Check for updates

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

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 different scheduling strategies. These include regular priority scheduling, coscheduling or gang scheduling, process control with processor partitioning, handoff scheduling, and affinity-based scheduling. We also explore 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 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 are shown to 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 small.

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 this is frequently not the case, since many applications may be executing simultaneously, 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.

When using multiprogramming, 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, 18], 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 [17, 9, 10] 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 [20]. Alternatively, partitioned scheduling may be used [3], 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. The process control strategy proposed in [23] addresses both synchronization and cache problems, by partitioning processors into groups and by dynamically ensuring that the number of runnable processes of an application matches the number of physical processors allocated to it.

While many scheduling strategies have been proposed, detailed evaluations using parallel applications have not been carried out — most of the previous studies [10, 13, 15, 19, 20, 24, 25, 26] have used synthetic workloads and analytical models or high-level simulations. The advantage of this approach is that a much wider range of options can be studied, though unfortunately, 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.

In this paper, we use detailed simulation studies to explore the interaction between scheduling strategies and synchronization methods, and their impact on the system throughput and cache performance of programs. 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. The use of blocking synchronization primitives, especially with limited spinning before suspending, significantly improves the performance of the applications — the average processor utilization

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM Ocopyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 089791-392-2/91/0005/0120...$1.50

goes up to 65%. The addition of processor affinity to the scheduling strategy results in a small but noticeable improvement in the cache behavior of all applications. 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 priority-based 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. The amount by which the timer is incremented on executing an instruction 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. 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 four applications that we have chosen are MP3D, LU, Maxflow, and Matmul. 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 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. Table 1 provides some basic statistics about the programs when each application is run individually with twelve processes on twelve processors. 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 ($\times 10^6$)	Total Refs ($\times 10^6$)	Shared Refs ($\times 10^6$)	Cache Misses ($\times 10^6$)	Speedup
MP3D	160	38	30 (80%)	2.7 (7%)	11.1
LU	93	17	17 (99%)	0.7 (4%)	18.1
Maxflow	194	48	29 (60%)	3.1 (6%)	10.0
Matmul	600	107	54 (50%)	4.0 (4%)	12.0

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. However, the scheduling strategies we evaluate should also provide reasonable response time as they all allocate

[1] We considered simulating a large scale multiprocessor with a hierarchical memory system, such as DASH [12], but eventually decided against it. Since scheduling strategies for simple uniform-memory access multiprocessors are not well understood so far, we decided that making the architecture more complex would be counterproductive.

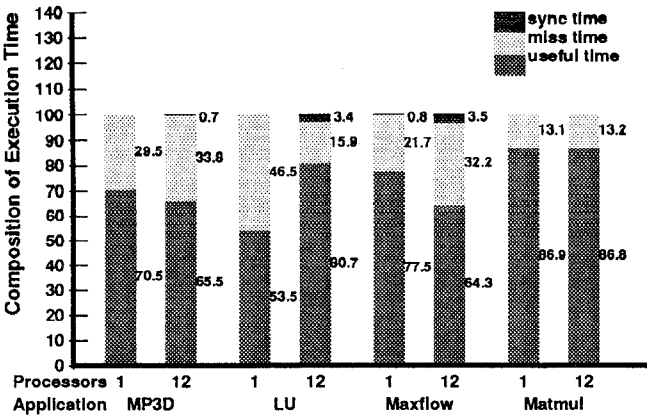

Figure 1: Application performance with 1 and 12 processors.

processor time fairly between applications. Where relevant, we provide additional measures of performance.

3.1 MP3D

MP3D [16] is a particle-based 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. The particles are statically divided among the processors to reduce overhead. The main synchronization consists of barriers between each time step.

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. As a result, the caches are expected to miss on each particle on each time step (as is expected in real-life runs of the program). The space cell data structure is 192 Kbytes in size. The space cells are also expected to bounce around between the caches of the processors and not result in very good hit rates.

For MP3D, Figure 1 shows that the processor utilization for the uniprocessor version is only about 71% because of the high cache miss rate (7%). The processor utilization drops even further to 66% for the parallel version, because of an increase in the miss rate for the space cells. The overall speedup for the application is quite good though, approximately $12 \cdot (0.66/0.71)$, 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. 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 (54% versus 81%). 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 uses 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 3% 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 removing 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. 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. 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 22% to 32% as nodes and edges bounce between the processors' caches. The time spent performing synchronization also increases by about 3% 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.

[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.

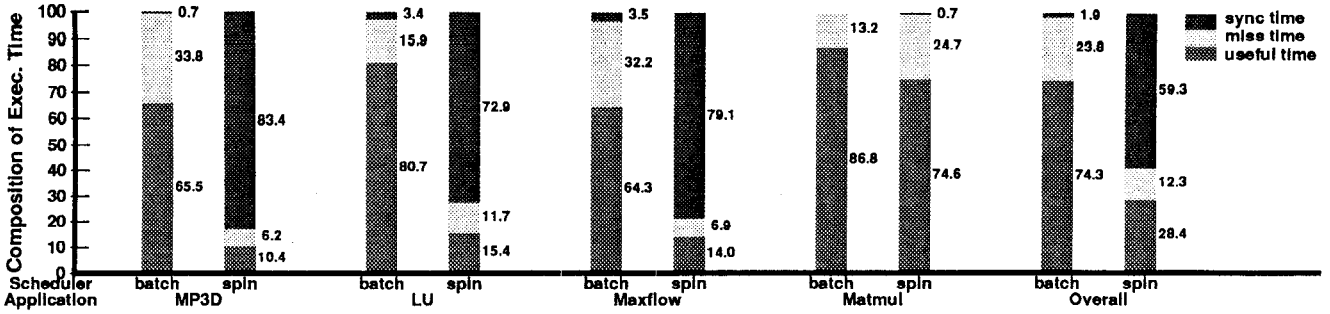

Figure 2: Comparison of batch mode and regular priority scheduling.

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. 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.

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 inversely proportional to recent CPU usage. To compute recent CPU usage, past processor usage is exponentially decayed with age. Every 6.25 million cycles of wall clock time (250ms on a 25MHz processor), CPU usage of each process is decayed to 0.66 of its existing value. 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 and under the scheduling strategy described above. In the latter 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.[3] Long before LU finishes, the behavior of MP3D, Maxflow, and Matmul has reached a steady state, so the distortions introduced in the results are minimal.

As might have been expected [9, 23, 24], the degradation in system performance with priority scheduling and busy-waiting locks is extreme. The overall processor utilization is down to 28%, compared to 74% when the applications are run in batch mode. The poor performance is caused by the interaction of the preemptive scheduling of processes with the use of busy-waiting synchronization primitives by the applications. The average time spent in the busy-loop goes up from 2% for the batch mode to 59% when all processes are time multiplexed together.

Looking at the performance of individual applications, for MP3D, the utilization drops from 66% to 10%. This happens 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. During the particle-move phases, the processor utilization is around 60%, which is reasonable. Between particle-move phases, however, the processes must pass through a series of seven barriers, where the processor utilization is close to zero.

For LU the utilization drops from 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 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 87% to 75% solely due to cache effects. The cache miss rate goes up from 4% to 8% since each Matmul process, whenever it is rescheduled, has to reload the block of matrix with which it was previously working.

As the above data shows, cycles wasted due to busy-waiting are the primary cause of poor performance. 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 spent 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 often not 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, if processes within critical sections may be preempted 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 [24, 25].

For the results reported in this section, we replaced all busy-waiting locks in the applications with blocking locks. The scheduling strategy still maintains a single run queue sorted by

[3] 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.

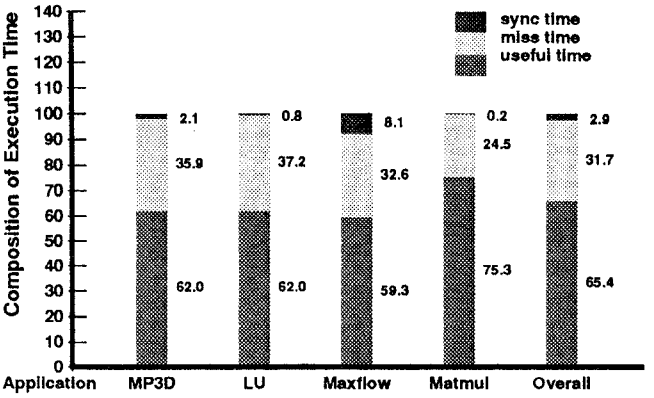

Figure 3: Performance with blocking locks.

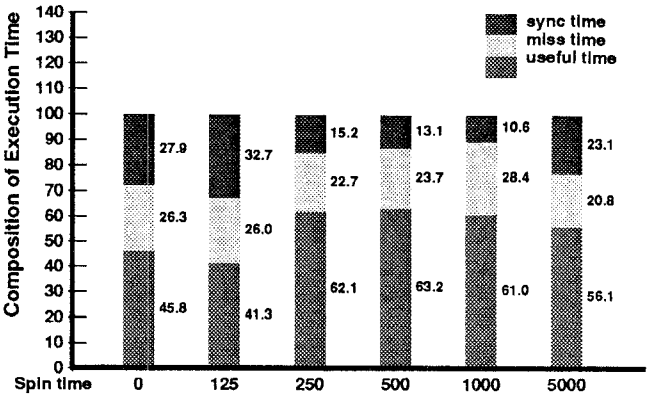

Figure 4: Comparison of Maxflow performance with varying amount of spin time before blocking.

priority. In our implementation of blocking locks, a process first spins for up to 500 cycles (equal to the cost of context switching to another process) trying to obtain the lock. 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 running on another processor. (The basic idea was proposed by Ousterhout [17].) 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), one or all blocked processes are moved to the run queue.

Figure 3 shows the performance of the applications when using priority scheduling with blocking locks. The results show a great improvement in performance with the overall processor utilization going up from 28% for busy-waiting (see Figure 2) to 65% for blocking locks. All applications gain significantly, except for Matmul, which changes little since it has no synchronization. The performance with blocking locks, however, is still lower than the 74% utilization achieved under the batch scheduler. This is primarily due to the higher cache-miss rates experienced under priority scheduling. We note that since the priority scheduling algorithm uses *recent* CPU usage (and not *total* CPU usage) for computing priorities, all applications do not get an equal share of CPU time when blocking locks are used. For the results in Figure 3, the MP3D, LU, Maxflow, and Matmul applications got 29%, 14%, 27%, and 30% of total machine time respectively. The LU application got the least time since its processes block for relatively long durations, thus giving the other applications' processes time to decay their CPU usage. In computing overall machine utilization (the rightmost bar in Figure 3), the individual utilizations are weighted by the fraction of machine time each application got.

When using blocking locks, one might expect that the overhead due to spinning and context switching (shown as synchronization time on the bars) would be large. However, it remains below 2.5% for all applications except for Maxflow. The overhead for Maxflow is about 8% since it uses 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 cost 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. Our compromise was to have the process spin for an interval equal to the context-switch time (500 cycles) before blocking. This guarantees that the performance will be within a small constant factor of the optimal blocking strategy.

In Figure 4 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 be more sensitive to such variations. The data was gathered with the Maxflow application running in isolation with twelve processes on three processors. The six values for the spin time that we consider are 0, 125, 250, 500, 1000, and 5000 cycles. From the bars we can see that the corresponding processor utilizations are 46%, 41%, 62%, 63%, 61%, and 56%, respectively. The context switch rates, that is, the number of context switches per time slice per processor, are 119, 113, 40, 17, 6, and 7 respectively. The data make the disadvantages of allowing no spin time before blocking clear. The data also show how too little spin time before blocking can hurt performance rather than helping it.

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 this subsection, we explore the performance of *handoff scheduling*, an alternative strategy for deciding which process to relinquish the processor to on blocking. Handoff scheduling is based on an approach used for remote procedure calls in Mach [3] and Topaz [21], where a client process calling a server process may "hand off" its processor to the server. When the server responds, the processor is returned to the client process. The basic idea is that a blocking process should be able to turn over the remainder of its time slice to another process that may expedite the event for which the blocking process is waiting.

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 based on priority. 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

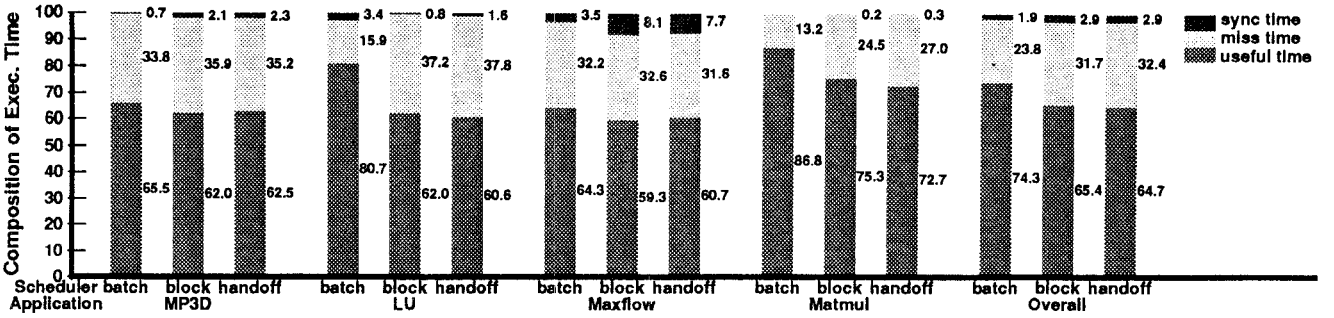

Figure 5: Comparison of batch scheduling and normal and handoff scheduling with blocking locks.

"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 that owns that column, if that process is not already running.

Figure 5 shows the performance of a system with blocking locks and handoff scheduling. Compared to regular blocking scheduling, the overall processor utilization with handoff goes down from 65.4% to 64.7%.[4] 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, in general, the early scheduling of the process holding the lock did not reduce the processor cycles spent synchronizing, except slightly for Maxflow. This is because regular blocking locks already eliminate virtually all busy-waiting time, and there is enough concurrency in the system that 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 3.9 switches per time slice per processor for normal priority scheduling to 4.3 for handoff scheduling. This is because handoff scheduling runs a process only for the remaining portion of the time slice.

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 to another process from the same application rather than to a random process, possibly from a different application. The benefits depend on the degree of sharing between processes of the same application. While the miss rate increases slightly for LU and Matmul, it decreases slightly for Maxflow and MP3D. For LU, the miss rate increases because there is very little actively shared data between the processes (each process is responsible for a distinct set of columns) and the context switch rate goes up with handoff from 1.9 to 3.4 per time slice. The reasons for Matmul are similar to that for LU (each process has a distinct active block that it wants to have in the cache), with the context switch rate going up from 1.0 to 1.3. For Maxflow and MP3D the context switch rate remains almost the same with handoff, and since these applications have actively shared data between processes (nodes and edges for Maxflow and space cell array for MP3D), 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 offer higher performance as there are always other ready processes to be run. Similarly, if the concurrency in the system is low so that there are idle processors, then again handoff scheduling will 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 system throughput is concerned, the usefulness of handoff seems limited.[5]

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 [20]. 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 will 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 size of the application *footprint* [22], 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 a 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 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 (or even only one process) wipe out most of the 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, Squillante and Lazowska [20] 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. The second question is how to balance between fairness to all processes (which helps improve response time) and affinity (which helps improve throughput). Here again

[4]The fraction of machine time devoted to MP3D, LU, Maxflow, and Matmul under handoff was 25%, 20%, 26%, and 29% respectively.

[5]Regarding the effect of handoff on mean response time, we do not expect a substantial change unless the application uses very fine grain synchronization. This is primarily because the underlying priority-based scheduler is reasonably fair in its allocation of CPU time to applications. As a result, since the system throughput is almost the same with or without handoff, and since CPU time is similarly split between the applications with or without handoff, response time is not expected to change significantly.

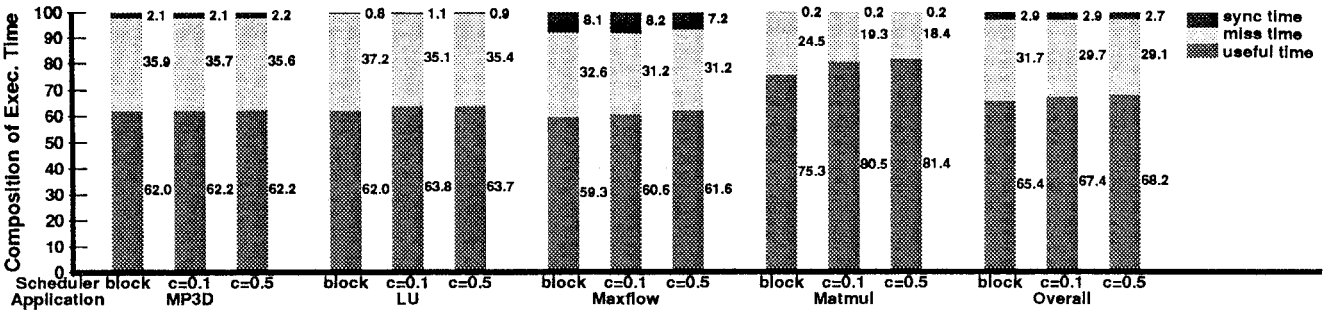

Figure 6: Comparison of blocking scheduling and affinity scheduling with $c = 0.1$ and $c = 0.5$.

there is a range of possibilities. For example, one extreme that stresses affinity is batch scheduling, while another extreme that stresses fairness is vanilla priority scheduling. The intermediate strategy that we have adopted for the current affinity study is described below.

For our affinity studies we again use blocking locks and priority scheduling, but priority is computed differently than before. The priority is assumed to be inverse of $[T + (c \cdot l/ldfactor)]$. In the formula, T is the recent CPU usage of the process (this factor ensures that some semblance of fairness is maintained); c is a weighting factor indicating the importance to be given to affinity (for example, if $c = 0$ then we are back to normal blocking scheduling); l is the duration since this process last ran on the processor under consideration (the opposite of affinity); and $ldfactor$ is the average number of runnable processes per processor (this ensures that affinity is not given excessive weight when load is high). To avoid starvation, the value of l is limited to 20 time slices. (In early experiments, where we did not bound l, there were serious problems with fair allocation of processor time.) The value of l was also bounded because, beyond a point, the duration since the process last ran on a processor does not matter. 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 scientific applications to behave like these two.

Figure 6 shows the performance of applications with blocking locks and affinity scheduling with $c = 0.1$ and $c = 0.5$. Without affinity, the probability that a process is rescheduled on the same processor it last ran is about 22%. With affinity scheduling and $c = 0.1$ and $c = 0.5$ this probability goes up to 36% and 44% respectively, a significant increase. Similarly, without affinity the median number of intervening processes that are scheduled on a processor between two successive runs of the same process is 13. With affinity of $c = 0.1$ and $c = 0.5$ this number drops to 9 and 9 respectively. The effect of affinity is uniformly positive, with each application obtaining slightly higher processor utilization. For $c = 0.5$ processor utilization for MP3D goes up from 62.0% to 62.2%, for LU from 62.0% to 63.7%, for Maxflow from 59.3% to 61.6%, and for Matmul from 75.3% to 81.4%. The overall increase in utilization is about 2.8%. We tried higher values of c, but they did not increase the benefits significantly. We also tried a lesser degree of multiprogramming, where only the LU and Maxflow applications were run, but the benefits were still small.

For MP3D, the benefits of affinity are small because most cache misses are steady state misses and not initial loads of a footprint into the cache. For LU the footprint is reasonably large, but the benefits are still 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 from slightly reduced cache miss and synchronization time. For Matmul, the benefits are the most since 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.

In summary, while the gains of affinity are small, they are consistently positive for all applications. The reasons for the small gains are either 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). It is interesting to note how the final benefits obtained from affinity are governed by the rich and complex interactions between the various applications running at the same time. It is in this aspect we believe that our application-based study complements the analytic-model based study done by Squillante and Lazowska [20].

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 [17]. 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 [12], a process may allocate storage from a memory that is physically close by to reduce cache miss penalty. Such optimizations cannot be used in the schemes described in earlier sections, as there processes and processors are not as 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 [10]. 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

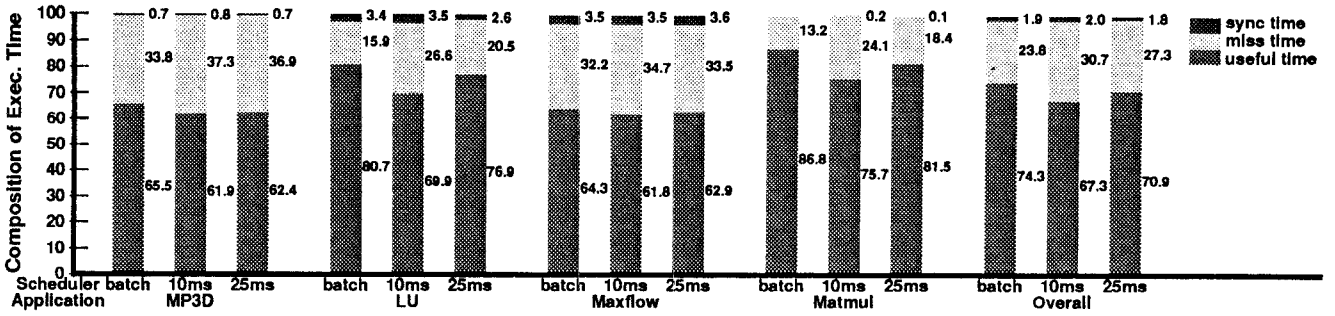

Figure 7: Comparison of batch scheduling and 10ms and 25ms time slice gang scheduling.

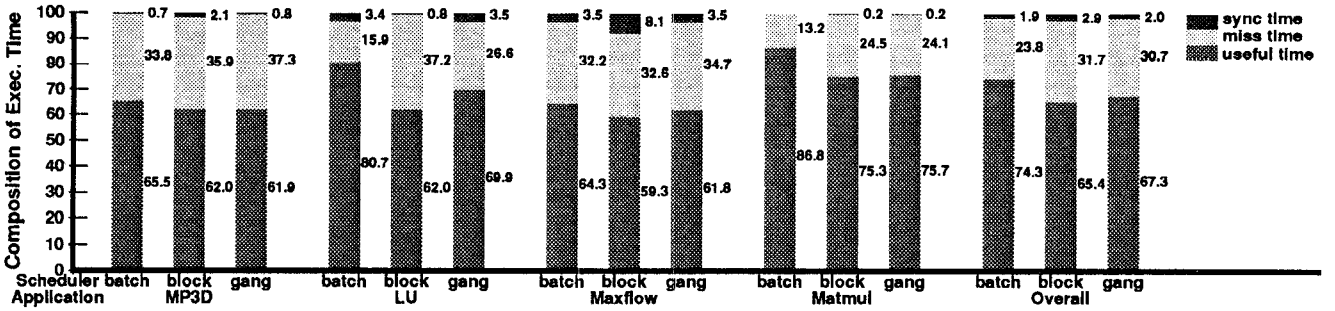

Figure 8: Comparison of batch, blocking, and gang scheduling.

scheduling can lead to fragmentation of processors when there are applications that do not need all of the processors in the system, but do not leave enough free processors to fit all of the processes of another application [17]. 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 7 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 65% 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 11% 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 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 7. 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 over 10ms gang scheduling were observed for LU (7%) and Matmul (6%). An increased time slice duration 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. This performance is shown in Figure 8. The performance is approximately the same for MP3D. Gang scheduling results in 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 (3.9 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 processes relinquish their processors whenever the pivot column is not ready, but the frequent blocking results in substantially increased miss time (11% more compared with gang). For Maxflow, gang scheduling again wins over blocking locks by 3%. 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 8%). Finally, for Matmul the performance is approximately the same for both cases. Gang scheduling does slightly better 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 the motivation for using blocking locks is reduced. Without blocking locks, the frequency of context switches is cut down, and applications that rely on reuse of cached data can perform well for most of a time slice. Finally, an increase in the duration of the time slice is shown to improve gang scheduling performance noticeably.

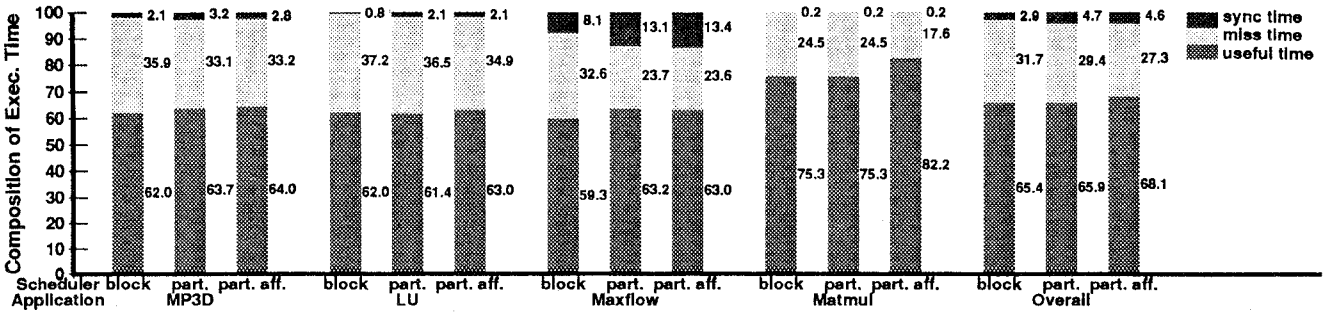

Figure 9: Comparison of blocking, partitioned blocking, and partitioned blocking with affinity scheduling ($c = 0.5$).

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 [23].) 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 9 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 65.9%, only 0.4% higher than without partitioning. The performance increases slightly for MP3D due to the fact that only related processes with shared data are scheduled on the processors. The performance decreases for LU because its processes have mostly 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.

Maxflow also shows interesting behavior. The cache miss time drops by about 10% while the synchronization time increases by 5%. 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 6.5 switches per time slice per processor for the non-partitioned case to 17.5 for the partitioned case. Finally, the performance for Matmul remains the same, since the context switch rate remains the same at about 1 switch per time slice per processor and the active data sets of the Matmul processes have little in common.

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 reduced time between repeated executions of a process on the same processor, even without affinity.

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* [23], 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, 5], where 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 [23]. In this subsection we look at the cache and synchronization behavior of process control in a more controlled setting.

Figure 10 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

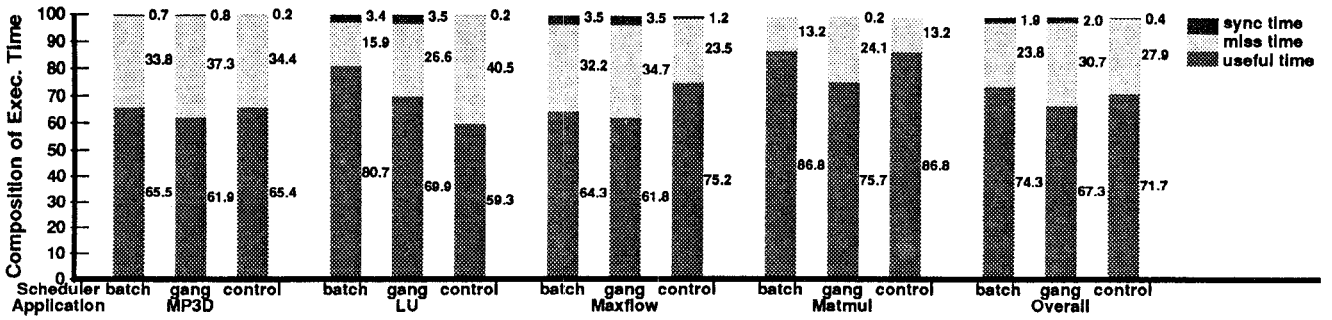

Figure 10: Comparison of batch, 10ms gang, and process control scheduling.

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 block-based algorithm 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. Most parallel applications give sublinear speedups with increasing number of processors. For example, an application may give 7-fold speedup with 8 processors but only 12-fold speedup with 16 processors. This can be due to several reasons, including load balancing problems, greater memory interference, greater contention for locks, etc. Consequently, when a large number of processors are used to run an application, the processor efficiency is less than when fewer processors are used. The result is that in the presence of multiprogramming, gang scheduling and batch scheduling may exhibit low processor efficiency, since they use all the processors in the machine for each application.[6] In contrast, process control dynamically partitions the machine into several smaller machines, one per application, with each smaller machine providing higher processor utilization.

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 (with time slice of 10ms) 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

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 propose an alternative solution [9], where individual processes can temporarily prevent their own preemption, in the form of a hint to the scheduler, while they are holding a lock. While such a mechanism helps solve the busy-waiting problem for locks, it does not improve the performance of more complex synchronization mechanisms like barriers and events.[7] The approach also does nothing to reduce the frequency of context switches or to lessen the cache corruption problem. The researchers do not present any quantitative data on the effectiveness of their proposal.

Zahorjan et al. [24] 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 busy-waiting cycles by not rescheduling processes that are waiting for a held lock. A third approach combines the previous two approaches. Using analytic models, the researchers found that all three approaches work well in avoiding idle spinning time. The second approach to lock contention, avoiding rescheduling waiting processes, also improves the performance of barriers under light multiprogramming loads. While it would have been interesting to contrast the results from the above schemes to those obtained with blocking locks in this pa-

[6]Even if the speedup is sublinear, the gang scheduling approach encourages that 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.

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

per, it was unfortunately not possible to do so due to different underlying models and assumptions.

The handoff primitives used in our study are similar to those found in CMU Mach [3] and DEC Topaz [21]. These systems use handoff scheduling to provide efficient remote procedure call (RPC) implementations. Preliminary results from Mach [3] show that handoff scheduling can significantly reduce the latency of round-trip synchronization, especially if the system load is high. In this paper, we did not focus on the round-trip latency but on the overall throughput of the system. Because our base system used blocking locks, we found that there were only minimal cycles spent in busy-waiting, and that handoff showed almost no benefits in throughput under high load conditions.

On the topic of gang scheduling, while much effort has been spent on how such schedulers are to be implemented in practice, little data is available characterizing their benefits. Below we discuss some studies that address the implementation issues and present results from analytical modeling.

An early approach to gang scheduling was *coscheduling*, proposed by Ousterhout [17] for the Medusa system on Cm*. The coscheduling scheduler tries to run all runnable processes of an application simultaneously under a varying application load. The approach implemented 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.

Another approach to gang scheduling has been taken in the Silicon Graphics' IRIX operating system. In IRIX, the user can determine whether an application should be gang-scheduled or normally scheduled. (A similar approach was 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 running processes are preempted. The processes of the gang application are then scheduled and executed for a time slice. The Silicon Graphics' approach is attractive because it nicely integrates traditional scheduling and gang scheduling.

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. It is not clear at this time whether the extra hardware associated with the control processors is justified.

Moving to performance studies, Zahorjan et al. [25] have compared gang scheduling with partitioned round-robin scheduling, the latter with spinning locks and blocking locks. 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 that occurs when all processes of an application are running 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.

Leutenegger and Vernon [13] have also studied several different scheduling policies, including round-robin scheduling, gang scheduling, process control, and another approach called *round-robin job scheduling*. The round-robin job policy schedules all processes of an application simultaneously, like gang scheduling, but adjusts the time slice length of the processes so that the total amount of processor time allocated remains constant. Thus if an application has fewer processes than there are processors, the time quanta of those processes is increased proportionally. The schedulers were compared based on the mean turnaround time of each application using a synthetic workload, where the number of processes per application and their computing requirements are highly variable. Caches were not simulated, so the effects were the result of load balancing characteristics and handling of busy-waiting synchronization in the applications. Process control and round-robin job scheduling were found to be superior because of their emphasis on fair allocation of CPU resources — applications with small numbers of processes were given an equal amount of processor time as those with more processes, decreasing the mean turnaround time. Interestingly, the results of this paper, while focusing on system throughput instead of response time, also find process control and gang scheduling to offer the highest performance.

The effect of processor affinity on multiprocessor caches has been studied before by Squillante and Lazowska [20] 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 varied in amount of affinity and in load-balancing ability, essentially trading off affinity versus utilization. The policies included 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 on the scheduling processor; and minimum intervening scheduling, where a processor picks and runs the process it had executed most recently in terms of number of intervening processes. The results showed that minimum intervening performed best under light to moderate loads, and the fixed scheduler performed best with heavy loads where load imbalance was not a problem. Our study is different in that it uses real parallel applications (with processes sharing data and interfering with each other) rather than a synthetic workload consisting of identical single-process applications. We also model caches in much greater detail and use a different affinity function. To the extent that the results of the two studies can be compared, it appears that the benefits of affinity are smaller with real parallel applications since averages and statistics do not always apply (for example, even a single intervening time slice of Matmul can wipe out most of the previous contents of the cache).

9 Summary and 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. 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 pro-

cessor 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 65%. 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.

We found that a significant portion of the performance loss when using blocking locks, as compared to the batch mode scheduler, was due to lower cache hit rates. To improve cache hit rates we explored affinity scheduling. We found that when a small degree of affinity ($c = 0.5$) was added to the base priority scheduling, it improved the cache hit rates for all applications and increased the average machine utilization to 68%.

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 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 (e.g. 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 the time slice.

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, aside from the exceptional case of LU. The overall processor utilization was 72%. The excellent performance of process control, compared to batch mode and 10ms 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 for highly parallel machines, 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, most of which carry over to highly parallel machines, 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 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 and to examine their performance when handling a more realistic workload.

Acknowledgments

We would like to thank Kourosh Gharachorloo and the referees for detailed comments on an earlier draft of this paper. We would also like to thank Helen Davis and Steve Goldschmidt for creating and supporting Tango, and for helping us modify it for this paper. The research was supported by DARPA contract N00014-87-K-0828. Anoop Gupta is also supported by a NSF Presidential Young Investigator Award.

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, 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.

[5] Rohit Chandra, Anoop Gupta, and John Hennessy. *COOL: A Language for Parallel Programming*. Research Monographs in Parallel and Distributed Computing. The MIT Press, 1990.

[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. Ultracomputer Note 136, New York University, 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 et al. Design of scalable shared-memory multiprocessors: The DASH approach. In *Proceedings of COMPCON '90*, pages 62–67, 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] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of SIGMETRICS '88*, pages 104–113, 1988.

[16] 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.

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

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

[19] Kenneth C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of SIGMETRICS '89*, pages 171–180, 1989.

[20] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity in shared-memory multiprocessor scheduling. Technical Report 89-06-01, Department of Computer Science, University of Washington, June 1989.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] John Zahorjan and Cathy McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of SIGMETRICS '90*, pages 214–225, 1990.