the applications are based on busy-waiting.

Figure 2 shows the performance of the four applications when they are run under batch mode (each application run individually on twelve processors until termination) and under the scheduling strategy described above. Thus there are a total of forty eight application processes, twelve from each application, being run on twelve processors. In our simulations, the LU application finishes earliest. Since we are only interested in the performance when all applications are running, the results are based on data taken up to the point LU finishes.[5] By the time that LU finishes, the behavior of MP3D, Maxflow, and Matmul has reached steady state for a long time, so this does not introduce any distortions.

The degradation in system performance with priority scheduling and busy-waiting locks is extreme. The overall processor utilization is down to 28%, compared to about 74% when the applications are run in batch mode. The poor performance is almost totally due to the busy-waiting synchronization primitives used by the applications. The average time spent in the busy-loop goes up from about 2% for the batch mode to 59% when all processes are time multiplexed together. Let us look at the performance of the applications from left to right.

For MP3D, the utilization drops from about 56% to 10%. This is despite the fact that most of the parallel tasks in MP3D are very coarse grain (moving thousands of particles at a time) and involve no internal synchronization. The performance degradation actually results from the barriers between the particle-move phases. This can be more easily observed from Figure 3 which shows the mean utilization of processors running MP3D processes for every time slice. The narrow columns, where the processor utilization is around 60%, represent the phase when the particles are being moved (the bulk of the useful computation in the application). The large gaps between the columns, where the processor utilization is close to zero, correspond to the seven barriers that occur between each phase when all particles are moved. During this phase the only computation that a process may do during a complete time slice is to increment the barrier count variable. These barriers take negligible time in batch mode, but enormous time here.

For LU the utilization drops from about 81% to 15%, a 5-fold loss in performance. This is because of the fine-grain synchronization that is involved between the producers and consumers of completed columns. The

[4]If the accumulated running time of two processes is within 1000 cycles, their priority is considered to be the same. The random selection also helps avoid regular patterns in which process assignment in our controlled experiments.

[5]As a result, our notion of batch mode actually involves running each application for an amount of time corresponding to the total running time of LU, and not until termination. This avoids weighting some applications more than others in overall performance.

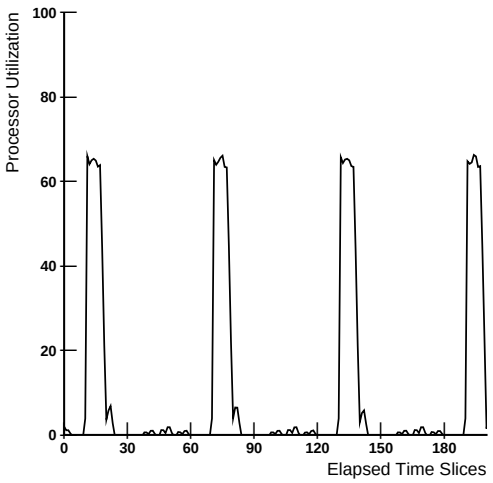

Figure 3: Utilization of processors executing MP3D as a function of time.

performance drop for Maxflow from 64% to 14% is also due to fine grained synchronization. Matmul is the only application that retains reasonable performance as it uses no synchronization. Matmul's processor utilization drops from around 87% to 75% solely due to cache effects. The cache miss rate goes up from 4% to 8% since each Matmul process has to load back the block of matrix that it was previously working with whenever it is rescheduled.

As the above data show, almost all of the performance loss in the current scheme is occurring due to the cycles wasted by busy-waiting synchronization primitives, and not due to poor cache performance or context-switching overhead. Consequently, we focus on the busy-waiting problem first. There are several possible solutions. We first explore using blocking instead of busy-waiting synchronization primitives. We subsequently explore gang scheduling and two-level scheduling with process control. The latter techniques allow busy-waiting primitives to be used, but still ensure good performance.

5 Priority-Based Scheduling with Blocking Synchronization

On finding a synchronization variable busy, a blocking lock relinquishes the processor, making it available to other processes in the run queue. In this way it potentially reduces the waste of processor cycles spinning. In fact, there is some chance that the process it was waiting for can now be run on the just released processor. Although operating systems have long provided such blocking semaphores [7], they are rarely used by parallel applications programmers. The reason is that parallel applications frequently use small critical sections where the probability of blocking for long periods is small (from the perspective of the application programmer), and the large overhead of accessing and blocking on semaphores is thus unattractive. However, as we saw in the previous section, if the process within the critical section is unscheduled due to multiprogramming, the waiting times can become very large. Consequently, it becomes desirable to study the performance of applications where the busy-waiting primitives are replaced with blocking synchronization primitives.

For the results reported in this section, we replaced all busy-waiting locks in our four benchmark applications with blocking locks. The scheduling strategy still maintains a single run queue sorted by priority. Priority as before is inversely proportional to running time accumulated by a process. In our implementation, when a process wishes to obtain a lock, it first spins for up to 500 cycles (equal to the cost of a context switch). This keeps the process from incurring the overhead of blocking when a lock is being held for a short time by a process that is scheduled.[6] If the process has not obtained the lock by the end of the 500 cycles, it releases the processor to the highest priority process in the run queue, and blocks itself on a queue associated with the lock. When the lock is released by some process, depending on the type of the lock variable (semaphore or event variable), one or all blocked processes are moved to the run queue.

Figure 4 shows the performance of the four applications when using batch scheduling, priority scheduling

[6]The V operating system [5] also uses blocking locks that spin in the beginning.

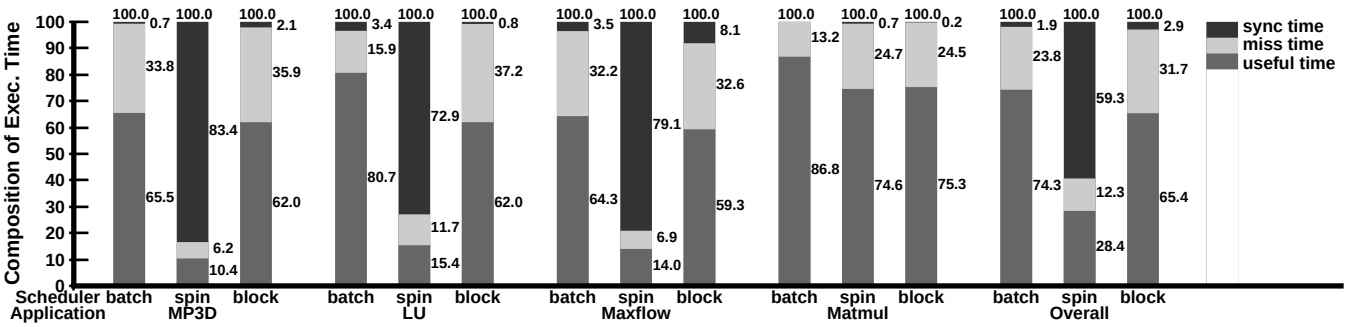

Figure 4: Comparison of batch scheduling and regular priority scheduling with busy-waiting and blocking locks.

with busy-waiting locks, and priority scheduling with blocking locks. The format is the same as in the previous section. The right most set of three bars show the overall system performance, while the four sets of bars on the left show the contributions of the individual applications. The results show a great improvement over scheduling with busy-waiting locks for all applications except Matmul, which changes little since it has no synchronization. The overall processor utilization goes up from 28% for busy-waiting to 64% for blocking locks. (It is still lower than the 74% achieved by the batch mode, but we will look at locality enhancing optimizations later.) When using blocking locks, the overhead due to spinning and context switching, shown as synchronization time on the bars, remains below 2% for all applications except for Maxflow. The overhead for Maxflow is about 9% due to its use of very fine grained synchronization. One important implication of the low overhead time is that blocking locks are expected to perform reasonably well even if the overhead of context switching were larger.

As stated earlier, one parameter in the implementation of blocking locks is the amount of time a process spins before blocking. At one extreme, the process could spin forever before blocking, which would be the same as a busy-waiting lock. At the other extreme, the process could block as soon as it finds that it must wait. This avoids spending any time spinning, but often results in excessive context switches. This is especially true when critical sections are small. Excessive context switches are a problem both due to the time taken to switch contexts and due to the loss of data accumulated in the cache. Our compromise was to have the process spin for an interval equal to the context-switch time (500 cycles) before blocking. The reason is that it guarantees that the performance will be within a small constant factor of the optimal blocking strategy.

In Figure 5 we show the performance of Maxflow as the amount of spin time before blocking is varied. (We only considered Maxflow as we expected it to show the most response to such variations.) For this study the Maxflow application is considered in isolation with twelve processes running on 3 processors. The five values for the spin time that we consider are 0, 250, 500, 1000, and 5000 cycles. From the bars we can see that the corresponding processor utilizations are 44%, 42%, 61%, 61%, and 54%, respectively. The context switch rates, that is, the number of context switches per time slice per processor, are 127, 93, 21, 8, and 7 respectively. The cache miss times drop slightly from 26% to 20% as we go to larger spin times. The data makes the disadvantages of allowing too small a spin time before blocking clear. The data also seem to indicate that erring on the side of too much spin time is safer than erring towards too little time.

In summary, we see that blocking synchronization primitives substantially improve performance over busy-waiting primitives when there is significant contention for the processors. We now explore some possible enhancements to the basic blocking strategy.

5.1 Handoff Scheduling

In the scheduling strategy we just studied, when a process blocks or at the end of a time slice, the processor simply picks the highest priority task from the run queue as the next task to execute. In this subsection, we explore the performance of *handoff scheduling*, an alternative strategy for deciding which process to relinquish

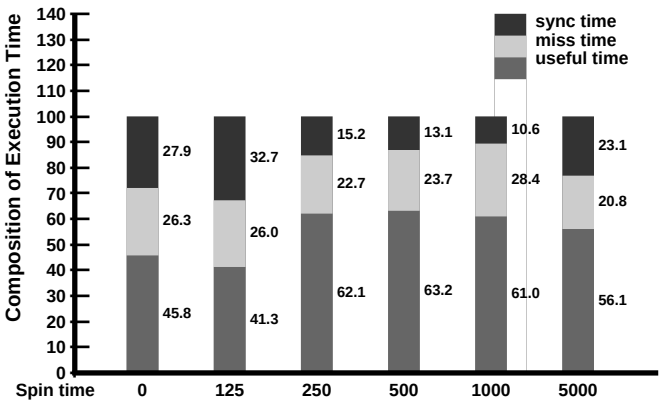

Figure 5: Comparison of Maxflow performance with varying amount of spin time before blocking.

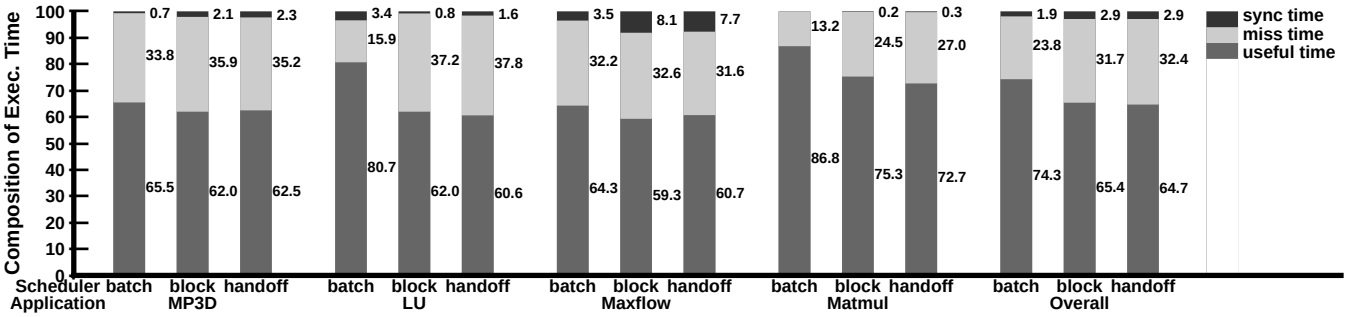

Figure 6: Comparison of batch scheduling and normal and handoff scheduling with blocking locks.

the processor to on blocking. Handoff scheduling is based on an approach used for remote procedure calls in Mach [3] and Topaz [19], where a client process calling a server process may "hand off" its processor to the server. When the server responds, the processor should be returned to the client process. The basic idea behind handoff scheduling is that a blocking process should be able to turn over the remainder of its time slice to another process that may help expedite the event for which the blocking process is waiting. For example, a process trying to acquire a lock may hand off its processor to the process holding the lock, so that the lock will be released as quickly as possible.

In our implementation of handoff scheduling, the processor of a blocking process is assigned to a ready process from the same application that is not currently running. This process can either be specifically identified by the application or selected automatically from the available processes according to least accumulated running time. The processor is assigned only for the remaining duration of the time slice of the blocking process. In our study, we specified the target process that should be handed off to in two situations. First, when a preempted process holds a lock, processes blocking on that lock hand off to it. Second, in LU, each pivot column is "owned" by a specific process. In order for other processes to make use of that column, an event associated with that column must be set by the owner process. Whenever a process blocks waiting for a column to be ready, it hands off to the process owning that column, if that process is not already scheduled.

Figure 6 shows the performance of a system with blocking locks and handoff scheduling. Compared

to regular blocking scheduling, the overall improvement due to use of handoff is negligible—the combined processor utilization goes up from 64.0% to 64.2%. The performance decreases slightly for LU and Matmul and increases slightly for Maxflow and MP3D. These mixed performance results are due to a number of factors. We found that the early scheduling of the process releasing a lock did not reduce the synchronization time, as might have been expected, for any of the applications. This is because regular blocking locks already eliminate virtually all busy-waiting time, and there is enough concurrency in the system that, with 48 processes, there are no idle processors. The overhead due to context switching, on the other hand, goes up slightly for all applications with handoff scheduling—the context switch rate goes up from 4.3 switches per time slice per processor for regular blocking scheduling to 5.0 for handoff scheduling. This is because handoff scheduling only runs a process for the remaining portion of the time slice. This factor adds at most an extra context switch per time slice, and therefore its effect is minor (a loss of 0.2% in performance).

The third factor affecting performance is change in the cache miss rate for the applications. The changes in the miss rate occur because, on blocking, the processor is given over to another process from the same application rather than to a random process from a totally different application. The benefits depend on the degree of sharing between processes of the same application. While the miss rate remains the same for LU, it increases slightly for Matmul, and decreases slightly for Maxflow and MP3D. The miss rate remains the same for LU as there is little sharing between its processes (only the pivot column is shared). For Matmul, the processes do not block due to synchronization, so handoff plays no role. However, since the overall rate of context switching goes up due to execution for partial time slices, the miss rate increases as matrix blocks have to be refetched into the cache when the process is rescheduled. Both Maxflow and MP3D have limited amount of shared data (nodes and edges for Maxflow and space cell array for MP3D), so slight improvements in cache miss rates are observed.

In summary, if the overall concurrency in the system is high, handoff scheduling does not appear to buy extra performance as there are always ready processes to be run. Similarly, if the concurrency in the system is low so that there are idle processors, then again handoff scheduling does not result in higher performance. It is only in intermediate situations that handoff scheduling may improve performance. From the data that we have, at least as far as throughput is concerned, the usefulness of handoff is unclear.

5.2 Affinity Scheduling

While handoff scheduling can sometimes help improve the cache hit rate, it does so only indirectly. In this subsection, we evaluate *affinity scheduling*, a class of scheduling strategies that directly focus on cache performance. The scheduling strategies exploit processor affinity, the amount of data that a process has in a particular processor's cache, in making scheduling decisions. The idea behind processor affinity is that if processes are rescheduled onto processors for which they have high affinity, much of the processes' original data may still be in the cache and hit rates will improve.

There are several factors that determine the effectiveness of affinity. The first factor is the footprint size of the application [20], that is, the data set that needs to be almost immediately loaded into the cache when a process is scheduled to run on a processor. The second factor is the duration for which a process runs on a processor once it is scheduled. This is a function of the time slice length and the probability of blocking before the end of time slice. The third factor is the number of intervening processes (and their footprint sizes) that are scheduled on a processor between two successive times that a process is scheduled on that processor. Finally, the importance given to affinity in relation to fairness makes a difference. Given these factors, we expect the benefits to be small if the time to load in the footprint is small relative to the interval for which the process runs on the processor. Similarly, the benefits will be small if the intervening processes (even if it is only one process) wipe out the complete previous contents of the cache.

There are two important questions that must be addressed by any affinity-based scheduling strategy. First, how is affinity to be computed, given that it is not possible to know in any practical way the exact amount of useful data in a processor's cache. For example, Lazowska and Squillante [18] use the inverse of number of intervening processes that have run on a processor as a measure of affinity. In our study, we use a similar measure, the inverse of the number of cycles since the process last ran on a given processor.[7] The second question is how to balance between fairness to all processes (which helps to improve response time) and affinity

[7] Actually, after some time the duration since the process last ran on a processor does not matter since the data is probably already gone from the cache. For example, both MP3D and Matmul applications fill up most of the cache during a time slice. Therefore, even one intervening time slice of these applications is sufficient to wipe out most of the useful data from the cache. We expect many blocked scientific applications to behave like these two.

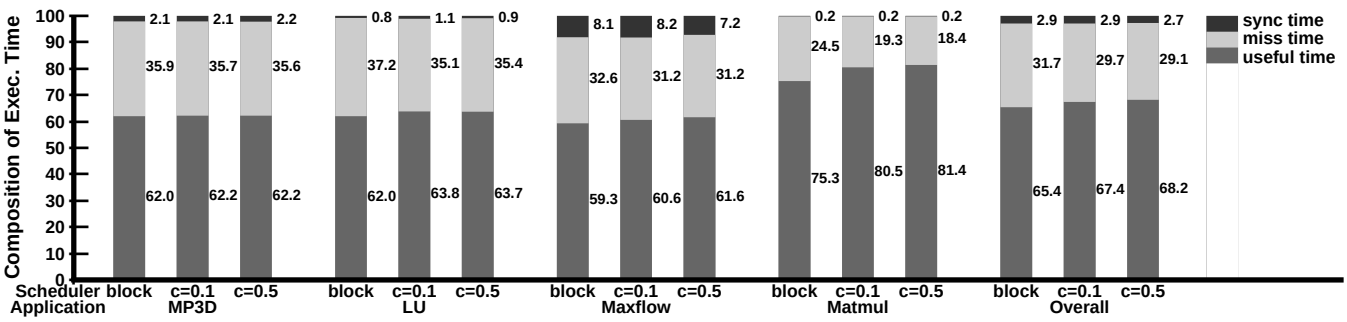

Figure 7: Comparison of blocking scheduling and affinity scheduling with $c = 0.1$ and $c = 0.5$.

(which helps to improve throughput). Here again there is a range of possibilities. For example, one extreme is batch scheduling, where all processes of the same application run on the processors, each process stays with the same processor for its lifetime, and only when one application ends is the second one scheduled. The results for this case were described in Figure 1. The other extreme is vanilla priority scheduling, where the priority is the inverse of accumulated running time. The results for this case were shown in Figure 4. The intermediate strategy that we adopted for the current affinity study is described below.

For our affinity studies we use blocking locks and priority scheduling, except that priority is computed differently. We use the formula $p = 1/(T + c \cdot l)$ to compute the priority of a process in the run queue. In the formula, T is the total accumulated running time of the process (this factor ensures that some semblance of fairness is maintained), l is the duration since this process last ran on the processor (inverse of affinity), and c is a weighting factor that indicates the importance to be given to affinity. For example, if $c = 0$ then we are back to normal blocking scheduling.

Figure 7 shows the performance of our applications with blocking locks and affinity scheduling with $c = 0.1$ and $c = 0.5$. With c set to 0.1, the factor c is small enough that scheduling is fair, but large enough to ensure that processes end up on the same processor when they are rescheduled. For example, without affinity, the probability that a process is rescheduled on the same processor where it last ran is about 19%. With affinity scheduling and $c = 0.1$ this probability goes up to 67%, a significant increase. Similarly, the average number of intervening processes that are scheduled on a processor between two successive runs of the same process is 33.4 without affinity. With affinity of $c = 0.1$ this number comes down to 9.0. Regarding fairness, at the time when LU finishes, MP3D, LU, Maxflow, and Matmul have used 26%, 20%, 26%, and 29% of the total machine cycles respectively. The effect of affinity is uniformly positive, with each application obtaining slightly higher processor utilization. Processor utilization for MP3D goes up from 61.7% to 62.4%, for LU from 62.1% to 63.7%, for Maxflow from 57.6% to 61.3%, and for Matmul from 74.2% to 79.5%. The overall increase in utilization is about 3.3%.

For MP3D, most cache misses are steady state misses and not initial loads of a footprint into the cache, and consequently the benefits of affinity are small. For LU the footprint is reasonably large, but the benefits are small because if there is even one intervening process from MP3D or Matmul, it displaces most of the previous contents of the cache. For Maxflow, the benefits come both from slightly reduced cache miss time and synchronization time. For Matmul, the benefits are the most because it has a large footprint size (the matrix block is 25 Kbytes in size), and if Maxflow or LU processes intervene, they do not totally destroy useful data. This is partly because Maxflow and LU block frequently.

With $c = 0.5$, affinity is considered even more important in scheduling decisions. For example, the probability of rescheduling a process on the same processor rises to 86% and the average number of intervening processes goes down to 5.6. The performance is even greater than with $c = 0.1$, but the scheduling is no longer fair between applications. Since it does not block, Matmul receives about 53% of the total machine cycles, as compared to 17% for MP3D, 12% for LU, and 18% for Maxflow. This is because on any given processor there

is always a runnable Matmul process that has been executed recently on that processor. With other applications, the most recently executed process is often blocked waiting for other processes to run. Since Matmul has a higher utilization than the other applications, its domination of processor time results in a very high overall utilization, 74%. If instead we were to weight each application equally, the overall performance drops to 69%. While higher than the utilization with $c = 0.1$, the slight increase is probably not justified by the lack of fairness and the resulting increase in mean response time.

In summary, while the gains of affinity have been small, they have been consistently positive for all applications. The reasons for the small gains have either been a small footprint size compared to the time slice duration (e.g., MP3D) or the fact that relevant data is almost totally replaced by intervening processes (e.g., data for LU is replaced by that of MP3D and Matmul processes). We further observe that too much emphasis on affinity, while resulting in higher processor utilization, can lead to serious problems with fair allocation of processor time.

6 Gang Scheduling Strategy

In the previous section we evaluated the effectiveness of blocking locks in reducing the time wasted by busy-waiting locks. In this section, we evaluate an alternative solution initially proposed by Ousterhout [16]. In coscheduling, or *gang scheduling*, all runnable processes of an application are scheduled to run on the processors at the same time. When a time slice ends, all running processes are preempted simultaneously, and all processes from a second application are scheduled for the next time slice. When an application is rescheduled, effort is also made to run the same processes on the same processors.

There are two main advantages of gang scheduling. First, it solves the problems associated with busy-waiting locks by scheduling all related processes at the same time. Second, it provides a more static runtime environment to the software, which can be used to optimize performance. For example, in a partitioned shared-memory machine, a process may allocate storage from a memory that is physically close by to reduce cache miss penalty. Such optimizations can not be used in the schemes described earlier, as processes and processors are not so tightly bound.

Of course, gang scheduling also has several disadvantages. First, it is a centralized scheduling strategy, with a single scheduler making decisions for all applications and all processors. This centralized nature can become a bottleneck for large machines. Second, gang scheduling can result in poor cache performance because by the time an application is rescheduled on the system, the caches may not contain any of its data. This will be most noticeable for modern machines with large caches and applications that exploit significant locality. Third, gang scheduling can lead to fragmentation of processors when there are applications that do not need all of the processors in the system, but which do not leave enough free processors to fit all of the processes of another application. These considerations can make the implementation of a gang scheduler that works in a real multiprogramming environment quite difficult.

In our study, we simulated a simple form of gang scheduling by interleaving execution of MP3D, LU, Maxflow, and Matmul applications for one time slice each. Each application ran with twelve processes and used busy-waiting locks. Each process from an application is always run on the same processor. The time slice, as before, is 250,000 cycles, corresponding to 10ms on a 25MHz processor. The time slice is long enough for MP3D and Matmul to fill up large portions of the cache with their data. As a result, little relevant data will be left in the cache by the time any application is rescheduled. Figure 8 shows the performance of gang scheduling with the 10ms time slice. The overall processor utilization is 67%, much better than the 28% achieved when busy-waiting locks are used with regular priority scheduling. The utilization is also better than the 64% achieved when blocking locks are used, but still worse than the 74% achieved by batch scheduling.

Comparing the processor utilizations obtained under batch mode execution and under 10ms gang scheduling, we see that the processor utilization is less for each of the applications. It is lower by 4% for MP3D, 11% for LU, 3% for Maxflow, and 9% for Matmul. The reason is simply that the cache is virtually flushed every 10ms for gang scheduling, while no such flushes occur for batch mode. Consequently, applications like LU and Matmul that accumulate a large amount of useful data in the cache suffer more than MP3D and Maxflow. One obvious way to increase the performance of gang scheduling is to increase the duration of the time slices. To study the effects of such a change, we also experimented with time slices of 625,000 clock cycles (corresponding to 25ms on a 25MHz processor). The results of this experiment are also shown in Figure 8. The overall processor utilization of the machine goes up to 71%, compared to 67% for 10ms time slices and 74% for the batch mode. The most significant increases were observed for LU (7%) and Matmul (7%). An increased time slice duration

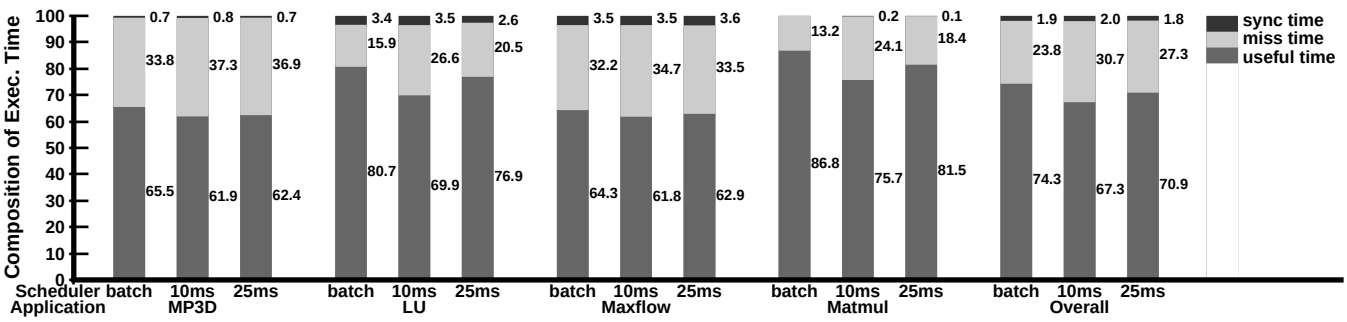

Figure 8: Comparison of batch scheduling and 10ms and 25ms time slice gang scheduling.

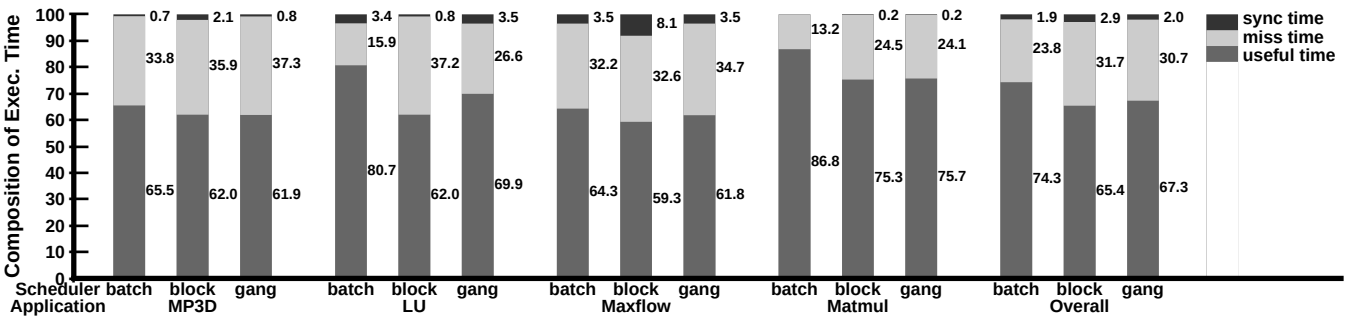

Figure 9: Comparison of batch, blocking, and gang scheduling.

also helps reduce the direct overhead of context switching.

We now compare the performance of gang scheduling with 10ms time slices with that of priority-based scheduling using blocking locks. The performance is approximately the same for MP3D. Gang scheduling has slightly higher miss time (since all MP3D processes are active at the same time and invalidate items from each other's cache), but blocking locks lose this advantage due to higher context switch rates (5.3 for blocking versus 1.0 for gang). For LU, gang scheduling performs significantly better than blocking locks (70% versus 62% utilization). Blocking locks reduce synchronization time slightly (by 2% compared with gang scheduling), since the process relinquishes the processor whenever the pivot column is not ready, but the frequent blocking results in substantially increased miss time (10% compared with gang). For Maxflow, gang scheduling again wins over blocking locks by 4%. The main reason is that the time wasted due to busy-waiting in gang scheduling (about 4%) is less than the time wasted by busy-waiting and context switching when blocking locks are used (about 9%). Finally, for Matmul the performance is approximately the same for both cases. Gang scheduling does slightly better (by 1.5%) due to processes being run on the same processors.

In summary, gang scheduling performs well despite the cache data lost at the end of every time slice. With all processes from an application running simultaneously, there is no danger of busy-waiting for an unscheduled process, and thus no need for blocking locks. Without blocking locks, the frequency of context switches is reduced, and applications that rely on reuse of cached data can perform well for most of a time slice. The

resulting performance is similar to that of a blocking lock implementation using processor affinity, and exceeds it when a larger time quantum is used.

7 Two-level Scheduling Strategies

The scheduling strategies that we have considered so far have all time-multiplexed the processors among the applications. An alternative set of scheduling strategies can be derived by partitioning (space multiplexing) the processors among the applications. Of course, the partitioning must be dynamic in a multiprogramming environment, since applications may be entering and leaving the system all the time. In this section we consider the performance of such two-level schedulers, where there is a high-level policy module that decides how the processors are to be partitioned and a low-level scheduling module that manages the processes of an application within a partition.

Since the focus of our current study is not resource distribution policy issues, we assume throughout this section that the processors are equally divided among the four applications. We also do not directly address issues of dynamically changing process and application loads. (Some of these issues were explored in a previous paper [21].) Thus, three processors are dedicated to each application. Within each partition one could schedule processes using priority scheduling with busy-waiting locks, priority scheduling with blocking locks, or a technique called process control. We do not present results for busy-waiting locks as the performance is very poor, as described in Section 4. We present results for the other two cases.

7.1 Two-level Scheduling with Blocking Locks

Figure 10 shows the performance of priority scheduling with blocking locks for three cases: (i) with no processor partitioning (leftmost bar for each application); (ii) with processor partitioning (center bar); and (iii) with processor partitioning and affinity (rightmost bar). Overall, the processor utilization achieved with partitioning is 64.7%, only 0.7% higher than without partitioning. The performance increases for all applications, except LU, due to the fact that only related processes with shared data are scheduled on the processors. The performance degrades for LU because its processes have large disjoint data sets (the columns that they own) that cause significant interference in the caches, and overwhelm the small benefits due to the shared pivot column. The performance of Maxflow also shows interesting behavior. The cache miss time drops by about 10% while the synchronization time increases by about 7%. The decrease in the cache miss time is simply due to the shared data between the processes. The increase in synchronization time, however, is due to the fact that in the partitioned approach a maximum of three Maxflow processes can be running at the same time. While on the average, the latter is also true for the non-partitioned approach, there are instances when many more Maxflow processes are scheduled thus reducing synchronization costs. The context switch rate of maxflow goes up from 7.8 switches per time slice per processor for the non-partitioned case to 21.1 for the partitioned case.

When affinity is added to the partitioned scheme, the overall performance increases by about 2%. These gains are slightly smaller than those achieved when affinity was added to the non-partitioned case, suggesting that the effect of affinity is lessened when all processes running on a given processor are from the same application, and thus using some of the same data. Also, the reduced number of processors used by a given application results in reducing the time between repeated executions of a process on the same processor, even without affinity. This reduces the effect when affinity is added.

7.2 Two-level Scheduling with Process Control

All of the approaches we have explored so far have tried to deal with the negative effects of having too many processes and too few processors. An alternative strategy, called *process control* [21], instead focuses on techniques that allow applications to vary the number of processes they use in response to the changing number of processors available to them. If the number of runnable processes matches the number of processors, context switches can be largely eliminated, and good cache and synchronization behavior is ensured.

In the process control approach, a policy module in the operating system continuously monitors the system load and dynamically adjusts the number of processors allocated to each application. The applications respond to the changing number of processors by suspending or resuming processes. The process control approach is most easily applied to parallel applications that are written in a task-queue or threads style [8], where a number

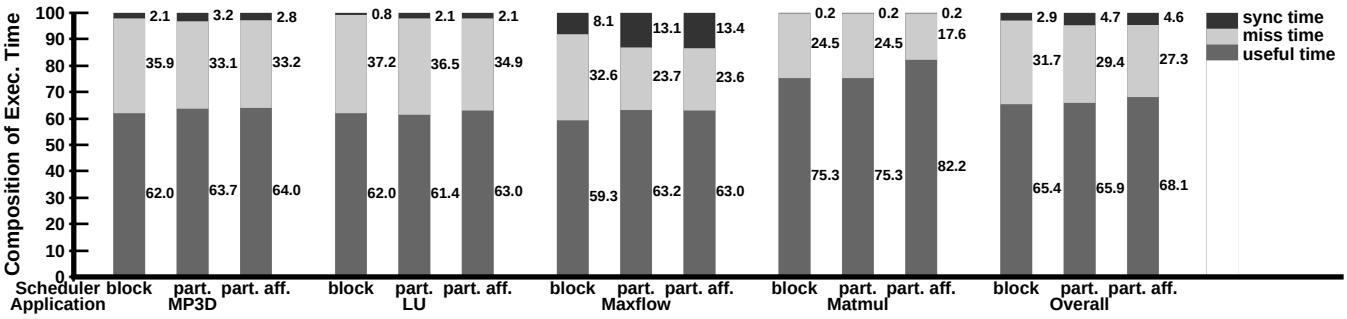

Figure 10: Comparison of blocking, partitioned blocking, and partitioned blocking with affinity scheduling.

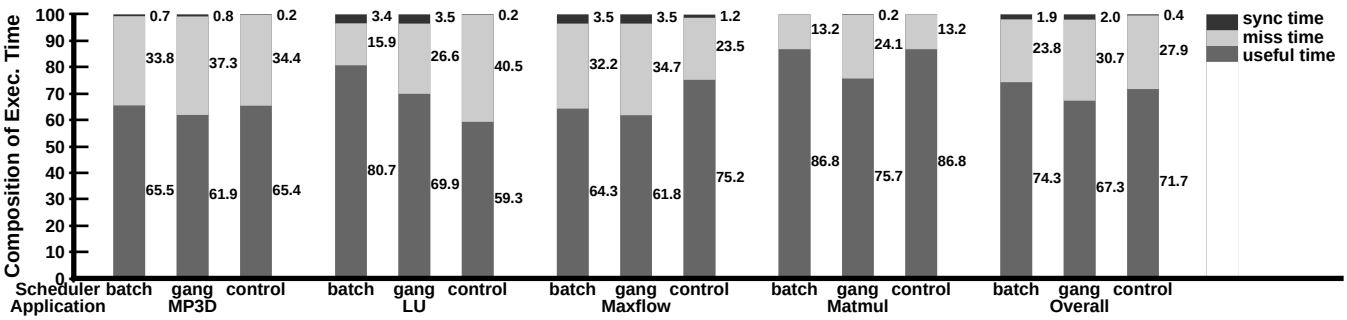

Figure 11: Comparison of batch, 10ms gang, and process control scheduling.

of user-level tasks (threads) are scheduled onto a number of kernel-scheduled server processes. In such an environment, the number of server processes can be safely changed without affecting the running of user-level tasks. Some performance numbers from an implementation of this approach on the Encore Multimax were presented previously [21]. In this subsection we look at the cache and synchronization behavior of process control in a more controlled setting.

Figure 11 shows the performance of a system using process control. Since there are twelve processors and four applications, each application partition has three processors. Each application thus uses three processes. The overall system performance is higher than that of gang scheduling with 10ms time slices, about the same as that of gang scheduling with 25ms time slices, and better than any of the other reasonable approaches. It is interesting to compare the performance of the individual applications under process control to that obtained under batch mode and using gang scheduling.

Comparing first to the batch mode, we see that for all applications except LU, process control does at least as well as the batch mode. The processor utilization is about the same for MP3D, it is lower for LU (59% versus 81%), it is substantially higher for Maxflow (75% versus 64%), and it is unchanged for Matmul. Focusing first on the performance of Maxflow, process control does better for two reasons. First, the cache miss time is significantly lower (down from 32% to 24%) since there are fewer cache invalidations when there are fewer processors working in parallel. Second, there is also less contention for the task queues and nodes, which

reduces the synchronization time from 3.5% to 1.2%. The performance of LU is lower with process control for an interesting reason. While the complete matrix being decomposed fits into the caches of twelve processors, it does not fit into the caches of three processors. The result is the cache miss time increases from 15.9% to 40.5%. It is not clear, however, how genuine this advantage of the batch mode is. If we were to use a much larger matrix, that did not fit into the caches of 12 processors, the difference in performance will be reduced. Similarly, if we were to use a blocking method for LU (as used in Matmul) the advantage would again be lost, as the process control performance would increase. In either of the above scenarios, we would find the overall performance of the process control approach to be even higher than that of the batch mode.

Comparing the performance of process control to gang scheduling, process control does better for all applications but LU. In general, the performance gains are larger than those for the batch mode because gang scheduling has worst cache behavior due to cache replacements that happen with each intervening application. As a result of the cache replacements, the processor utilization of Matmul is 86.8% for batch mode and process control, and only 75.7% for gang scheduling. When 25ms time slices are used, the utilization of Matmul with gang scheduling goes up to 81.5%, but still stays below that of process control.

The noticeably lower performance of Maxflow under batch mode and gang scheduling, as compared to the process control approach, brings up an interesting though subtle point that we illustrate with the help of an example. Most parallel applications give sublinear speedups with increasing number of processors. For instance, consider two applications A and B that give 7-fold speedup with 8 processors but only 12-fold speedup with 16 processors. If both these applications are to be run concurrently on a 16 processor machine, it seems best to run them each with 8 processors. If, however, only one of the applications is to be run, it is best to use all 16 processors, since otherwise the remaining processors will be idle.

In a multiprogramming environment, where it is not known what applications will be running in parallel, the gang scheduling and the process control approaches take opposing positions on how applications should be executed. The gang scheduling approach encourages that the applications be spawned with processes equal to number of processors, even if there are other applications running. This is because the other running applications may finish soon, and thus spawning with fewer processes will leave processors idle. Spawning with fewer processes is also undesirable because if other applications are running with a larger number of processes they will get an unfair share of the machine cycles. As a result, the gang scheduling approach may result in suboptimal performance when multiple applications are running. Since the process control strategy dynamically adjusts the number of processes with the system load, it performs well under all circumstances.

In summary, the process control approach appears to handle synchronization and cache behavior problems well for all applications. For the applications considered, its performance is about 4% higher than that of gang scheduling and only 3% below that of batch scheduling. For large multiprocessors, we expect its two-level structure to also help remove the bottleneck of a single global scheduler required by the gang scheduling approach.

8 Related Work

8.1 Blocking versus Busy-Waiting Locks

In Section 5 we explored the use of blocking locks to avoid performance loss due to busy-waiting for preempted processes. To address this issue, researchers at the NYU Ultracomputer project [9] proposed an alternative solution, where individual processes can temporarily prevent their own preemption in the form of a hint to the scheduler. This mechanism can be used to prevent processes from being preempted while they are holding a lock. While such a mechanism helps solve the busy-waiting problem for locks, unfortunately, it does not improve the performance of more complex synchronization mechanisms like barriers and events.[8] This approach also does not address the cache corruption issue that arises in context-switching multiprocessor schedulers. The researchers do not present any quantitative data on the effectiveness of their proposal.

Zahorjan et al. [22] at the University of Washington also propose a scheduler that avoids preempting a process while it is inside a critical section. They additionally suggest an approach for reducing cycles wasted due to spinning by not rescheduling processes that are waiting for a held lock. A third approach combines the previous approaches, both avoiding preempting a process holding a lock and avoiding scheduling a process waiting for a lock. The researchers used analytic models to study the performance of the above techniques.

[8] The overall proposal also included provisions for gang scheduling, so the performance of barriers could be addressed in that manner.

They found that all three approaches worked fairly well in avoiding waiting for spinning locks. With locks normally held for a long duration, avoiding rescheduling waiting processes worked best. With short-duration locks, avoiding preempting the process holding the lock worked best. The combined approach worked well at all times. The second approach to lock contention, avoiding rescheduling waiting processes, was also applied to the problem of barrier synchronization. While the time taken to achieve a barrier improved under this approach by as much as 30%, performance still degraded significantly under heavy multiprogramming loads.

Another work by the same authors [23] compares gang scheduling with partitioned round-robin scheduling, the latter with spinning locks and with blocking locks. Again using analytical models, they found that partitioned scheduling with spinning synchronization performed very poorly due to processes being preempted while holding locks. Between gang scheduling and blocking synchronization, they found that blocking performed slightly better. They attribute this to the increased lock contention occurring when running all processes of an application simultaneously. The results in our paper are different. We find that gang scheduling does slightly better because of increased cache hit rates, especially when longer time slices are used. The Zahorjan study did not model cache behavior.

8.2 Gang Schedulers

One difference between the gang scheduling strategy as evaluated in this paper and those used in real systems is the requirement that the latter must deal with applications that do not neatly fit into a gang strategy. In real time-sharing systems, serial applications such as editors and compilers and parallel applications with few or many processes must be considered, and varying loads are the norm. Below we discuss some of these issues.

An early approach to gang scheduling was *coscheduling*, proposed by Ousterhout [16] for the Medusa system on the Cm* multiprocessor. The coscheduling scheduler tries to run all runnable processes of an application simultaneously as much as possible under a varying process and application load. The approach implemented in Medusa schedules entire applications in a round-robin fashion, with more than one application being run simultaneously if the sum of the number of processes of the applications does not exceed the number of processors. In this manner, applications with smaller number of processes do not result in many idle processors. Also, if there are not enough coscheduled processes to fill the processors, other processes slated to run at a later point in time are executed on the idle processors. Although these processes will probably not run long before being forced to wait, they nevertheless are an improvement over an idle processor (particularly since the target machine had no caches, and thus no cache corruption problems).

Another approach to gang scheduling was taken in the SGI multiprocessor system. There, users can determine whether an application should be gang-scheduled or normally scheduled. (This is similar to the gang scheduling approach proposed for the NYU Ultracomputer [9].) Until a gang-scheduling process is selected from the run-queue, the system operates using a normal priority-based central queue. When a gang process is scheduled, all other processes in the system are preempted. The processes of the gang application are scheduled and executed for a time slice. Other processes are then scheduled and the system proceeds normally. Since the process priorities are based on accumulated run time, with the process with least running time having the highest priority, the gang applications will not use more than their fair share of the computing resources. One drawback of the SGI scheme is that it can lead to undesirable behavior when the degree of multiprogramming is high and applications have varying numbers of processes. For example, if a gang scheduled application is running but not using all the processors, it is possible that an idle processor may pick a process from another gang application. This may lead to the latter application replacing the earlier gang-scheduled application in the middle of its time slice. Our current study corresponds to the coscheduling approach, but the SGI approach also needs careful evaluation.

To address the centralized-scheduler bottleneck of gang scheduling, Feitelson and Rudolph [10] have proposed *distributed* gang scheduling. Under this scheme, regular processors are controlled by a tree of dedicated control processors, with each control processor handling the scheduling in the subtree below it. A given application is gang scheduled onto a subtree of the system just large enough to run all processes in the application simultaneously. The gang scheduling of applications that are to be executed on distinct subtrees is independent and decentralized. Although they present some results from a synthetic workload, it is not clear at this time whether the extra hardware associated with the control processors is justified.

8.3 Other Scheduling Studies

The effect of processor affinity on multiprocessor caches has been studied before by Lazowska and Squillante [18] at the University of Washington. Using queueing theory techniques they modeled a multiprocessor system running multiple single process applications with different affinity-based scheduling policies. The suggested policies vary in amount of affinity and in load-balancing ability, essentially trading off affinity versus utilization. The policies include fixed scheduling, where each process is permanently assigned to a processor; work conserving scheduling, where a processor tries to pick a process that has not run on another processor since it ran in the scheduling processor; and minimum intervening scheduling, where a processor schedules the process it has executed most recently in terms of number of intervening processes. The results show the cache performance of the work conserving and minimum intervening schedulers to be similar to that of the fixed scheduler. Minimum intervening performed best under light loads, and the fixed scheduler performed best with heavy loads where load imbalance is not a problem. Since the work did not look at the cache characteristics of actual parallel programs, it is difficult to extrapolate what may be observed in a real environment. Our work on affinity complements their work by presenting results for real parallel applications in a more realistic machine environment.

Leutenegger and Vernon [13] also studied a variety of different scheduling policies; in their case the focus of was on the ability of schedulers to handle varying process loads. Their policies included normal round-robin scheduling, gang scheduling, process control, and another approach called *round-robin job scheduling*. Round-robin job scheduling schedules all processes of an application simultaneously, like gang scheduling, but adjusts the time slice length of the processes so that the total number of processor time allocated remains constant; that is, if an application has fewer processes than there are processors, the time quanta of those processes increase proportionally. The schedulers were compared based on the mean turnaround time of each application within a synthetic workload. Caches were not simulated, so the effects were the result of load balancing characteristics and handling of busy-waiting in the applications with synchronization. Process control and round-robin job scheduling were found to be superior because of their emphasis on fairness; applications with small numbers of processes were given an equal amount of processor time as those with more processes, decreasing the mean turnaround time.

9 Conclusions

In this paper, we have considered the performance of parallel applications on a multiprogrammed system. In particular, we studied how the choice of synchronization primitives and operating system scheduling policy affect a system's efficiency. The results were obtained using detailed simulations of four applications running on a shared-memory multiprocessor. The applications selected had considerable variation along several important dimensions, such as the granularity at which parallelism is exploited, the frequency of synchronization operations, and the spatial and temporal locality exhibited by the processes.

We first explored the performance of a simple priority-based scheduler while assuming that the applications use busy-waiting synchronization primitives. We found that the performance was very poor due to the large amount of time spent spin-waiting for processes suspended within the critical section. The average processor utilization was only 28% as compared to 74% achieved by a batch mode scheduler.

The first alternative strategy we explored was the use of blocking synchronization primitives. We found that blocking locks significantly improved the performance taking the average utilization up to 64%. The synchronization overhead was quite low, less than 2%, for all applications except Maxflow, which uses very fine grain synchronization. We also showed that it is important to implement the blocking synchronization primitives so that they busy-wait for a short time before the process is suspended. Otherwise, the number of context switches and the associated overhead can increase enormously for fine grain applications like Maxflow.

We found that a significant portion of the performance loss when using blocking locks, as compared to the batch mode scheduler, was due to worse cache hit rates. (Even if a preempted process is rescheduled onto the same processor on which it had been executing, intervening processes may have wiped out much of its data from the cache.) To improve cache hit rates we explored affinity scheduling. We found that when a small degree of affinity ($c = 0.1$) was added to the base priority scheduling, it improved the cache hit rates for all applications and increased the average machine utilization to 67%. With this small amount of affinity, the resources of the machine were still fairly distributed among the applications. When the importance of affinity was increased ($c = 0.5$), we found that the overall performance increased some more, but the allocation of machine resources

became very skewed. One of the four applications got more than 50% of the machine cycles, while another got only 12%. This indicates that affinity should be added to a scheduler only in a very controlled manner.

The second major strategy that we explored was gang scheduling. We presented results for time quanta of both 10ms (the default for the other strategies) and 25ms. The processor utilization achieved with the 10ms time slice was similar to that with blocking locks with affinity, about 67%. The main source of performance loss, compared to batch mode execution, was the cache flushes experienced between successive reschedulings of the same application. This affected the matrix multiply application most, since it has a large cache footprint. To decrease the frequency of cache flushes, we studied the performance with 25ms time slices. This increased the processor utilization to 71%, which is only 3% below that of batch mode. At a more global level, we expect to see applications that make effective use of caches and have large footprints (for example, blocked scientific applications) more frequently in the future. For gang scheduling to be effective for such applications, the time slice duration should be large enough so that the time to load the footprint into the cache is only a small fraction of it.

Finally, we explored the effectiveness of processor partitioning with process control. This strategy showed the best performance for all applications, even better than the batch mode, except for LU.[9] The overall processor utilization was 71%. The excellent performance of process control, compared to batch mode and gang scheduling, is primarily due to its use of fewer processes and processors per application. The use of fewer processors decreases the performance lost due to load balancing, lowers the synchronization costs, and increases the spatial locality of the application. The latter was most clearly demonstrated by Maxflow, where the cache miss time dropped from 32% for the batch mode to 24% for process control. Compared to gang scheduling, the performance was also better as there are no periodic cache flushes.

The results presented in this paper have corresponded to a small bus-based multiprocessor. It is interesting to consider the implications of the results for large scalable machines with a hierarchical structure and distributed shared memory [12]. It appears that gang scheduling and partitioning with process control still hold the most promise. While the centralization implicit in gang scheduling is a disadvantage, a major advantage is the more or less static runtime environment it presents to the software—each process is always run on the same processor. This static nature can be exploited to allocate storage from memory physically close to the processor thus reducing cache miss penalties. Such optimizations can be difficult with process control if the number of partitions and thus the processors allocated to the partitions is changing dynamically. The main advantages of process control over gang scheduling were listed in the previous paragraph. In addition, partitioned process control removes the centralization bottleneck of gang scheduling.

The focus of our current study has been the performance of long-running, CPU-bound parallel applications. We have mostly ignored issues of I/O and issues of response time for serial applications that must be considered in a practical implementation. Efficient handling of these issues can greatly complicate the implementation of scheduling strategies like gang scheduling and process control. Our next step is to implement the better-performing scheduling policies on a real system, examining their performance when handling a more realistic workload.

References

[1] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[2] Forest Baskett, Tom Jermoluk, and Doug Solomon. The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. In *Proceedings of COMPCON '88*, pages 468–471, February 1988.

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

[4] F. J. Carrasco. A parallel maxflow implementation. CS411 Project Report, Stanford University, March 1988.

[9] While the complete matrix being decomposed fits into the caches of twelve processors, it does not fit into the caches of three processors used with processor partitioning. This significantly increases the miss rate and lowers the performance. The performance of LU would have been the same as or better than that of batch mode, if a blocked LU program had been used.

[5] David R. Cheriton. The V distributed operating system. *Communications of the ACM*, 31(3):105–115, March 1988.

[6] Helen Davis, Stephen Goldschmidt, and John Hennessy. Tango: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90-439, Stanford University, 1990.

[7] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[8] Thomas W. Doeppner, Jr. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Brown University, 1987.

[9] Jan Edler, Jim Lipkis, and Edith Schonberg. Process management for highly parallel UNIX systems. In *Proceedings of the USENIX Workshop on UNIX and Supercomputers*, pages 1–17, 1988.

[10] Dror G. Feitelson and Larry Rudolph. Distributed hierarchical control for parallel processing. *IEEE Computer*, 23(5):65–77, May 1990.

[11] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 136–146, 1986.

[12] Dan Lenoski, Kourosh Gharachorloo, James Laudon, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of scalable shared-memory multiprocessors: The DASH approach. In *Proceedings of COMPCON '90*, pages 62–67, February 1990.

[13] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of SIGMETRICS '90*, pages 226–236, 1990.

[14] Ewing Lusk et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[15] Jeffrey D. McDonald and Donald Baganoff. Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[16] John K. Ousterhout. Scheduling techniques for concurrent systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[17] Edward Rothberg and Anoop Gupta. Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations. In *Proceedings of Supercomputing '90*, November 1990.

[18] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[19] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[20] Dominique Thiebaut and Harold S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.

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

[22] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Seminar on Performance of Distributed and Parallel Systems*, pages 455–472, December 1988.

[23] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. Technical Report 89-07-03, Department of Computer Science, University of Washington, July 1989.