Using Scheduler Information to Achieve Optimal Barrier Synchronization Performance

Leonidas Kontothanassis and Robert W. Wisniewski

Abstract

Parallel programs frequently use barriers to synchronize successive steps in an algorithm. In the presence of multiprogramming the choice of spinning versus blocking barriers can have a significant impact on performance. We demonstrate how competitive spinning techniques previously designed for locks can be extended to barriers, and we evaluate their performance. We design an additional competitive spinning technique that adapts more quickly in a dynamic environment. We then propose and evaluate a new method that obtains better performance than previous techniques by using scheduler information to decide between spinning and blocking. The **scheduler information** technique makes optimal choices incurring little overhead.

1 Introduction

Although the decision between spinning and blocking at synchronization events is straightforward on a dedicated multiprocessor, multiprogramming introduces significant difficulties in many cases. Past work [3, 11, 14] has shown that on a multiprogrammed multiprocessor the best scheduling method is dynamic hardware partitioning. In such an environment it is not possible to determine *a priori* the number of processors on which an application will run. In fact, this number may vary over the execution of the program.

There are several techniques for barrier synchronization ranging from always spin to always block. In a dynamic environment the best techniques lie somewhere between the extremes. A competitive algorithm is an online algorithm whose worst-case performance is within a constant factor of the best offline algorithm. Competitive spinning for barriers tries to approximate the best offline algorithm by heuristically determining the overall distribution of spinning times. Competitive spinning uses this information to choose the appropriate policy and to minimize the expected cost over all executions of a barrier. Keeping track of this information is costly so there are several approximation algorithms that attempt to determine this distribution based on recent executions of the barrier.

The main difficulty in chosing an appropriate strategy in a hardware partitioned environment is that the number of processors devoted to a particular application varies with time. However, if the program could be made aware of how many processors it was allotted when it reached a particular barrier, a better performing choice could be made between blocking and spinning. We propose a method whereby the scheduler provides this information to the application. When the application is attempting to cross a barrier it will know how many processors are currently in its partition. A particular thread will block when it reaches a barrier if the number of threads not yet at the barrier is greater than the number of processors in its partition, else it will spin. A technique that decides either to block all threads or to have all threads spin, even if it does so very well, will not perform as well as possible. To perform well in a hardware partitioned environment, a method must allow some processes at a barrier to spin, and others to block. The **scheduler information** technique is optimal, meaning that, in addition to allowing the maximum possible number of processes to spin, it always chooses the correct action between blocking or spinning incurring only the small overhead needed to check the number of remaining threads against the number of processors in the partition.

The rest of this paper is organized as follows. Section 2 discusses related work and distinguishes our approach from previous ones. Section 3 presents the programming model used for the experiments, describes the competitive spinning techniques, and explains how scheduler information is used to perform barrier synchronization. Section 4 presents results comparing the **scheduler information** technique to other techniques. Finally, Section 5 summarizes our work, and suggests directions for future research.

This material is based upon work supported by DARPA grants MDA972-92-J-1012 and N000014-92-J-1801 and the National Science Foundation grant CDA-8822724. The Government has certain rights in this material.

2 Related Work

Previous work on synchronization under multiprogramming has considered the performance effects of: multiprogramming on synchronization events, alternative implementations of synchronization data structures, and alternative scheduling strategies on synchronization events.

Early work by Tucker and Gupta [11] has shown that the main reason for performance degradation on a time-shared, multiprogrammed multiprocessor is the existence of more processes than physical processors. They have proposed a scheme of hardware partitioning that prevents the number of processes systemwide from exceeding the number of processors. However, this scheme requires applications to adjust the number of processes they use. For many programs it would be difficult to dynamically adjust the number of processes used, especially for the class of programs with coarse grained threads and barrier synchronization.

Using simulation, modeling, and experimentation, Crovella *et al* [3], Leutenegger and Vernon [7], and Zahorjan and McCann [14] have shown that dynamic hardware partitions are preferable to timesharing, co-scheduling, and static hardware partitions. Other work by Gupta *et al* [5] has studied the behavior of a multiprogrammed multiprocessor environment by simulating the concurrent execution of four applications. Their results indicate that hardware partitioning outperforms other policies most of the time.

Zahorjan, Lazowska, and Eager [12, 13] have determined the worst case performance degradation due to spinning in parallel systems when the number of processors available to the application varies with time. They have shown that although for locks, spinning introduces only moderate overhead, in the case of barriers spinning can degrade performance severely. They propose that processor allocation and individual thread scheduling be handled cooperatively by the system and the application.

Other work suggesting cooperation between the operating system scheduler and the runtime environment or the application includes the work of Black [2], Edler *et al* [4], Marsh *et al* [9], and Anderson *et al* [1]. Black has suggested that the application provide hints to the scheduler indicating which application thread to schedule next. Edler *et al* have suggested temporarily preventing a thread from being preempted while it is in a critical section. Marsh *et al* and Anderson *et al* have suggested that preemption information be communicated from the kernel to user space.

Markatos *et al* [8] have analyzed the suitability of alternative barrier data structures in a multiprogramming environment. They conclude that combination barriers, using blocking synchronization within a node and spinning across nodes, perform the best.

Karlin *et al* [6] have developed and tested a set of different competitive spinning techniques for a multiprocessor environment. Competitive spinning assumes that the behavior of a lock does not change rapidly with time, and that past behavior is an appropriate indicator of future behavior. While their techniques were developed for spinlocks, they can easily be extended to barriers. These techniques are most useful in a hardware partitioned environment where scheduling decisions are infrequent and many barriers are reached between scheduling decisions.

Our work extends previous work in several ways. Based on work by Karlin *et al*, we build and test several competitive spinning algorithms for barriers and conclude they have a significant performance benefit over always spinning or always blocking. Further, we propose that with only a little information (i.e. the size of the partition) from the kernel scheduler, it is possible to implement an efficient policy for barriers that performs better than all other policies. We verify this hypothesis on an Iris 4D/480 eight node multiprocessor using two real applications and a synthetic program.

3 Algorithms

As discussed in the previous sections, this work focuses on building barriers for a multiprogrammed environment. Since most researchers agree that the best way to multiprogram a multiprocessor is dynamic hardware partitions, this will be the environment assumed for the rest of the paper. It is unrealistic to assume that all programs can be written to dynamically adjust the number of processes they use to the number of processors provided. It is therefore not possible to obtain good barrier performance in these programs by always blocking or always spinning. Well performing barrier methods in a hardware partitioned environment must use a combination of spinning and blocking. The difficulty is dynamically determining the appropriate method for a particular thread in a particular instance of a barrier.

Patterned after the competitive spinning techniques used by Karlin *et al* [6] for locks, we built a set a competitive spinning techniques for barriers. These techniques decide whether to spin or block based on information from the last several barriers. The success of this technique stems from the fact that the hardware partition of an application will change considerably less frequently than the rate barriers are encountered. Pseudocode for the different competitive spinning algorithms is presented in the Appendix. These algorithms heuristically decide whether to block or spin based on their view of the hardware partition. Our final technique uses scheduler information. Instead of requiring the application to guess the partition configuration, this technique allows the size of the partition to be known, allowing the application to make a better decision than any technique limited to heuristically determining the partition. The rest of this section describes the different competitive spinning approaches and their motivation followed by the **scheduler information** technique.

All the competitive spinning techniques have a similar structure. They spin for a certain amount of time, and then block. Barriers that spin for a constant amount of time and then block are called fixed spinning barriers. Those that vary

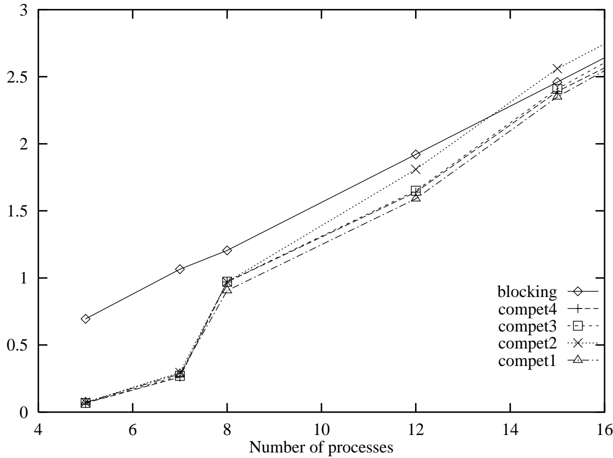


Figure 1: Competitive barriers for the synthetic program

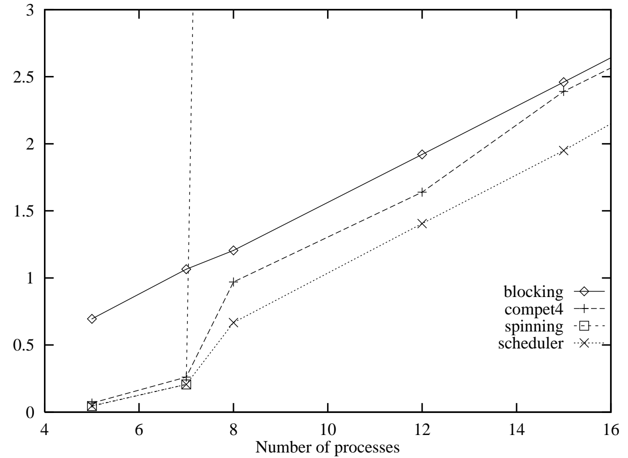


Figure 2: Scheduler information barrier for the synthetic program

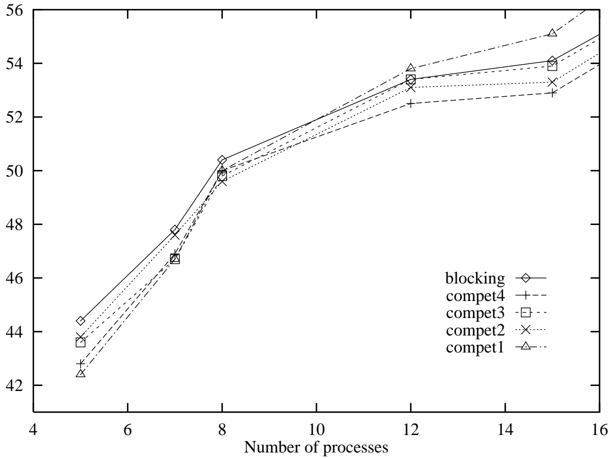
the amount of time they spin before blocking are called adaptive. Without precise information of the hardware partition, adaptive techniques heuristically determine how the hardware partition is configured. The adaptive techniques estimate whether they spent longer or shorter at the barrier than expected. Based on this, they adjust their spin time for the next instance of the barrier. In the algorithms presented in the Appendix, whenever the command "block" occurs, the process that blocks will remain blocked until the last process enters the barrier and unblocks the other processes.

A complete description and motivation behind the **fixed spinning**, **last one**, and **average three** can be found in Karlin *et al* [6]. Briefly, **fixed spinning** implies that a process spins for a constant amount of time before blocking. It requires low overhead and is easy to implement. **Average three**, which adapts the spinning period based on the last three barrier episodes, avoids the excessive cost of storing the entire distribution while still maintaining the smoothing effect. **Last one** is patterned after **average three** but has lower overhead since it retains only information from one barrier. The pseudocode for these are in figures 9 through 11 respectively (in Appendix).

The last adaptive technique, **coarse adjustment** is similar to the **last one** in that it only uses data from the previous barrier. However instead of slowly moving the spin time up or down, it adjusts it entirely at once. The reasoning behind this strategy is that partition changes are infrequent and after a partition change the desired barrier behavior is fixed. Rapidly adjusting the spin time could lead to instability and poor performance if there are very few barriers per scheduling decision. However, for normal programs this proved to be an effective method. The pseudocode for **coarse adjustment** is in figure 12 (in Appendix).

With only a little bit of information about the size of the partition, it is possible to design a barrier with much better performance than the competitive spinning techniques. The scheduler was modified to constantly provide a shared variable containing the number of processors in a partition. The barrier was designed so that when any thread reached a synchronization point it compared the number of processors in the partition to the number of remaining threads needed to pass through the barrier. If the number of remaining threads was less than the number of processors in the partition, the checking thread chose spinning, else it blocked. This approach makes optimal choices and incurs only the little overhead required to check the number of processes left against the number of processors in the partition. Pseudocode for the **Scheduler Information** barrier appears in figure 13, and code for a process of a typical application using barriers can be seen in figure 14 (in Appendix).

In the general barrier case there are N processes trying to get through the barrier using P processors where $N \geq P$. If the `work` section is roughly the same for each thread, then all the threads can be separated into groups of size P, and the threads in each group can execute simultaneously. When those P threads reach the barrier point they can either spin for the remainder of their quantum or yield and let the next P threads run. Under these assumptions the optimal policy would force the first $N - P$ threads to block so that the remaining P can run. However blocking is not necessary for the last P threads since there is no remaining thread to perform any work. Blocking in this case would only incur the extra overhead of having to switch the thread back in when the barrier is completed. The **scheduler information** technique uses this method, yielding an optimal cost of $Cost_{sched} = \lceil \frac{N-P}{P} \rceil * c + overhead$, where *overhead* is the cost of checking the number of threads against the number of available processors, c is the cost of a context switch, and *Cost* is the amount of time spent at a barrier in spinning or context switching when useful work could have been done.



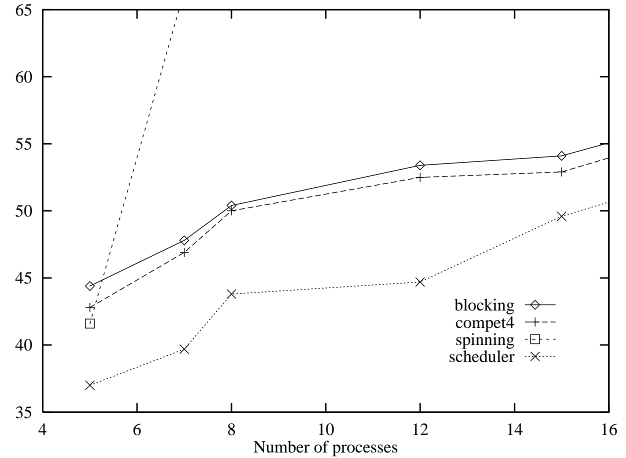


Figure 3: Competitive barriers for successive overrelaxation

Figure 4: Scheduler information barrier for successive overrelaxation

4 Experiments and Results

We tested each technique on three different programs. The first was a synthetic program allowing us to isolate barrier performance. Each process of the synthetic program executes a loop that performs a controlled amount of work and then enters a barrier. The other two programs used were successive over-relaxation (SOR) and Gaussian elimination. These programs are the intensive portion of larger applications; thus, they represent a typical bottleneck in the performance of some important applications. This section shows results for each of these programs using the different barriers.

In order to verify our claims, a user level scheduler was developed and linked into all programs. The scheduler was prevented from interfering with the user program by restricting it to a single processor and disallowing other processes to run on that processor. A shared data structure between the scheduler and the program provided the application with the current number of processors in its partition. Processes were created using the Silicon Graphics parallel programming primitives, which are adequate for expressing coarse grain heavy-weight process parallelism. A production quality implementation of our ideas could be accomplished by using a shared data structure between the kernel scheduler and the user application, an approach suggested by some modern multiprocessor operating systems [1, 9, 10].

The different implementations of barrier synchronization that we tested were: plain blocking, plain spinning, different competitive strategies, and the scheduler information policy. Other parameters included the time between scheduling decisions and the minimum number of processors allotted to an application. Another parameter that might have been important in this analysis was the quantum of processes within a partition. However, since the experiments indicated that only the plain spinning approach was affected by this, and this effect is well understood, we have omitted results that explore this parameter. The default quantum size of thirty milliseconds was used in the experiments.

In many cases speedup is used to measure the effectiveness of tuning parallel programs. However, in a multiprogramming environment, speedup is not well defined. To best illustrate the comparative performance of different barriers, our results are presented on a time per phase basis. For the synthetic program, we factored out the work delay, thus the time presented represents solely barrier overhead. For Gauss and SOR, the time presented includes work and barrier overhead. Time was determined by measuring wall time on dedicated processors.

The graphs in figures 1 through 6 show the performance of a synthetic program and Gauss elimination and SOR applications as the number of processes varies. For clarity, a pair of graphs is presented for each application. The first graph compares the performance of the different competitive algorithms with blocking, and the second graph compares spinning, blocking, the best competitive strategy, and the **scheduler information** policy. All three applications use barrier synchronization between their phases. The scheduling policy makes decisions every eighty milliseconds, and gives the application seven processors for five percent of the time. The other 95% of the time the application is given between two and six processors with equal probability. This effectively simulates a hardware partitioned environment with applications arriving and leaving at the rate of the scheduling decision frequency. By collecting the history of scheduling decisions, it was verified that the average number of processors in a partition agreed with the statistically expected result computed from the probabilities stated above. As can be seen, the competitive approaches outperform both spinning and blocking in many cases, and deviate very little from

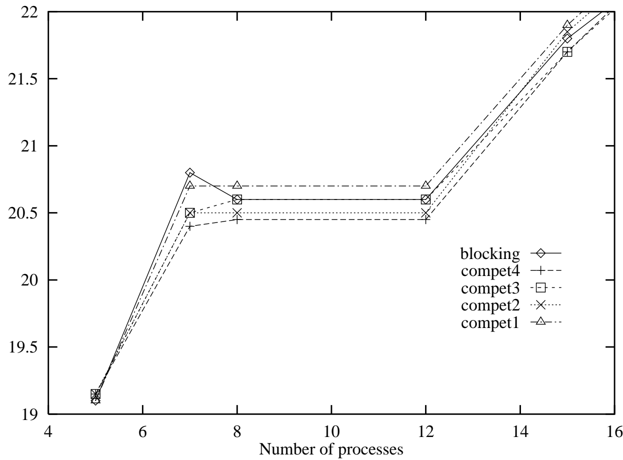


Figure 5: Competitive barriers for Gaussian elimination

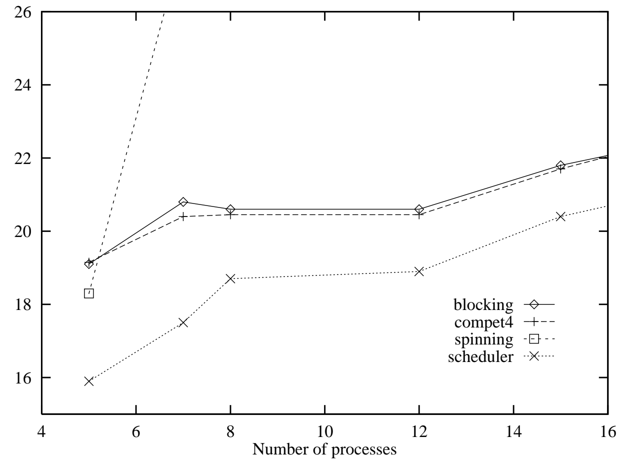


Figure 6: Scheduler information barrier for Gaussian elimination

the best of the two in the rest. Competitive spinning often outperforms both blocking and plain spinning in the cases of five and seven process applications because the scheduling policy will give the application anywhere between two and seven processors. When there are as many processors as processes, spinning outperforms blocking, otherwise the converse is true. Since the competitive approaches are able to adjust, they can outperform both plain spinning and blocking. When the number of processes increased to the point where there were always more processes than processors, the competitive approaches approximated blocking (with a small overhead) and were significantly better than spinning.

In addition to the above experiments, others were performed by varying the time between scheduling decisions from 80 to 800 milliseconds, by allowing the application full use of the multiprocessor for as little as five percent of the time or as much as fifty percent of the time, and by giving the application a small number of processors. While the results were quantitatively different, the qualitative behavior of the different barrier implementations remained the same. Thus, the graphs presented are representative of all our experiments.

The graphs in figures 7 and 8 show the performance of the barrier implementations across different periods of scheduling decisions. Varying times between scheduling decisions is different from varying the quantum of process execution, and that is why spinning is not affected as much as might be anticipated. As expected, the competitive approaches improve when the time between scheduling decisions increases. The reason is that the competitive approaches need a short period of time to adjust to the new environment after a scheduling action. When scheduling actions are in effect for longer periods of time, the adjustment cost is amortized over more barriers yielding better performance. Blocking is largely invariant to the time between scheduling decisions (figures 7 and 8). Spinning performs in a more complicated way. A possible explanation for this behavior is the interaction between scheduling decisions and quantum duration within a partition. This question wasn't pursued further, since spinning performs much worse than other policies and the qualitative analysis of a bad policy is not interesting.

The unexpected result was the small but steady improvement of the **scheduler information** policy when the time between scheduling decisions increased. While this policy should have been invariant to time between scheduling decisions, it is possible for the policy to err when a scheduling decision occurs at the same time the application is going through a barrier. In this case, some threads will use old information to guide their decision and thus may decide sub-optimally. When the time between scheduling decisions is large, this happens rarely, and is why the application has a small performance improvement. It is possible to correct this behavior by delaying scheduling decisions until the application has cleared the barrier but the performance benefit is too small to warrant this change.

5 Conclusion

We feel the work presented in this document has several interesting continuations. We are interested in examining the performance of scheduler information in conjunction with other synchronization primitives that may be adversely affected by multiprogramming, i.e., FIFO locks. We are also interested in studying the effect of scheduler information in systems where priorities are important, i.e., real-time applications. We believe that significant performance benefits can be achieved by sharing information between the kernel and application in such environments.

There are three primary contributions of this paper. We ex-

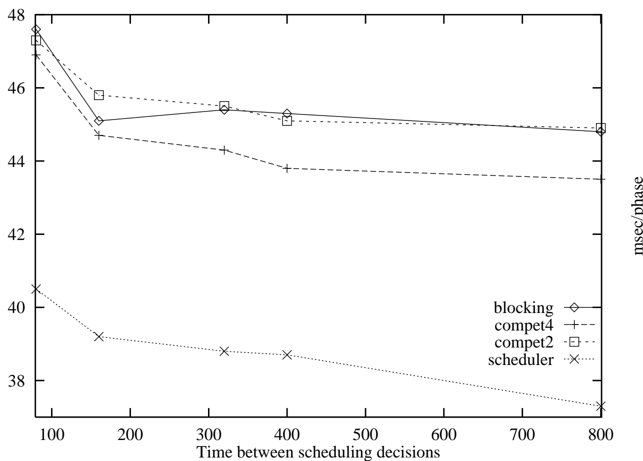


Figure 7: Effect of scheduling decision frequency on seven processors in SOR

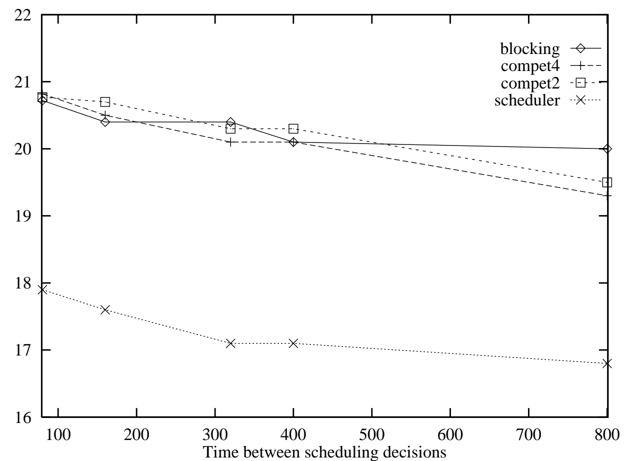


Figure 8: Effect of scheduling decision frequency on seven processors in Gaussian elimination

tended the work performed by Karlin *et al* on locks by showing these techniques perform well for barriers. We proposed **coarse adjustment**, another competitive spinning technique that performs better than the previous competitive spinning techniques applied to barriers. Finally, we proposed **scheduler information**, a new technique that outperformed all the previous techniques consistently and provided considerably better performance in many cases. This new technique is both optimal and easy to implement.

Acknowledgements

We would like to thank Michael Scott for his helpful comments on this document.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *Transactions on Computer Systems*, pages 53–79, February 1992. Also presented at the Thirteenth Symposium on Operating Systems Principles.

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

[3] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, December 1991.

[4] J. Edler, J. Lipkis, and E. Schonberg. Process management for highly parallel Unix systems. In *Proceedings of the USENIX Workshop on Unix and Supercomputers*, September 1988.

[5] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 120–132, May 1990.

[6] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 41–55, October 1991.

[7] Scott T. Leutenegger and Mary K. Vernon. Performance of multiprogrammed multiprocessor scheduling algorithms. In *Proceedings of the 1990 ACM SIGMETRICS conference in Measurements and Modeling of Computer Systems*, pages 226–236, May 1990.

[8] E. Markatos, M. Crovella, P. Das, C. Dubnicki, and T. LeBlanc. The effects of multiprogramming on barrier synchronization. In *Proceedings of the third symposium on parallel and distributed programming*, pages 662–669, December 1991.

[9] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium in Operating System Principles*, pages 110–121, 1991.

[10] Michael Scott, Thomas LeBlanc, and Brian Marsh. Design rationale for Psyche, a general purpose multiprocessor operating system. In *In Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[11] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 159–166, Litchfield Park, Arizona USA, December 1989. ACM.

[12] J. Zahorjan, E. D. Lazowska, and D. L. Eager. Spinning versus blocking in parallel systems with uncertainty. Technical Report TR-88-03-01, Department of Computer Science and Engineering - University of Washington, 1989.

[13] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 180–198, 1991.

[14] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 214–225, May 1990.

Appendix

```
global_t gsense,bar_count,numb_procs;
global_t SpinThreshold;
local_t lsense,count;
SpinThreshold = switch_time;
bar_count = AtomicAdd(bar_count,1);
if bar_count < numb_procs then
   for count = 1 to SpinThreshold do
       if lsense == gsense then
          lsense = 1 - lsense;
          exit barrier;
else
   bar_count = 0;
   gsense = 1 - gsense;
   wakeup blocked threads;
   lsense = 1 - lsense;
```

Figure 9: **Fixed Time** Competitive Barrier

```
global_t gsense,bar_count,numb_procs;
global_t SpinThreshold;
global_t tcount[numbprocs];
global_t ttime[numbprocs][3];
local_t lsense,count,now,tblkd,id;
bar_count = AtomicAdd(bar_count,1);
if bar_count < numb_procs then
   for count = 1 to SpinThreshold do
       if lsense == gsense then
          tblkd = 0;
          goto exit_barrier;
   now = GetCurrentTime();
   block;
   tblkd = GetCurrentTime() - now;
 exit_barrier:
   ttime[id][tcount[id]%3] = tblkd;
   tcount[id] = tcount[id] +1;
   if Average(tcount[id]) < switch_time
     then
      SpinThreshold = min(switch_time,
                SpinThreshold+Adjust);
     else
      SpinThreshold = max(0,
               SpinThreshold - Adjust);
   lsense = 1 - lsense;
else
   bar_count = 0;
   gsense = 1 - gsense;
   wakeup blocked threads;
   lsense = 1 - lsense;
```

Figure 10: **Average Three** Competitive Barrier

```
global_t gsense, bar_count, num_procs;
global_t SpinThreshold;
local_t lsense, now, t_blkd, count;

bar_count = AtomicAdd(bar_count,1);
if bar_count < num_procs then
   for count = 1 to SpinThreshold do
       if lsense == gsense then
          tblkd = 0;
          goto exit_barrier;
   now = GetCurrentTime();
   block;
   tblkd = GetCurrentTime - now;
 exit_barrier:
   if tblkd < 2*switch_time
    then
      SpinThreshold = min(switch_time,
              SpinThreshold + Adjust);
    else
      SpinThreshold = max(0,
              SpinThreshold - Adjust);
   lsense = 1 - lsense;
else
   bar_count = 0;
   gsense = 1 - gsense;
   wakeup blocked threads;
   lsense = 1 - lsense;
```

Figure 11: **Last One** Competitive Barrier

```
global_t gsense, bar_count, num_procs;
global_t SpinThreshold;
local_t lsense, now, t_blkd, count;
bar_count = AtomicAdd(bar_count,1);
if bar_count < num_procs then
   for count = 1 to SpinThreshold do
       if lsense == gsense then
          tblkd = 0;
          goto exit_barrier;
   now = GetCurrentTime();
   block;
   tblkd = GetCurrentTime - now;
 exit_barrier:
   if tblkd < 2*switch_time ;
    then
      SpinThreshold = switch_time;
    else
      SpinThreshold = 0;
   lsense = 1 - lsense;
else
   bar_count = 0;
   gsense = 1 - gsense;
   wakeup blocked threads;
   lsense = 1 - lsense;
```

Figure 12: **Coarse Adjustment** Competitive Barrier

```
global_t gsense, bar_count, numb_procs;
global_t numb_blocked, numb_processors;
local_t lsense;

bar_count = AtomicAdd(bar_count,1);
if bar_count < numb_procs
   if (numb_procs - numb_blocked)
            <= numb_processors then
      while lsense <> gsense do
         Spin;
   else
      block;
   lsense = 1 - lsense;
else
   bar_count = 0;
   gsense = 1 - gsense;
   wakeup blocked threads;
   lsense = 1 - lsense;
```

Figure 13: **Scheduler Information** Barrier

```
for i:=0 to NUM_PHASES do
  begin
    work();
    barrier();
  end
```

Figure 14: Code for a typical process in a barrier application

This code is available via anonymous ftp from cayuga.cs.rochester.edu (/pub/barrier.src.tar.Z).