Spinning Versus Blocking in Parallel Systems with Uncertainty

John Zahorjan and Edward D. Lazowska

Department of Computer Science
University of Washington

Derek L. Eager

Department of Computational Science
University of Saskatchewan

February 1988

Abstract

In waiting for an event on a parallel machine, a thread of control may either spin (busy wait) or block (relinquish the processor). The appropriate mechanism depends on the relationship of the expected spin time to the context switch time on that machine.

If the programmer has accurate information about the behavior of an application, the choice between spinning and blocking can be made relatively easily. This might be the case, for instance, when a parallel machine is dedicated to a single, well understood application. However, in the presence of uncertainty, the choice of mechanism is more difficult.

In this paper we examine the choice between spinning and blocking in environments characterized by two kinds of uncertainty: multiprogramming, where the applications programmer does not have control over which threads are running at any point in time, and data-dependent programs, where expected running times can depend heavily on input data. We compare the loss incurred by spinning in these two environments to that in systems running a single, "well-behaved" application. Our goal is to determine how multiprogramming and data-dependent behavior affect expected spin time, and so complicate the job of selecting the right mechanism.

We examine the base, multiprogrammed, and data-dependent environments for two different situations: lock acquisition for mutual exclusion and for barrier synchronization. Using simple analytic models we conclude that for the case of lock acquisition neither multiprogramming nor data-dependent behavior significantly increase the expected spin time, and thus do not complicate the choice of mechanism. However, for barrier synchronization both kinds of uncertainty lead to sharply increased spin times, and thus must be taken into consideration when choosing between spinning and blocking.

Index Terms – Multiprocessors, locking, performance, parallel software, parallel computing.

This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, Digital Equipment Corporation (the External Research Program and the Systems Research Center), and the Natural Sciences and Engineering Research Council of Canada. A portion of this work was done while Zahorjan was on sabbatical leave at Laboratoire MASI, University of Paris VI.

Authors' addresses: John Zahorjan and Edward D. Lazowska, Department of Computer Science and Engineering FR-35, University of Washington, Seattle, WA 98195; Derek L. Eager, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada.

1. Introduction

When a thread of control on a parallel machine must wait for some event before proceeding, it may be reasonable for the thread to spin (or busy wait) – that is, to sit in a tight loop continuously checking for the required condition. The time spent spinning is overhead, and since the processor is occupied and cannot be allocated to another thread, the effective processing rate of the system is decreased. An alternative to spinning is blocking – that is, relinquishing the processor. The context switch time required to block also is overhead, so blocking too decreases the effective processing rate of the system.

The appropriate choice between spinning and blocking depends on the relationship of the expected spin time to the context switch time. This choice is not always clear, and a mistake can have major performance implications. For example, one field test release of the DYNIX operating system for the Sequent multiprocessor [Beck et al. 1987] included the substitution of blocking for spinning in a single routine as a "performance enhancement". Under high loads, this change in fact caused a severe performance degradation, something that was first noticed by a Sequent competitor and used as the basis of an advertising campaign [Rodgers 1986].

Although adaptive mechanisms are possible (see, for example, [Ousterhout 1982] and [Lo & Gligor 1987]), the decision of whether to spin or to block is most often made statically at program creation time by the programmer. In highly controlled environments where the programmer has accurate information about the expected spin time, this decision may be straightforward. For example, when the parallel machine is dedicated to a single application that uses locks for mutual exclusion, the programmer may know that a particular lock is held infrequently and for only a very few instructions. Spinning would be the clear choice in this case. Similarly, parallel solutions of large numerical problems often are obtained by partitioning the problem among a number of threads equal to the number of processors. Spinning is the clear choice here, too, since there are no other threads that could run.

The choice between spinning and blocking is not so straightforward when the expected spin time depends on run-time factors. How to make this choice in the presence of such uncertainty is the subject of our paper.

We consider two typical situations in which threads must wait, requiring that either spinning or blocking be employed. The first situation involves waiting because of competition among threads. Here we assume a set of largely independent threads that use a lock to provide mutual exclusion when accessing some resource. When a thread wanting the resource finds the lock in use, it waits until the lock becomes free. The second situation involves waiting because of cooperation among threads. Here we model a set of threads that attempt to synchronize at a barrier. Each thread reaching the synchronization point (that is, the barrier) waits until all other threads have also reached the barrier.

Our "baseline case" is a controlled environment in which the choice between spinning and blocking is straightforward: a single parallel application running on a dedicated multiprocessor with as many processors as there are threads of control.

We consider the effect on this choice of two run-time dependencies. The first is multiprogramming. As parallel architectures become more common, parallel machines and parallel algorithms increasingly will be the choice for general-purpose computing. Clearly, multiprogramming of user jobs is a requirement in this environment. However, because the programmer does not have explicit control over the scheduling decisions made on a multi-user multiprogrammed machine, it is impossible to know which threads of a parallel application will be running at any particular time. Thus, the spin time can be highly variable, depending on whether or not the thread that will eventually generate the event being waited for is currently allocated a processor.

The second source of uncertainty that we consider is data-dependent software. Here we assume that the thread that eventually will generate the awaited event sometimes completes quickly and sometimes runs for a long time. Its behavior depends on its state and the data with which it is presented, and so is not predictable at the time that the application is coded.

The goal of our work is to determine how spin time is affected by these two forms of uncertainty when compared with the baseline case. If the expected spin time is roughly the same in all three situations, the decision between spinning and blocking can be made as if the application were running in a well controlled environment. Since programmers are already dealing with this problem in that environment,

this would mean that no new special procedures are required. On the other hand, if the spin time can be significantly lengthened in environments with uncertainty, the programmer's task is greatly more difficult, since it will be necessary to estimate at program creation time parameters that become known only at run time.

Our study is conducted using analytic models validated via simulation. We examine the degradation arising from using spinning under uncertainty relative to that arising using spinning in a controlled environment. We decided against constructing models of performance under blocking with which to compare our spinning models. Information concerning the relative merits of spinning and blocking can be drawn from the spinning models alone, and the results obtained from models of this single type are less sensitive to the precise modelling assumptions made, since any inaccuracies appear consistently. Thus our performance comparisons (if perhaps not the absolute performance values) will be accurate. (See, for example, [Dubois & Briggs 1982] as a contrast in the complexity and flexibility of modelling approaches.)

In Section 2 we describe more precisely the models employed in our comparisons, and we outline briefly the analytic approach. A more detailed discussion of the analysis is found in Appendix A. Section 3 is a discussion of the results for waiting due to lock contention. Section 4 presents the results for waiting due to barrier synchronization. Section 5 contains our conclusions.

2. The System Models

Clearly, a useful performance model must embody enough of the details of the system it represents so that the model's behavior parallels that of the system. At the same time, "unnecessary detail" should be avoided, and the model kept as simple as possible [Lazowska et al. 1984], for at least two reasons. First, simplicity aids in understanding the interaction of the model parameters. A model with many parameters implies an enormous parameter space and consequently a potentially unmanageable set of experiments and results to explore the significance and interaction of those parameters. Second, a simple model is usually more quickly analyzed than a complex one, and so eases the practical burden of running the necessary experiments.

We have constructed two different but similar models, one for lock contention and the other for barrier synchronization. Each model accommodates the three environments (baseline, multiprogramming, and data-dependence) in a natural way. This is important because our goal is to compare spin times between environments, so consistency across those boundaries lends confidence that the comparison is valid.

Performance predictions for our models may be obtained by either simulation or numerical analytic techniques. In fact, we have developed and run software for both approaches. However, all of the results given here are taken from the analytic solutions. Simulation was used only for a sample set of cases with the sole intention of verifying that the analytic software was functioning correctly. The analytic approach is preferable to simulation in this application because the results it provides are exact equilibrium performance measures (rather than stochastic estimates and confidence intervals) and because in general the analytic software is able to obtain results much more quickly than the simulation software.

2.1. Lock Contention

Our lock contention model consists of P identical processors and J threads. We model explicitly the contention for a single lock. Each thread is in one of three states: computing, spinning, and critical section. A thread computes for an average of T time units between attempts to obtain the lock.[1] When a thread requires the lock, if the lock is free the thread immediately acquires it. A thread holding the lock uses it for an average of L time units, then releases it and returns to the computing state. If the lock is not free when requested, the thread spins until the lock is released. If multiple threads are spinning when the lock is released, one of these threads is chosen at random to acquire the lock next.

---

[1] In reality, this means that the thread occupies the processor for an average of T time units between requests. The thread may be performing useful work or spinning on another lock or for some other reason during this time.

For the baseline case of a controlled system, the model is exactly as described above with $J = P$, that is, one thread per processor. This represents the situation in which the machine is dedicated to a single application at a time, and thus presents the application programmer with the least amount of uncertainty regarding the behavior of the software.

A multiprogramming environment is reflected in the model in two ways. First, there are more threads than processors ($J > P$), thus reflecting the fact that, in a multiprogramming environment, an application might at times have more threads than it has been allocated processors. Note that it is most appropriate to increase the number of threads rather than decrease the number of processors when deriving a multiprogramming instance of the model from a baseline instance, because the inherent degree of lock contention is thereby kept constant (as this depends on the total instruction delivery rate as determined by the number of processors, not on the number of threads), and thus any changes in performance can be attributed solely to the effects of multiprogramming.

At any one time, P threads are *scheduled* (allocated processors) and $J-P$ are *unscheduled* (without processors). The second way in which multiprogramming must be reflected in the model is the introduction of a scheduling rule that controls which threads fall into each category. We define a parameter Q that represents the mean scheduling quantum. Each scheduled thread is allowed to use its processor for an average of Q time units before it is unscheduled. (The scheduling of threads on any particular processor is independent of that on all other processors.) When a thread's quantum expires, its processor is assigned at random to a currently unscheduled thread.

This random scheduling is the major simplification of our model, as the replacement policy in a real system is more likely to be FCFS in nature. However, note the mean time that a thread remains unscheduled between successive uses of a processor is identical under random and FCFS replacement, as is the mean amount of computing provided to each thread per time unit. Thus, intuitively we expect the mean performance measures observed under the two scheduling disciplines to be similar.

The primary motivation for assuming random replacement is that it enormously simplifies the model state space. In particular, our model has $2P(J-P)+2J-P+1$ states while an identical model with FCFS scheduling has more than 2^{J-P}. This simplification not only allows results to be obtained more quickly, but also permits the analysis of models with larger numbers of threads and processors than could be examined otherwise.

To model data-dependent behavior we again let J equal P, that is, the model is not multiprogrammed since we wish to isolate the particular effect of data-dependence. However, here we let the lock holding time be highly variable. In particular, with probability $1-p$ a thread acquiring the lock releases it instantly, and with probability p holds it for mean time $\frac{L}{p}$. This results in a mean holding time of L, just as in the baseline case, but with much greater variance.

2.2. Barrier Synchronization

The model of barrier synchronization is quite similar to the lock contention model. In the baseline case there are P processors and $J = P$ threads. Each thread is in one of two states: computing or spinning. A thread computes for an average of T time units before reaching the barrier. If not all other threads have already reached the barrier, it begins to spin. When the last thread reaches the barrier, all threads return to the computing state.

For the multiprogramming environment, we keep constant at P the number of threads involved in the barrier, but add K additional threads. These threads are always in the compute state, but their presence on the processors interferes with the progress of the P "barrier threads". As previously, Q is the mean scheduling quantum and random replacement among the $J+K-P$ unscheduled threads is used as the scheduling discipline.

It is important in this model that we keep the number of barrier threads the same as in the baseline case. The mean time to reach a barrier increases naturally with the number of threads involved in the synchronization. Since we are trying to isolate the effect of multiprogramming, it would not be suitable to simplify the model further by having all $P+K$ threads be involved in the barrier, as it would be difficult to separate the increased spin time due to multiprogramming from that due to the increase in the number

of barrier threads. (Note that this is in contrast to the lock contention situation, where the inherent lock contention is determined not by the number of threads but by the instruction delivery rate as determined by the number of processors.)

To reflect data-dependent behavior in the model, K is zero as in the baseline case (i.e., there are only the $J = P$ barrier threads), but there is much greater variance in the compute time required before a thread reaches the barrier. With probability $1-p$, a thread computes for zero time units before reaching the barrier, while with probability p it computes for an average of $\frac{T}{p}$ time units. This maintains the average compute time of T time units, as in the baseline case, but greatly increases the variance.

A more detailed discussion of the analytic formalities of these models can be found in Appendix A.

2.3. Choosing Parameter Value Settings for the Model

A problem that must be confronted immediately in attempting to compare spin times among the three environments (base, multiprogramming, and data-dependent behavior) is how to set the model parameters. Unfortunately, there does not exist currently an extensive set of measurement data of real systems on which to base the parameterization, nor even a compelling folklore about what range of values are reasonable. We have therefore run a large number of experiments with parameters varying over a wide range. The results presented here represent a subset of those experiments that we believe fairly represents the "typical" behavior of the systems.

There are two kinds of parameters that must be given values: those involving time (T, L, and Q) and those involving size (P, J and K).

Considering first the parameters involving time, we have chosen to let the compute time T be the unit of time against which all other time parameters are measured; that is, we have set $T = 1$.

In all of our experiments we let the lock holding time L vary over an extensive range, in particular from 0.01 to 0.5. At the low end this represents extremely low lock contention, while at the high end it represents saturation of the lock.

We have run our experiments with a number of widely differing values for the scheduling quantum Q. Quantitatively, the results vary, sometimes significantly, with the value of Q. As might be expected, larger values of Q result in greater amounts of spinning. (Note that since we do not charge for context switches in our model, there is no performance penalty for smaller values of Q.) This is illustrated in Figure 1, which shows the mean number of processors spinning as a function of Q and the number of threads for a 5 processor system, in the lock contention situation. (The mean number of processors spinning has the advantage of being easily computed from our models, as well as being directly indicative of the mean spin time, and thus is shown in many of our graphs.) The increase in spinning with Q can be attributed to an increase in the variance of the spin time, which results from more occasional but longer lasting situations in which the thread being waited for (either the one holding the lock in the lock contention case illustrated in Figure 1, or the last one to reach the barrier in the case of barrier synchronization) is unscheduled. Note that this performance benefit for smaller scheduling quanta is quite distinct from the benefit in a sequential system of allowing the rapid completion of short jobs. While quantitatively our results depend on the specific value of Q chosen, the qualitative behavior is similar in all cases. We have chosen $Q = 1$ for all results presented here. This choice was in part motivated by the fact that the rate of increase in spin time with increasing values of Q (as illustrated in Figure 1 for the lock contention case) drops dramatically as Q increases beyond 1.

Turning our attention to the parameters involving size, we chose P and J by running a set of test experiments for the lock contention situation to determine how the behavior of the system depends on its size. In these experiments J was set equal to P and T was varied (rather than being fixed at 1 as it is elsewhere) so that the lock throughput (and thus the lock utilization) is nearly constant across all system sizes. (See Appendix B for details.) This allows us to isolate the changes in mean spin time caused by system size from those that would occur naturally because of increased lock contention in the larger systems if T were held invariant.

Figure 2a shows the mean number of processors spinning as a function of system size and mean lock holding time. Figure 2b is the same data normalized by the number of spinning processors in the 5

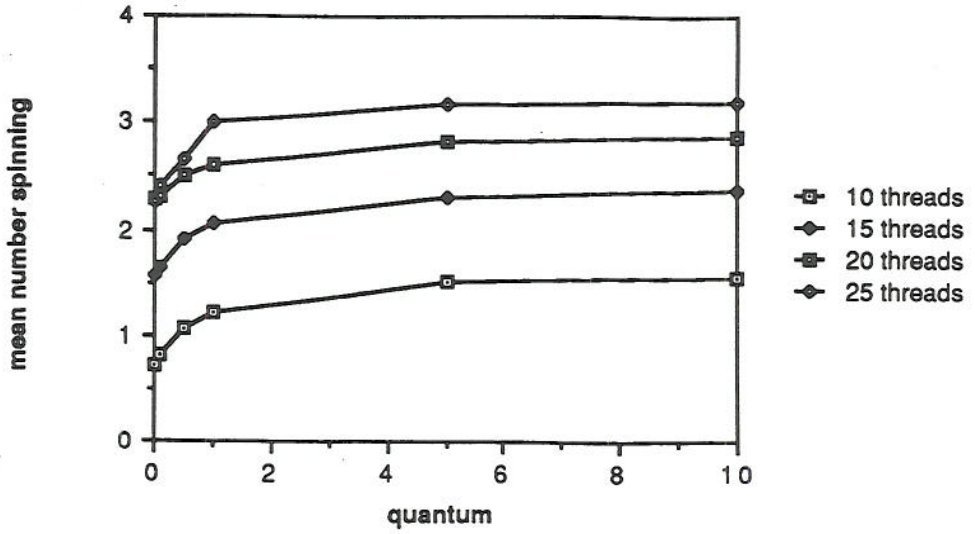

Fig. 1: Effect of Quantum Size; 5 Processor System

processor system. For short duration locks the behavior of the systems is nearly independent of system size (both in terms of the absolute difference in the spinning times, as shown in Figure 2a, and, somewhat surprisingly, in terms of the relative difference, as shown in Figure 2b). For long duration locks, on the other hand, the number spinning is nearly linearly proportional to system size. Based on this observation, we have chosen to present results for experiments with the two smallest system sizes, 5 and 10 processors, since these minimize the still considerable processing time required to run the experiments but still represent a factor of two difference in system size. Based on the above observations, results for larger systems for the lock contention situation can be safely extrapolated from those presented for the smaller systems.

Parameters P and J for the barrier synchronization case were chosen to be consistent with the lock contention results, that is, we again restricted P to 5 and 10 processors. In each model J is kept constant at the number of processors, since this lets us easily compare the baseline and multiprogramming cases. Finally, the number of other threads in the multiprogramming environment, K, was varied across an extensive range; we present selected results.

3. Results for Lock Contention

3.1. The Baseline Case

As noted previously, the baseline case represents the situation where the programmer has the greatest information available at implementation time about expected spin times. We assume that the parallel machine is dedicated to a single application during its execution and that the application has been partitioned to have precisely the same number of threads as there are processors. Thus, we model P processors and $J = P$ threads.

Performance measures for this baseline environment are used only for comparative purposes. The mean number spinning for this case is contained in the data given in Figure 3, in the context of a comparison with the multiprogramming case, which we discuss next.

3.2. Multiprogramming

We examine the effects of multiprogramming by comparing the mean number of spinning processors under multiprogramming (i.e., when the number of threads exceeds the number of processors) to that in the baseline case. We have run our experiments with J varying from P to $5P$. The performance results obtained are qualitatively similar, so we have extracted the results for $J = 2P$ for presentation here.

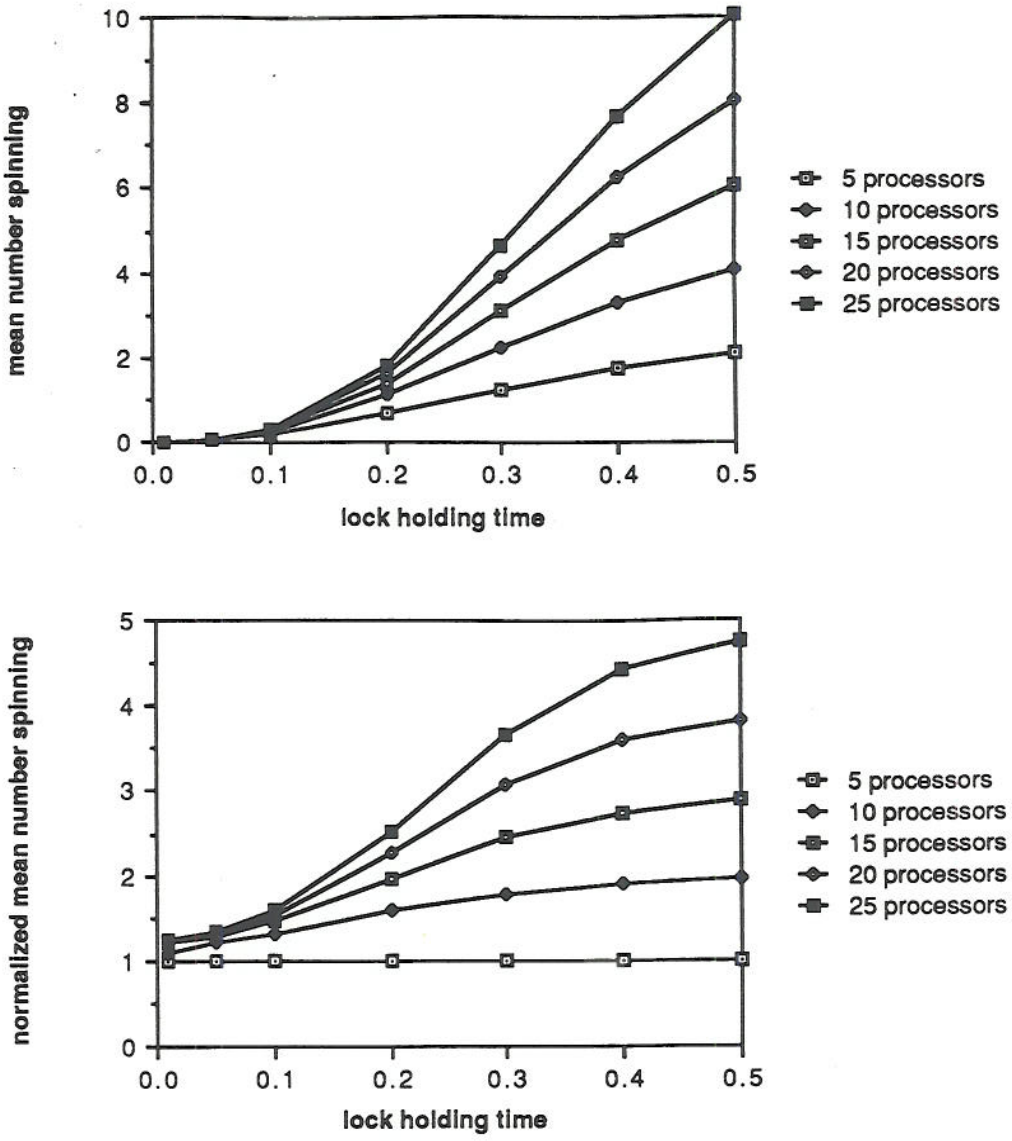

Fig. 2a/2b: Effect of System Size

It might appear at first that doubling the number of threads would result in increased lock contention, and that any increase in the number of spinning processors would be a combination of this effect and the effect of multiprogramming. As noted previously, though, lock contention is inherently dependent on the number of processors, not the number of threads. Because the number of processors is kept constant, the total instruction delivery rate, and so the total rate of lock requests and the resulting lock contention, also are kept constant, with the exception of changes due solely to differing patterns of thread executions. Thus, any change in system performance can be attributed solely to the introduction of multiprogramming.

Figure 3 presents a comparison of the baseline and multiprogramming environments for 5 and 10 processor systems. When lock contention is low (represented here by short lock holding times) system performance is not significantly affected by multiprogramming. Thus, in these environments the author of parallel software can choose between spinning and blocking as though the application were to be run standalone. However, at modest to high lock contention, multiprogramming causes a significant degradation in performance. This effect is the result of the fact that the thread holding the lock is occasionally unscheduled. In these instances other threads will spin for a scheduling quantum, a considerable period of time.

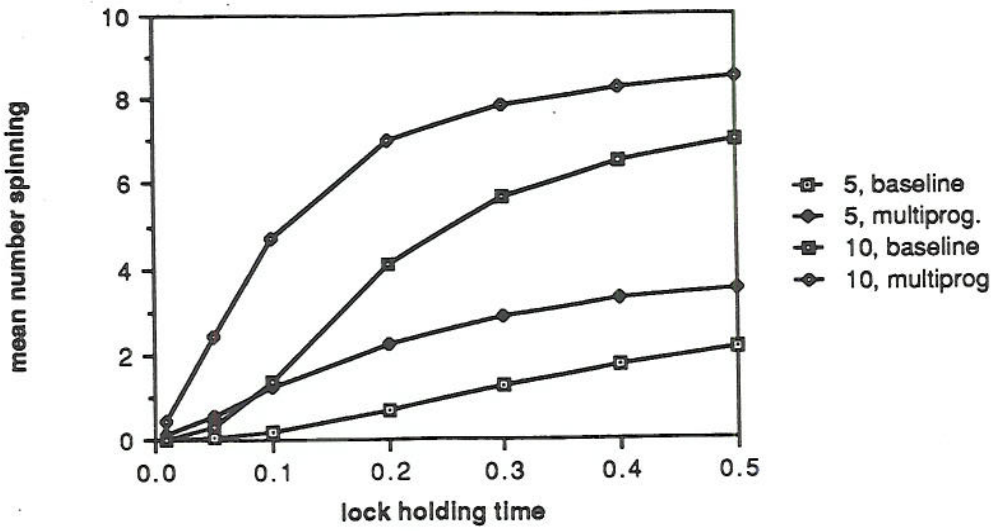

Fig. 3: Baseline Case and Multiprogramming; 5 and 10 Processor Systems

The above analysis has assumed that the scheduling discipline is oblivious to the internal behavior of the application software, that is, to the states of the threads involved in the scheduling decision. It might be possible to obtain better system performance if the scheduler had access to this information, since, for example, the scheduler could then avoid unscheduling the thread that holds the lock. To accomplish this, the scheduler must rely on the application software (perhaps through the code implementing the language primitives supporting locking) to set a flag when it acquires a lock, because (for efficiency reasons) the operating system is not involved in lock requests. However, this might have one or both of the following undesirable effects. First of all, if it required that an action on the flag be performed inside the critical section, it would increase by at least one instruction the lock holding time, which could be critical to system performance [Dritz & Boyle 1987]. Second, an unscrupulous user might be able to modify his code so as to set this flag for all his threads in an attempt to obtain better service [Coffman & Kleinrock 1968].

We have investigated the potential performance benefits that could be obtained if the scheduler had knowledge of the state of the threads, i.e., whether they were computing, spinning, or holding the lock. We have investigated three policies that use this information. In Discipline A the scheduler never unschedules a thread holding the lock. This eliminates the situation where threads are spinning uselessly waiting for the lock to be released by an unscheduled thread. Discipline B allows the thread holding the lock to be unscheduled, but will not schedule a currently unscheduled spinning thread unless the lock is free. This discipline has the same goal as the first, to reduce useless spinning, but it reduces the motivation of a user to lie about the state of his threads. The final policy, Discipline C, combines both of the previous modifications.

Figure 4 presents a summary of the effects of these improved scheduling policies on system performance. As is readily seen, all three policies result in significant improvements in system performance, and nearly eradicate the performance penalty imposed by multiprogramming (cf. Figure 3). Discipline A is preferable to Discipline B, and yields performance almost as good as when both modifications are combined.

Perhaps one of the less intuitive characteristics of Figure 4 is the shape of the curve for Discipline B. For low lock holding times (less than about 0.1 in Figure 4b, for example), Discipline B yields significantly worse performance than that of Disciplines A and C. The curve then exhibits a distinct change in shape (at around a lock holding time of 0.1 in Figure 4b), and for larger lock holding times quickly converges to the curves for Disciplines A and C. This shape can be (at least partially) explained as follows. For low lock holding times, there are usually no or very few threads (either with or without processors) that are in the spinning state. Thus, Discipline B yields little improvement in performance in this case. As the lock holding time increases, the average number of threads in the spinning state

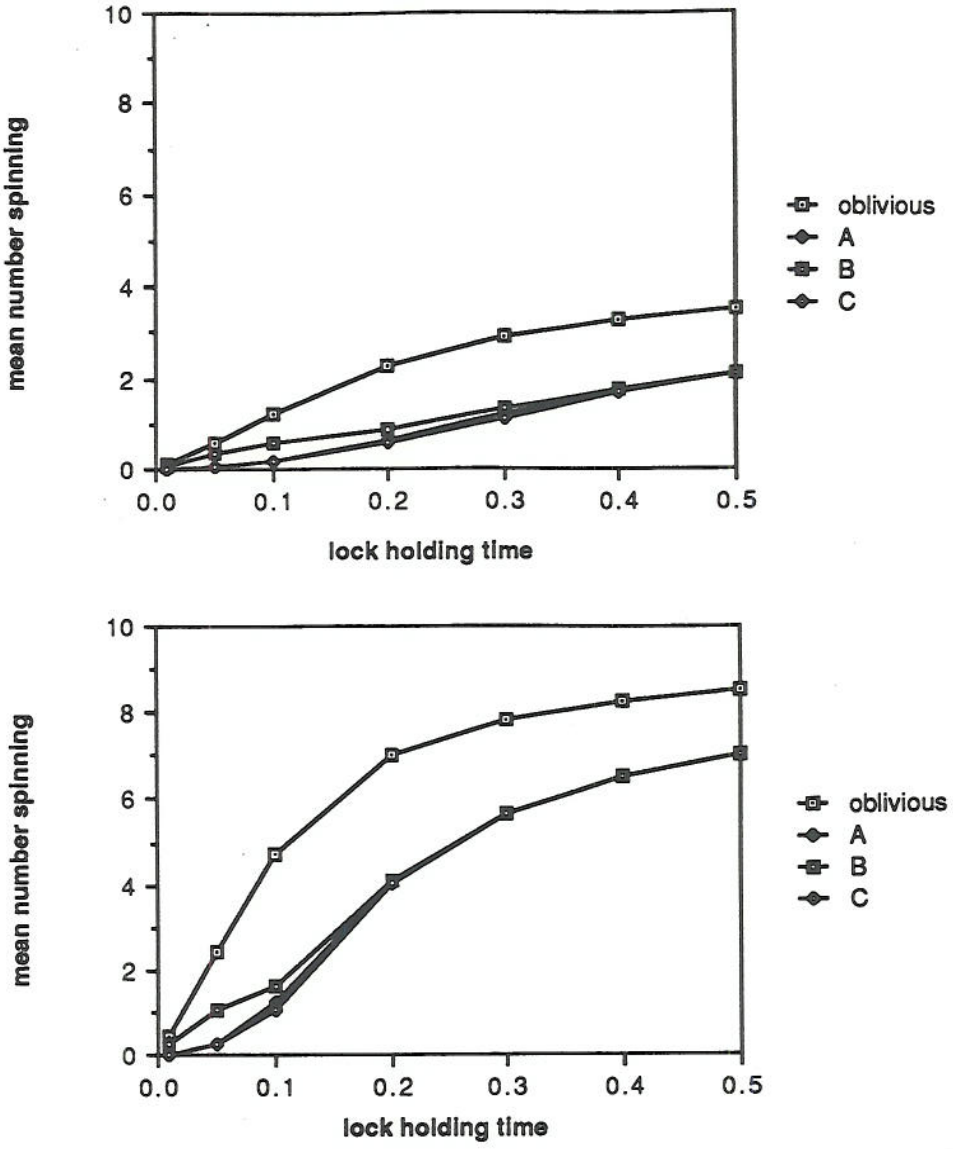

Fig. 4a/4b: Effect of Scheduling Policy; 5 and 10 Processors (cf. Fig. 3)

increases. With Discipline B, such threads tend to be kept unscheduled, diminishing the contention for processors among the remaining threads. Therefore, although the lock holder fImay be context-switched in discipline B, this becomes increasingly less likely (because of the absence of a suitable thread to switch it with), and in any case the average time until that lock holder is rescheduled becomes very small, as the lock holding time is increased. Thus, for large lock holding times, Discipline B closely corresponds to Disciplines A and C.

All three improved policies also render the system nearly insensitive to the total number of threads in terms of the mean number of processors spinning. Figure 5 illustrates this effect using Discipline A as an example.

It is natural to ask in the multiprogramming context which of spinning and blocking is the preferable waiting mechanism. To at least partially address this question, we have chosen to give an informal threshold for context switch times as a function of the parameters of our model. This threshold is such that context switch times less than the specified value should result in blocking being preferable to spinning (within the assumptions of the model). Context switch times greater than the threshold value *may* still result in blocking being preferable, although for context switch times much larger than the threshold this is unlikely. Note that, in practice, context switch times depend heavily on the specific

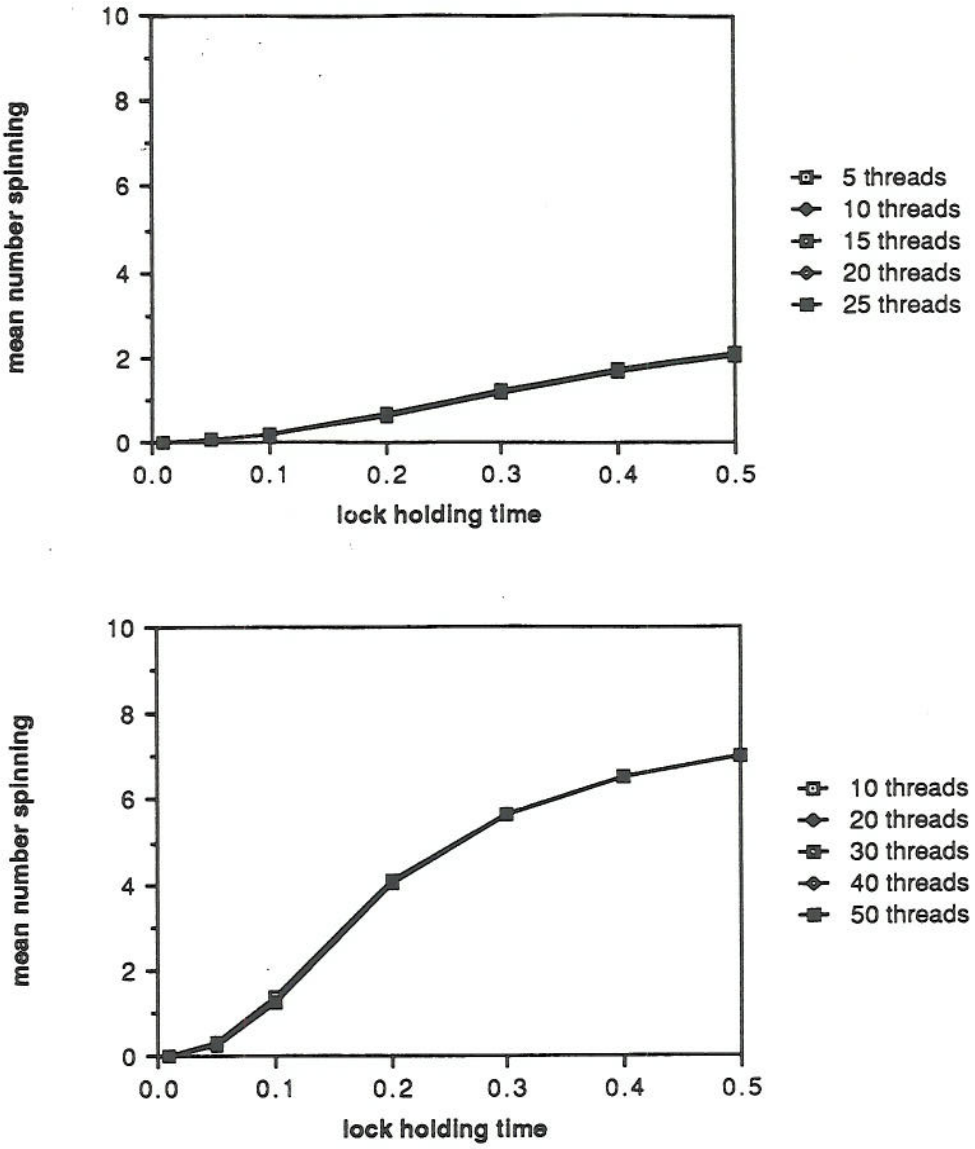

Fig. 5a/5b: Discipline A Performance vs. Number of Threads; 5 and 10 Processors

architecture and level of granularity of parallelism [Polychronopoulos & Kuck 1987]. For this reason, and due to the abstract nature of our model, the threshold values we obtain are more useful as relative values (when comparing the various scheduling disciplines) than as absolute values.

The threshold is computed using an approximation to the average spin time per initially unsuccessful lock request, considering only those time intervals during which at least one unscheduled thread is in the compute or critical section states.[2] We ignore spin time when there are no unscheduled threads in these states because blocking is not useful in that case. Blocking is likely to be advantageous if the context switch time is smaller than the threshold value, since blocking should result in a smaller amount of wasted processor time in this case.

[2] The approximation assumes that the rate of initially unsuccessful lock requests is independent of the presence or absence of unscheduled non-spinning threads. Perhaps surprisingly, we found that the nature of the results was insensitive not only to the use of this approximation (rather than an exact analysis), but also to the precise way in which the threshold was defined (alternate definitions gave equivalent results).

Figures 6a and 6b present the threshold values for the four multiprogramming scheduling disciplines in systems with 5 and 10 processors and twice the number of threads as processors. (The thresholds for larger numbers of threads mimic the shape of the values given, although of course are larger in value. The thresholds for the "oblivious" scheduling discipline and Discipline B are relatively sensitive to the number of threads in the system, while those for Disciplines A and C are largely unaffected by that parameter.) The threshold values decline for large lock holding times for some of the disciplines because of the decreasing probability that any of the unscheduled threads have useful work to do.

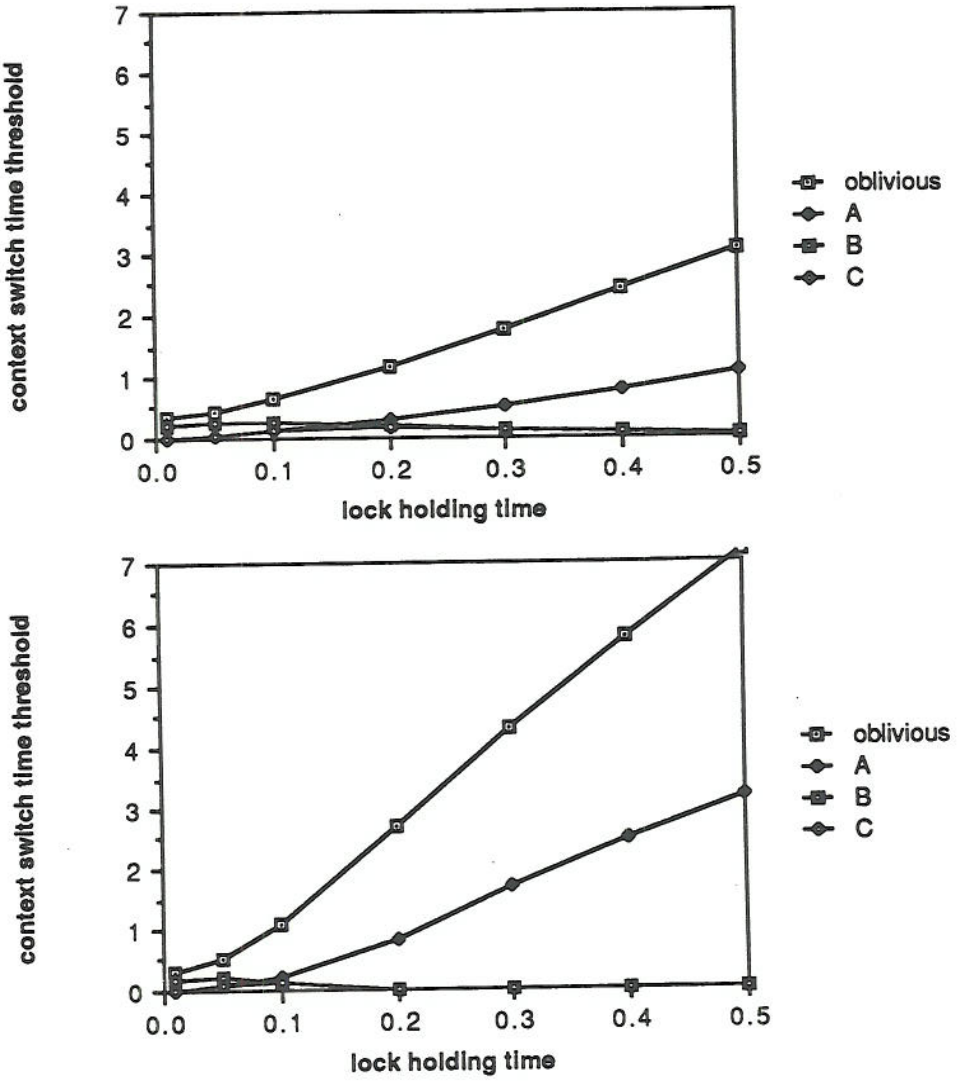

Fig. 6a/6b: Context Switch Time Thresholds; 5 and 10 Processors

It is clear from Figure 6 that there is a significant qualitative difference in the behavior of the various multiprogramming scheduling disciplines. In particular, the disciplines that refuse to schedule a spinning thread unless the lock is free (Disciplines B and C) greatly reduce the range of lock utilizations over which blocking may be an appropriate waiting mechanism. This is a characteristic that argues in favor of scheduling disciplines that embody this feature.

3.3. Data-Dependent Behavior

The final environment considered is that of data-dependent behavior. Recall that this is reflected in our model by a parameter p. A thread acquiring the lock releases it in zero time with probability $1-p$, and with probability p holds it for mean time $\frac{L}{p}$. The case $p = 1.0$ corresponds exactly to the baseline case. Figure 7 gives the ratio of the mean number of spinning processors for various values of p to the mean number spinning when $p = 1.0$. We make three observations based on this data. First, variability has the greatest effect when lock contention is low. In these cases the total amount of spinning is small in any case, so despite the potentially large percentage increase the difference in spinning is small in absolute terms. (The absolute amount of spinning occurring for the baseline case of p=1.0 can be found in Figure 2 for the 5 and 10 processor systems.) Second, quite high variability is required before any significant effect is observed. In our data a p of 0.5 is required before even a factor of two difference occurs. Finally, the behavior of the system as a function of data-dependence is nearly identical in the 5 and 10 processor systems. Thus, we conclude that the effect of data-dependent behavior is roughly independent of system size.

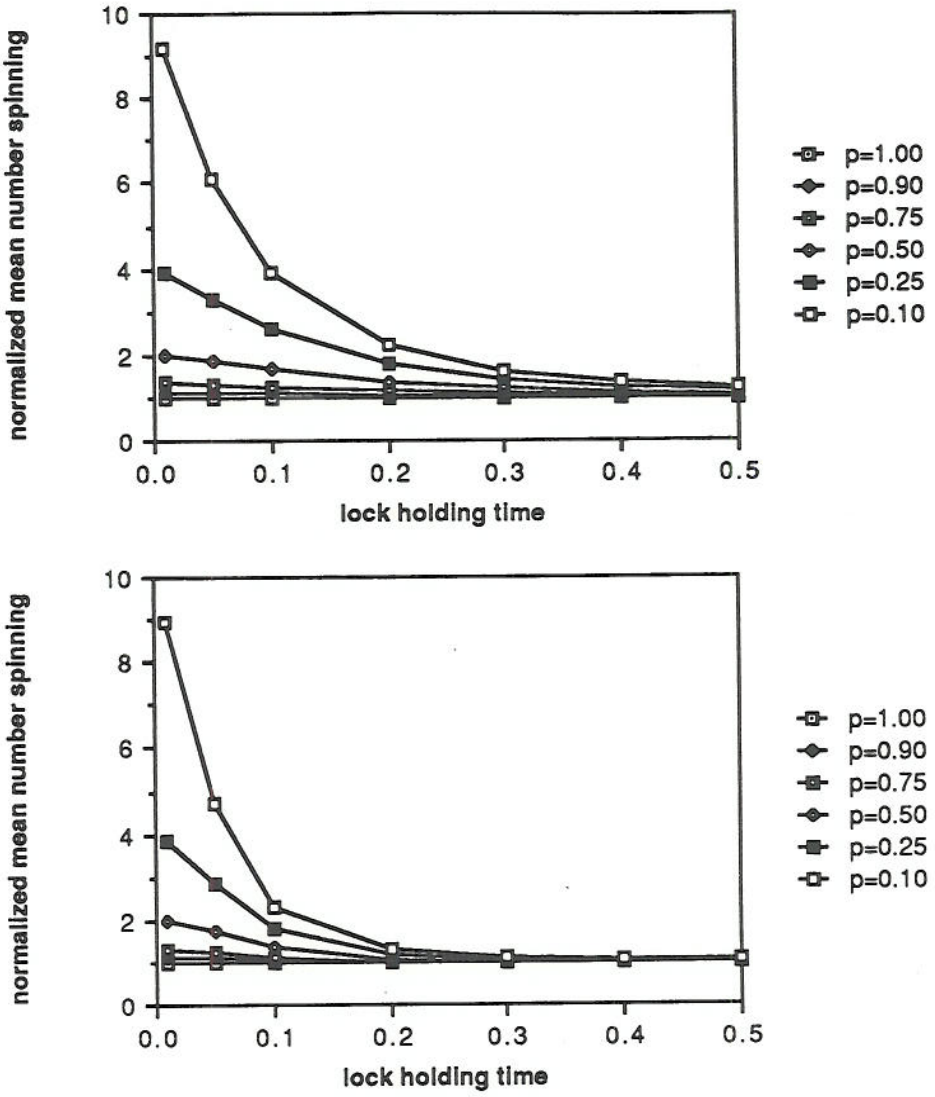

Fig. 7a/7b: Data-Dependent Behavior; 5 and 10 Processors

4. Barrier Synchronization

4.1. The Baseline Case

We now turn our attention from lock contention to barrier synchronization. Figure 8 shows how the amount of spinning is affected by the number of threads involved in the barrier synchronization. Here we have assumed that $J = P$, that is, that all threads have a processor dedicated to them. Under the assumptions of our model it is easily shown that the mean time to complete the barrier is $\sum_{i=1}^{J} \frac{1}{i}$.

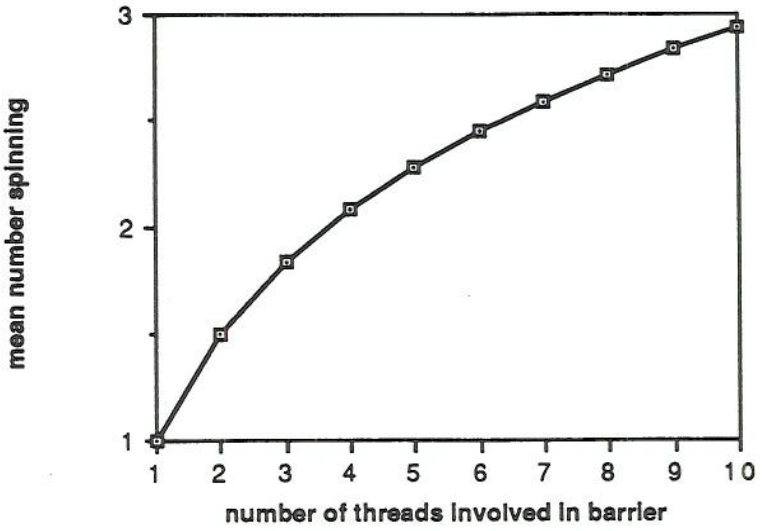

Fig. 8: Barrier Synchronization Baseline Case

4.2. Multiprogramming

To reflect multiprogramming, the number of threads involved in the barrier synchronization is kept constant at P, but K other threads are introduced that compete for processors. Figure 9a illustrates how the mean number of spinning processors is affected by the number of these "other threads" as a function of system size. Clearly, the amount of spinning per processor per time unit decreases with additional other threads because those threads never spin. Figure 9b shows how the amount of spinning per barrier thread per barrier synchronization increases with the amount of competition for processors. It gives the mean time required to achieve the barrier synchronization for various system sizes. There is a nearly linear relationship between the number of other threads and the mean time to achieve the barrier.

Just as in the lock contention situation, it is natural to ask if system performance can be improved by giving the scheduler some information about the internal state of the threads. For barrier contention, the only information that seems useful is which threads are spinning. Then if a spinning thread happens to be descheduled because its quantum has expired, it would seem to be beneficial not to reconsider scheduling it again until the barrier had been reached by all other processors.

Figure 10 shows the ratio of the performance under this modified discipline to the performance achieved under the "oblivious" discipline. It is not terribly surprising that the mean number of spinning processors is reduced by this modification. What is surprising is that this gain in overall system performance does not penalize the threads involved in the barrier. The explanation for this is that spinning threads compete with those threads still working toward the barrier. Thus, a mechanism that tends to eliminate the spinning threads helps the other threads achieve the barrier. This effect evidently outweighs the disadvantage that under the modified discipline many more of the barrier threads are unscheduled at the time the last thread reaches the barrier, and so at the time the threads begin working toward the next barrier, than under the "oblivious" discipline. Note though, that under the modified discipline, as with the oblivious discipline, the performance of the barrier threads still degrades considerably with increasing numbers of other threads.

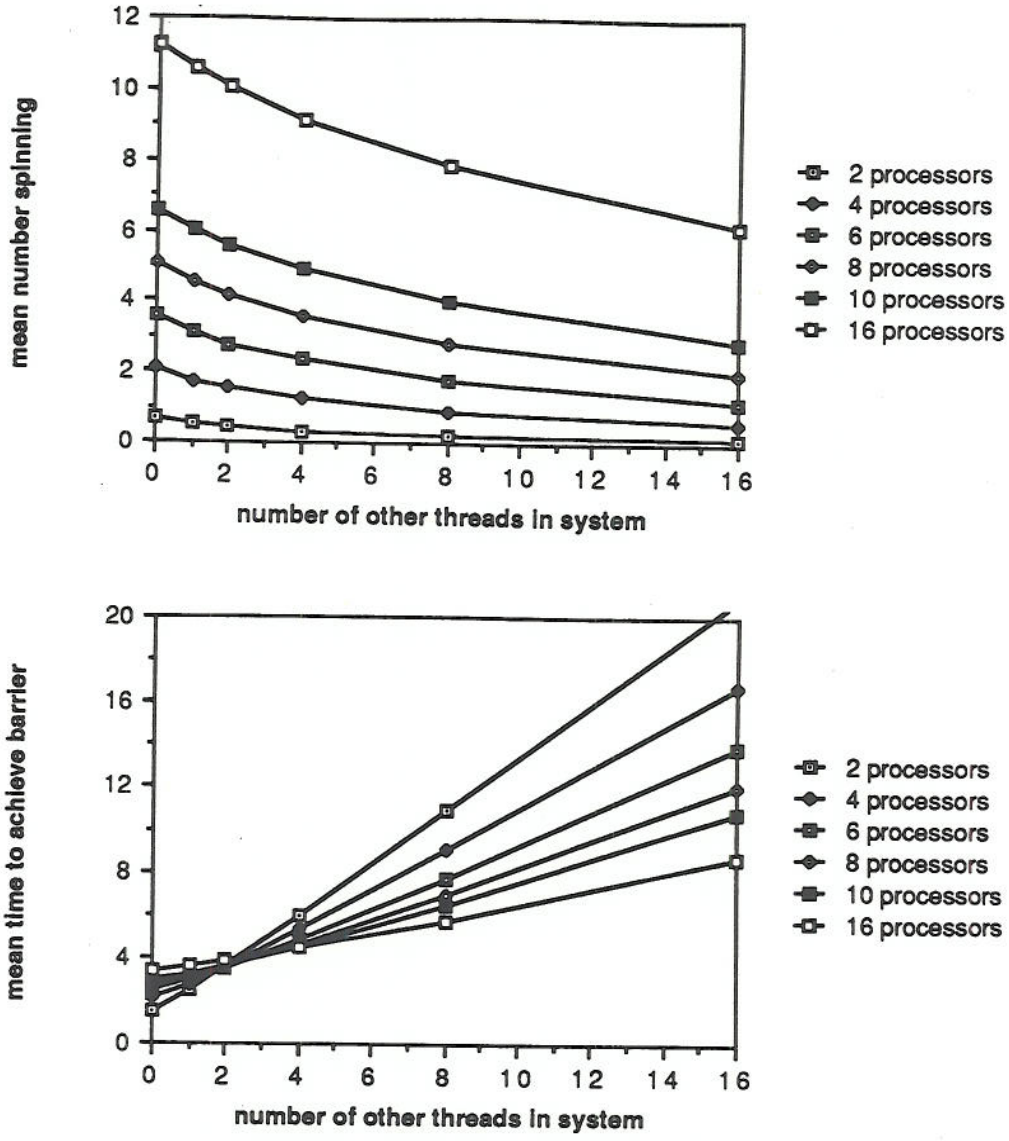

Fig. 9a/9b: Barrier Synchronization under Multiprogramming

4.3. Data-Dependent Behavior

The threads involved in the barrier synchronization in the baseline case were "balanced", in the sense that the amount of service each required before reaching the barrier was chosen from a single distribution. To model data-dependent behavior we examine the effect of introducing imbalance. We do this by letting some threads reach the barrier in zero time, while other threads take longer than the overall mean time. Recall that parameter p is the probability that a thread requires a non-zero service time.

Figure 11 shows how the amount of spinning is affected by the uncertainty in the thread execution times. For each value of p, we have graphed the ratio of the fraction of time each processor spends doing useful work against that value when $p = 1.0$. It is clear that variance can have a substantial effect on the expected spin times of threads using barrier synchronization. Further, the magnitude of this effect increases with the size of the parallel machine. Since we have experimented with quite modest system sizes, one would expect that in real systems data-dependent behavior could be quite significant to the performance of applications using barrier synchronization.

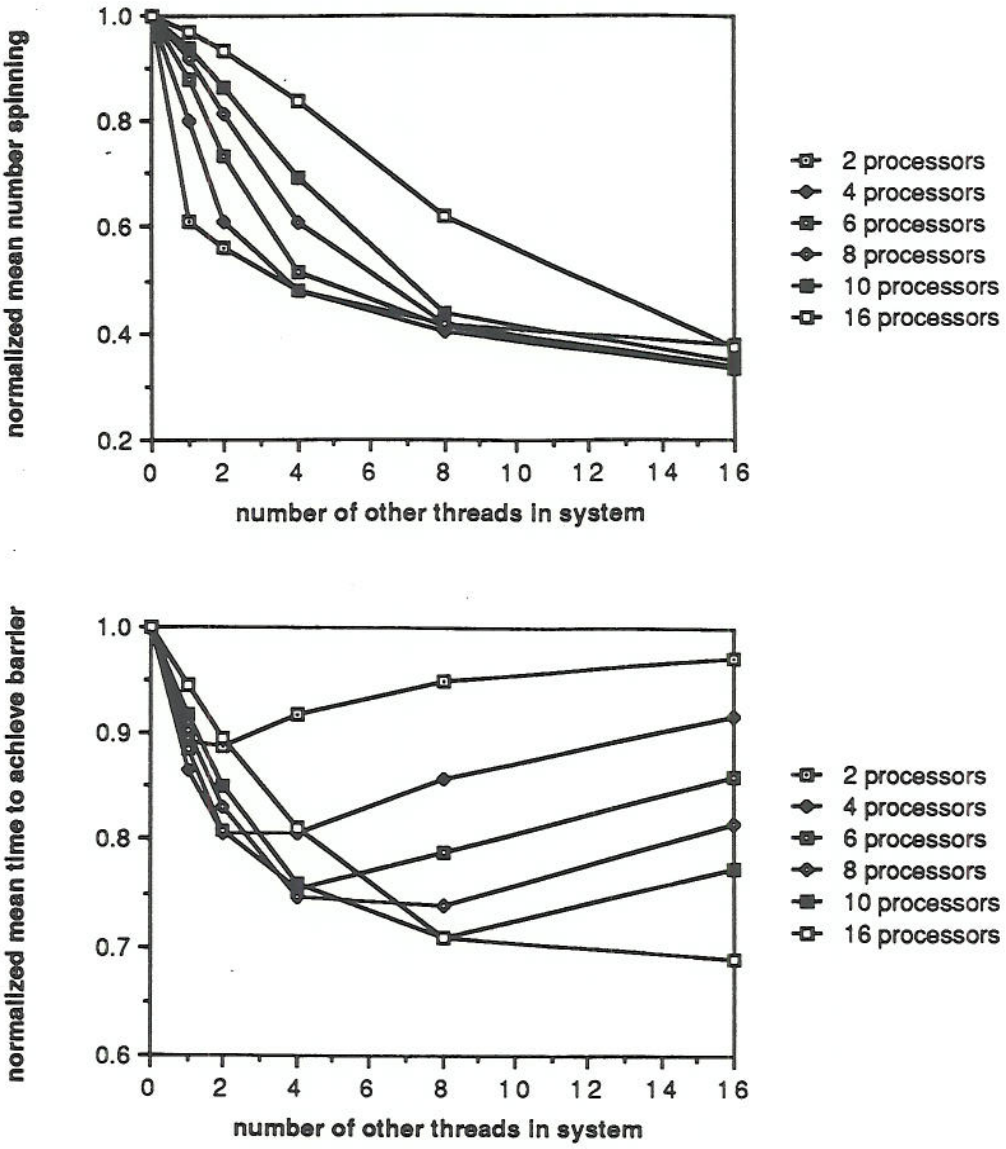

Fig. 10a/10b: Normalized Performance of Modified Discipline (cf. Fig. 9)

5. Conclusions

We have used two analytic models to compare the amount of spinning that occurs in various environments when threads either compete to obtain a lock or synchronize at a barrier. The purpose of our comparison is to determine if the uncertainty in performance caused by multiprogramming or by data-dependent behavior significantly increases the amount of spin time that occurs, and so complicates the task of choosing an appropriate waiting mechanism.

We have found that for lock contention, neither source of uncertainty poses much danger, assuming that the system scheduler has access to information concerning who holds the lock or who is spinning. However, for barrier synchronization, the amount of spinning is quite sensitive to these forms of uncertainty. Thus, to correctly choose a waiting mechanism the programmer requires fairly precise information about not only the behavior of his program but also about the load that will be placed on the machine when his application is run. For this case, then, the programmer's task is considerably more complicated in multiprogramming and/or data-dependent environments than in the case of the more controlled environment of a dedicated machine and predictable running times.

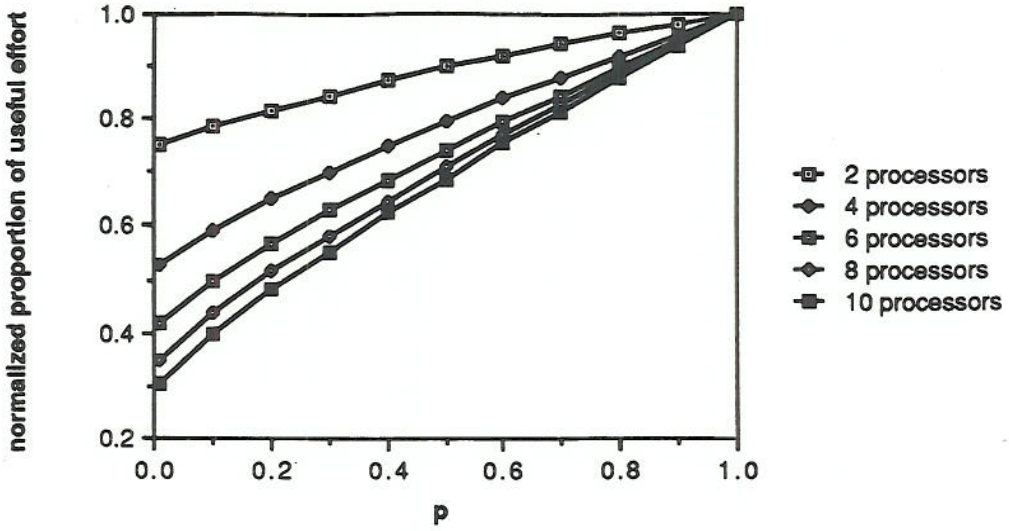

Fig. 11: Normalized Data-Dependent Behavior

Acknowledgements

Partial support for this work was generously provided by Bell Communications Research, Boeing Computer Services, Digital Equipment Corporation, Tektronix, Inc., the Xerox Corporation, and the Weyerhauser Company. The Centre National de la Recherche Scientifique, France, and Laboratoire MASI, University of Paris 6, provided generous support and resources for Zahorjan for the year sabbatical leave during which this work was performed.

References

[Beck et al. 1987]
B. Beck, B. Kasten, and S. Thakkar. VLSI Assist for a Multiprocessor. *Proc. 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1987), pp. 10-20.

[Coffman & Kleinrock 1968]
Edward G. Coffman, Jr., and Leonard Kleinrock. Computer Scheduling Methods and their Countermeasures. *Proc. 1968 Spring Joint Computer Conference*, pp. 11-21.

[Dritz & Boyle 1987]
Kenneth W. Dritz and James M. Boyle. Beyond "Speedup": Performance Analysis of Parallel Programs. Technical Report ANL-87-7, Mathematics and Computer Science Division, Argonne National Laboratory, February 1987.

[Dubois & Briggs 1982]
M. Dubois and F.A. Briggs. An Approximate Analytical Model for Asynchronous Processes in Multiprocessors. *Proc. 1982 International Conference on Parallel Processing*, pp. 290-297.

[Eager et al. 1988]
D.L. Eager, E.D. Lazowska, and J. Zahorjan. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. *Proc. 1988 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, May 1988.

[Kleinrock 1975]
L. Kleinrock. *Queueing Systems: Volume I: Theory*. John Wiley and Sons, 1975.

[Lazowska et al. 1984]
E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[Lipsky & Church 1977]
L. Lipsky and J.D. Church. Applications of a Queueing Network Model for a Computer System. *Computing Surveys 9*,3 (September 1977), pp. 205-222.

[Lo & Gligor 1987]
S.-P. Lo and V.D. Gligor. A Comparative Analysis of Multiprocessor Scheduling Algorithms. *Proc. 7th International Conference on Distributed Computing Systems* (September 1987), pp. 356-363.

[Ousterhout 1982]
John K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proc. 3rd International Conference on Distributed Computing Systems* (October 1982), pp. 22-30.

[Polychronopoulos & Kuck 1987]
C.D. Polychronopoulos and D.J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers C-36*,12 (December 1987), pp. 1425-1439.

[Rodgers 1986]
David P. Rodgers. Personal Communication. October 1986.

[Stewart 1978]
W.J. Stewart. A Comparison of Numerical Techniques in Markov Modelling. *CACM 21*,2 (February 1978), pp. 144-152.

Appendix A: Details of the Analysis of the Models

In this appendix we specify more precisely the models we have used and the analysis by which we have obtained the results presented in the body of this paper. We describe only the model for the lock acquisition situation. The model for barrier synchronization is similar.

We use a Markovian model to represent a system with P processors and $J \geq P$ threads. A state of the system is given by a six-tuple $(n_1,n_2,n_3 | n_4,n_5,n_6)$. Here n_1, n_2, and n_3 are the number of threads currently scheduled (that is, allocated processors) that hold the lock, are spinning, and are computing respectively, and n_4, n_5 and n_6 are the corresponding counts of unscheduled threads in those three states. Thus, there are a total of $2P(J-P)+2J-P+1$ states, of which $J-P+1$ are of the form $(0,0,P | 0,n_5,J-P-n_5)$, $P(J-P+1)$ are of the form $(1,n_2,P-n_2-1 | 0,n_5,J-P-n_5)$, and $(P+1)(J-P)$ are of the form $(0,n_2,P-n_2 | 1,n_5,J-P-n_5-1)$.

A computing thread makes a lock acquisition attempt after an amount of service (i.e., time on a processor) exponentially distributed with mean T. With probability p, a thread acquiring the lock releases it after an exponential amount of service exponentially distributed with mean $\frac{L}{p}$. With probability $1-p$ the lock is released in zero time. This is actually modelled by having "multi-step" transitions in the Markov model, that is, transitions between states that imply the movement of more than a single customer. Details on this follow when the state transition rates are defined.

Here only the "oblivious" multiprogramming scheduling discipline is considered. (The modifications required for the other scheduling disciplines are straightforward.) Each thread allocated a processor is descheduled after an amount of time exponentially distributed with mean Q. A currently descheduled thread is chosen at random as a replacement.

The steady state solution of this model is obtained by solving the state flow balance equations [Kleinrock 1975]. It is difficult to give the flow balance equations in a compact form. We therefore write down the same information in a different form, giving simply the rate of flow out of each state and the state to which that flow enters.

Let $S \equiv (n_1,n_2,n_3 \mid n_4,n_5,n_6)$ be a state of the system. Let S_i be obtained from S by subtacting 1 from n_i, S^i be obtained by adding 1 to n_i, and allow these operations to be applied repeatedly. For example, $S_{3,5}^{2,6} = (n_1,n_2+1,n_3-1 \mid n_4,n_5-1,n_6+1)$. Then the flow out of state S is given by:

$$S \Longrightarrow S_3^2 \text{ with rate } (n_1+n_4)\frac{n_3}{T}$$

$$S \Longrightarrow S_3^1 \text{ with rate } (1-n_1-n_4)\frac{n_3}{T}p$$

$$S \Longrightarrow (1,n_2-i,n_3+i \mid n_4,n_5,n_6) \text{ with rate } \frac{n_1 p}{L}(1-p)^{i-1}p \quad , \quad 1 \le i < n_2$$

$$S \Longrightarrow (0,0,P \mid n_4,n_5,n_6) \text{ with rate } \frac{n_1 p}{L}(1-p)^{n_2}$$

and, for $J > P$,

$$S \Longrightarrow S^{i,3+j,3+i} \text{ with rate } \frac{n_j}{Q}\frac{n_{3+i}}{J-P}, \qquad 1 \le i \ne j \le 3$$

We obtained the solution of the flow balance equations by the power method [Stewart 1978]. This is an asymptotically exact iterative technique involving repeated multiplication of the current estimate of the steady state probability vector with the transition matrix. We initialized the probability vector so that all states had equal probabilities. We stopped the iteration when the sum of the absolute values of the changes in the state probabilities in successive iterations was below a threshold of 0.00005. We use the sum of the changes in all states rather than the maximum change in any one state because we found that this latter measure lead to unreliable results. Our threshold value was determined to be adequate by solving a number of test cases starting with relatively large thresholds and then repeatedly halving the threshold and resolving. Comparing the results obtained each time the threshold was halved, we informally concluded that the results were reliable when halving the threshold did not produce an appreciable change. We know of no problems with the accuracy of the solutions we have obtained using this threshold, but we did observe that some models required a very large number of iterations to reach convergence. This was typically the case when there were large differences in time scales in the model. For instance, we ran some cases with T=1, Q=1 and L=0.001. These models often resulted in long execution times.

Given the steady state probability vector provided by the power method, computing performance measures is straightforward. For instance, denoting the steady state probability of state S by $P(S)$, the mean number of spinning processors is given by

$$\sum_{\text{all states } S} n_2 P(S)$$

and the lock throughput rate is given by

$$\sum_{\text{all states } S} \frac{n_1}{L} P(S)$$

One reservation that might be raised about our model is that all service time distributions are exponential. Indeed, the work by Dubois and Briggs [1982] presents a more complicated (and more restrictive) model requiring a heuristic analysis with the sole purpose that lock holding times can be made less variable than the exponential. We believe that exponentials are acceptable in our models for two reasons. First, as explained in the introduction, our results depend on comparing models all of which use exponentials. Experience shows that in general these sorts of comparisons are highly robust to inaccuracies such as the choice of service time distribution [Lipsky & Church 1977, Lazowska et al. 1984]. Second, in a similar model in the domain of load sharing [Eager et al. 1988] the specific effect of the exponential distribution was explored and compared against results obtained from a deterministic distribution. In that work, which was also based on the comparison of models, very little difference was observed between the exponential and the deterministic set of models.

Appendix B: Estimating Compute Times

In the set of test experiments that were ran to determine the effect of system size on its behavior (described in Section 2.3), T was varied so that the lock throughput was nearly constant across all system sizes. In this appendix, we briefly describe the manner in which this was done.

Because lock throughput depends on T, choosing the T required to exactly equalize lock throughputs across system sizes requires iteration and its consequent high cost. We therefore chose to use a simpler but effective approximate technique that does not require iteration. This approximation is obtained by assuming that no lock contention will take place. Under this assumption lock throughput is given by $\frac{P}{T+L}$, and so lock utilization is $\frac{PL}{T+L}$.

The procedure we followed was to solve the model with the chosen values of L and T for the 5 processor system and to note the resulting lock utilization U. For a P processor system we then solved for the appropriate compute time T_P as

$$U = \frac{PL}{T_P + L}$$

Our assumption of no contention is clearly valid for low values of lock holding time, but it is not clear how well it performs for larger values. We therefore monitored the lock throughput actually resulting from our choices for the T_P. In no case did the lock throughput rate differ from that of the 5 processor system by more than 10%.