The Effects of Multiprogramming on Barrier Synchronization[†]

Evangelos Markatos Mark Crovella Prakash Das
Cezary Dubnicki Tom LeBlanc
{markatos, crovella, prakash, dubnicki, leblanc}@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 380

August 1991

Abstract

One of the most common ways to share a multiprocessor among several applications is to give each application a set of dedicated processors. To ensure fairness, an application may receive fewer processors than it has processes. Unless an application can easily adjust the number of processes it employs during execution, several processes from the same application may have to share a processor. In this paper we quantify the performance penalty that arises when more than one process from the same application runs on a single processor of a NUMA (NonUniform Memory Access) multiprocessor. We consider programs that use coarse-grain parallelism and barrier synchronization because they are particularly sensitive to multiprogramming. We quantify the impact on the performance of an application of quantum size, frequency of synchronization, and the type of barrier used. We conclude that dedicating processors to an application, even without migration or dynamic adjustment of the number of processes, is an effective scheduling policy even for programs that synchronize frequently using barriers.

[†]This work was supported in part by NSF grant number CCR-9005633 and an NSF Institutional Infrastructure Program grant number CDA-8822724.

1 Introduction

For both uniprocessor and multiprocessor systems, efficient utilization of processor resources requires that applications share a machine. Many scheduling policies have been proposed and implemented for multiprogramming a multiprocessor [Black, 1990a; Leutenegger, 1990; Majumdar, 1988]. One of the simplest policies is for the system to give each application a set of dedicated processors [Black, 1990b; Tucker and Gupta, 1989; Zahorjan and McCann, 1990]. This policy, which we will call hardware partitioning, has low overhead, provides applications with guaranteed physical parallelism, minimizes the interactions between kernel and user-level scheduling, and preserves the affinity between processes and caches. However, this policy may require that the set of processors given to an application vary over time as new applications arrive and depart, and that the processes of an application share the processors dedicated to that application.

Since most performance studies of parallel applications assume a dedicated machine, there is little understanding of the performance impact of multiprogramming, even within hardware partitions. In this paper we investigate how parallel programs behave under a hardware partition policy, with an emphasis on programs that are likely to suffer performance degradation under such a policy. We consider programs that employ coarse-grain parallelism and barrier synchronization. Programs with coarse-grain parallelism offer few opportunities to adjust the number of processes during execution, and so are less able to adapt to a change in the number of processors within a partition. Barriers are a common form of synchronization that are particularly sensitive to multiprogramming [Zahorjan *et al.*, 1988].

Given a system scheduler that assigns each application its own hardware partition, we consider the following questions:

- Within a hardware partition, what type of barrier should be used? Is a scalable implementation preferable to a centralized implementation? Should processes spin or block?
- What barrier implementations are sensitive to the fact that the number of processors given to an application may change over time? Are there barrier implementations that are not particularly sensitive to such changes?
- What is the relationship between the frequency of synchronization and the overhead incurred as a result of multiprogramming with hardware partitions? Do programs that synchronize very frequently suffer disproportionately?
- If the scheduler has to take processors away from an application in order to allocate them to a new application, how many processors should be taken away?

These specific questions are part of the broader issue of how to multiprogram large-scale multiprocessors. Two of the best known scheduling policies are hardware partitioning and coscheduling. In comparison to hardware partitioning, the primary advantage of coscheduling is that all processes of an application execute simultaneously, eliminating context switch overhead in the presence of synchronization and communication. With hardware partitioning, an application that cannot dynamically adjust the number of processes it employs may be forced to multiplex several processes on a single processor within a small partition. Our focus in this paper is on the synchronization overhead incurred under these circumstances.[1] In particular, we would like to quantify the performance penalty suffered by applications that are *not* coscheduled, but are instead scheduled under some other reasonable policy, and describe those situations, if any exist, where the penalty is significant.

We address these questions using both simulation and experimental evaluation on a 16-processor BBN Butterfly, a NUMA (NonUniform Memory Access) multiprocessor. Although many of the questions we have enumerated have been addressed to some extent in the literature, previous work has assumed an UMA (Uniform Memory Access) multiprocessor. UMA machines, which typically have only a small number of processors, have a single global memory that facilitates migration for scheduling purposes. NUMA machines, on the other hand, scale to hundreds of processors, but have a distributed memory organization that makes migration prohibitively expensive for short-term scheduling. As a result of these architectural differences, the relative importance of migration for scheduling and scalability for synchronization are reversed for the two classes of machine.

The next section presents a survey of related work and lists the factors that distinguish our work. Section 3 describes our environment for experimentation and simulation. Section 4 presents our main results regarding spinning tree barriers, spinning centralized barriers, blocking tree barriers, and blocking centralized barriers under hardware partitioning. In particular, we show that for applications that use a particular form of blocking barrier, the performance penalty of processor deprivation (i.e., hardware partitions that are smaller than requested) is fairly small, even in the absence of migration and assuming frequent synchronization. We conclude, in section 5, that even programs that synchronize frequently with barriers do not require coscheduling, and therefore hardware partitioning is an effective scheduling policy for such programs on large-scale NUMA multiprocessors.

2 Related Work

Previous work has considered both the performance of hardware partitioning in comparison to other policies, and the impact of multiprogramming policies on barrier synchronization.

Tucker and Gupta [1989] have shown that dedicating processors to applications is particularly effective when each application is able to dynamically adjust the number of processes it requires during execution to match the number of processors it has been allocated. They attribute the performance degradation associated with time-sharing on a multiprocessor to the fact that there are more processes (virtual processors) than physical processors, and propose as a solution a process control system in which the number of processes always equals the number of processors. Processors are redistributed among applications periodically, and each application suspends or resumes processes to adjust to the number of available processors. This adjustment is practical within a programming model based on a global task queue since, in addition to its attractive load balancing properties, applications that use the task queue model do not depend on the number of processors available during execution.

Tucker and Gupta showed that dynamically adjusting the number of processes to equal the number of processors greatly improves performance when compared against a traditional time-sharing system. However the fact that hardware partitioning with process control is clearly better

[1] A complete comparison between hardware partitioning and coscheduling is beyond the scope of this paper, but is an area of active research [Gupta *et al.*, 1991; Crovella *et al.*, 1991]. In addition to synchronization overhead, such a comparison would consider cache affinity, memory affinity, contention, and the programming model.

than a naive centralized scheduler without process control does not imply that hardware partitions are a good general solution to the multiprogramming problem. Moreover, the process control scheme proposed by Tucker and Gupta requires that applications use the centralized task queue model.

Subsequent work by Gupta *et al.*[1991] describes a detailed simulation of a multiprogrammed execution of four programs on a 12-processor UMA multiprocessor under different scheduling policies. The simulations show that hardware partitioning outperforms other policies most of the time. Although these results provide evidence in favor of hardware partitions, the simulations are based on four specific applications, rather than general properties of applications. In particular, the results do not isolate the effect of frequent synchronization, so it is difficult to discern what applications, if any, are inappropriate for hardware partitions.

Leutenegger [1990] simulated several different scheduling policies for a mix of applications that synchronize using centralized barriers. He simulated a coscheduling policy [Ousterhout, 1982], a round-robin job policy [Leutenegger and Vernon, 1990], and a hardware partition policy. He concluded that both the coscheduling policy and the hardware partition policy (with a lightweight context switch among threads in an application) perform very well. His results for the hardware partition policy are optimistic however, since his simulations assume either no migration cost, or that the migration cost is equal to the context switch overhead. Neither of these assumptions is realistic, especially on NUMA machines. Moreover, the performance metric he used (average turnaround time) is very sensitive to the fairness properties of the scheduler, and yet the coscheduling policy he simulated was not fair to each application. As a result, it is difficult to distinguish performance effects caused by an unfair policy from performance effects due to other factors, such as synchronization, in his simulations.

Zahorjan, Lazowska and Eager [1988; 1989] examined the effect of the scheduling policy on the overhead of spinning, and compared spinning synchronization with blocking synchronization. They simulated the performance of a multiprogramming UMA machine with a centralized scheduler queue. They showed that spinning barriers introduce significant overhead under those circumstances. To reduce the overhead of spinning, they proposed a modified scheduler that does not schedule spinning threads. More generally, they argued that the decisions about how to allocate processors to jobs, and how to schedule the threads of a job on its processors, must be made cooperatively between the system and the application.

Our work differs from previous work in several important ways. First, our scheduling policies are heavily influenced by the fact that we use a NUMA multiprocessor for our experiments. In particular, our scheduling policies use processor-local ready queues and avoid migration, whereas other work has assumed global ready queues and zero migration costs. Second, our use of a NUMA machine results in a bias towards scalable synchronization primitives, while previous work on small-scale UMA machines does not distinguish between centralized and scalable implementations of synchronization. Third, we are particularly interested in a comparison between coscheduling and hardware partitioning, and we believe that the frequency of synchronization is one of the dominant considerations in this comparison, because coscheduling was specifically designed to support frequently synchronizing programs, while hardware partitions were not. Coscheduling rus *all* the threads of an application at the same time, while hardware partitions run only a subset of them simultaneously. Therefore, we would like to quantify the synchronization overhead as a function of the frequency of synchronization, and describe the range within which synchronization overhead becomes significant. Finally, we are interested in knowing whether hardware partitioning is a good general solution to the question of how to multiprogram a multiprocessor, and therefore we focus on the sources of overhead present under this particular multiprogramming policy. We argue that hardware partitioning is a reasonable solution by showing that even those programs most sensitive to multiprogramming (namely, programs with barriers that do not adjust the number of processes to match the number of processors) need not suffer serious performance degradation due to frequent synchronization under hardware partitioning.

3 Methodology

We will use a combination of simulation and experimentation to evaluate the effect of multiprogramming on programs that use centralized or tree barriers, either with spinning or blocking.

3.1 The Parallel Program Simulator

We built a parallel program simulator so that we could study the effects of scheduling policy on program execution. We simulate both a multiprogramming scheduling policy and the synchronization behavior of parallel programs during execution, and measure the system throughput. The simulator takes as input descriptions of applications and their arrival times, system parameters (such as the number of processors available), the particular scheduling policy to use, and scheduling parameters such as quantum size. The output of the simulation is the time needed to execute all jobs and, if requested, a graphical history of the simulated execution.

Application descriptions consist primarily of process profiles, which contain a sequence of events (computation and communication) for each process. Process profiles can be automatically generated according to a set of specified parameters, or collected during the execution of a parallel program. In our simulation studies we assume that process profiles are independent of the scheduling policy, which may not be true for all parallel programs.

3.2 Experimental Environment

All our experiments were carried out on a 16-node BBN Butterfly multiprocessor running the Psyche operating system [Scott *et al.*, 1990]. Under Psyche an application may create an arbitrary number of virtual processors (kernel processes), corresponding to the desired physical parallelism. A virtual processor is bound to a single physical processor for its lifetime and is assigned the cpu by the kernel according to priority; within a priority level, round-robin scheduling is used. The kernel also implements software timers and interrupts that can be used to implement time-slicing in user space.

In our experiments we use a thread package to allocate one virtual processor for each physical processor in an application's partition. Within a single virtual processor, the thread package uses software timer interrupts to schedule multiple threads of control using round-robin time-slicing. Scheduling operations within the thread package occur as a result of software timer expiration, or when a thread yields the processor voluntarily. In many cases, scheduling decisions caused by synchronization are implemented entirely in user space, thereby avoiding the overhead of context switching through the kernel.

In general, the Psyche kernel time-slices virtual processors from different applications onto a physical processor, while the thread package time-slices threads from a single application onto a virtual processor. Under hardware partitioning however, the kernel scheduler has little work to do, since no two applications share a processor. Nearly all scheduling decisions are made by the thread package, which resembles a traditional time-slicing environment. Thus, although our experiments employ a thread package and a user-level preemptive scheduler, our results apply equally well to programs that are implemented by more traditional heavyweight processes and preemptive scheduling by the kernel[2]. We will, therefore, use the terms *thread* and *process* interchangeably.

Our applications employ long-running user-level threads created and managed by the thread package. Each thread is assigned to a single virtual processor and is never migrated, unless a physical processor (and the corresponding virtual processor) is taken away from the application by the multiprogramming scheduler. We do not use thread migration for short-term load balancing because the cost of migration on NUMA machines is very high (on the order of milliseconds), even for lightweight threads. Since we assume frequent barrier synchronization (every several milliseconds or so), migration would be needed at each synchronization point to achieve short-term load balancing. Frequent migration would be quite costly and unlikely to be of much benefit.

All of our example applications create some number of threads that compute and periodically synchronize using a barrier. In our experiments, we vary both the frequency of synchronization and the type of barrier used. Each application is given fewer processors than it has threads, so more than one thread is forced to share a processor.

3.3 Barrier Types

Our study is based on application programs that synchronize using barriers. There are many different implementations of barriers however, and our experience has shown that the specific barrier used can affect performance. For this reason, we consider both centralized and tree barriers, and spinning and blocking implementations of each.

Centralized barriers

Centralized barriers use a global counter. Each thread that arrives at the barrier increments the counter, and waits for the other threads to reach the barrier. When the counter is equal to the number of threads that participate in the barrier, all threads are free to proceed.

Tree barriers

Tree barriers are representative of a large class of barriers whose completion time is logarithmic in the number of participants. This class includes tree barriers [Yew *et al.*, 1987; Mellor-Crummey and Scott, 1991], tournament barriers [Hensgen *et al.*, 1988a; Lubachevsky, 1989], and butterfly barriers [Brooks, 1986; Hensgen *et al.*, 1988b]. Tree barriers use a combining tree to record the arrival and departure of threads. Our implementation, based on work by Mellor-Crummey and Scott [Mellor-Crummey and Scott, 1991], uses a 4-way tree to combine the notification of arrivals.

[2] To ensure that our conclusions do not depend on a particular implementation of threads or context switching, several of our results are presented as a function of the context switch time.

Each thread is represented by a node in the tree. Each interior node waits for all its children to arrive at the barrier, and then informs its parent that it has also arrived at the barrier. When the root is so informed, it releases its children, and these in turn release their children, until all the leaves are released. The barrier combining tree is a 4-way tree because each processor can pack the information about its four children into one word and query the state of all four children with a single comparison. To minimize the time to wake up threads, we use a binary tree to propagate barrier completion information to the children.

Combination Barriers

Combination barriers [Axelrod, 1986] were originally suggested as a way to minimize the number of locks needed in the implementation of a tree barrier. Combination barriers incorporate both tree and centralized barriers. A centralized barrier is used for synchronization among the threads on a single processor; a tree barrier is used for synchronization across processors. If an application has as many processors as threads, then combination barriers behave exactly like tree barriers. If there are more threads than processors, then all threads on the same processor participate in a local centralized barrier. The last thread on each processor to reach the local centralized barrier participates in a tree barrier with the other processors. After the tree barrier has been completed, all the waiting threads on each processor are released from the local barrier.

Centralized barriers are easy to implement and are particularly efficient in the absence of contention. Access to the counter serializes execution however, and can cause significant performance degradation in the presence of contention. Tree barriers are more complicated and less efficient for a small number of participants, but they do not serialize execution and their performance scales logarithmically with the number of processors [Mellor-Crummey and Scott, 1991]. Combination barriers use the efficient implementation of centralized barriers on a single node, where there is no contention, and the scalable implementation of tree barriers across processors, where there is likely to be contention.

3.4 Spinning vs. Blocking

While a thread waits for others to reach a barrier, it can either spin or block. In the case of spinning barriers, a thread periodically checks to see if all other threads have reached the barrier. A thread will continue to spin until all threads reach the barrier or until it is preempted because of quantum expiration. In the case of blocking barriers, when a thread needs to wait for an event, it yields the processor to another thread of the same application running on the same processor. As an optimization, a thread might yield the processor only if there is another thread with which it shares the processor, spinning otherwise. We will assume this implementation of blocking barriers because it has all the advantages of spinning when there are as many processors as threads, and all the advantages of blocking when there are more threads than processors.

4 Barrier Performance Under Processor Deprivation

Our goal is to quantify the effect on application performance of a multiprogramming policy that allocates a barrier program fewer processors than it needs. The results depend both on the frequency of synchronization within the program and the number of processors allocated by the scheduler.

4.1 The Effects of the Frequency of Synchronization

Barrier programs with very frequent synchronization are natural candidates for coscheduling or a dedicated machine policy. If such a program is given fewer processors than it has processes, then every barrier will incur overhead in the form of spinning or context switching. Here, we quantify that overhead for the different implementations of barriers.

In order to investigate the additional overhead introduced by processor deprivation, we constructed an artificial program that creates M threads on top of $P(M > P)$ processors and schedules them according to a round-robin policy with local ready queues. All threads compute for the same amount of time, synchronize through a barrier, and then repeat the process. The code fragment for a thread is:

```
for (int i = NUM_BARRIERS ; i>0 ; i--) {
    for (int j = MAX_DELAY ; j>0 ; j--) ;  // compute
    barrier_synchronization() ;         // synchronize
}
```

We vary the `MAX_DELAY` variable to change the frequency of synchronization. We vary the implementation of the `barrier_synchronization` function to reflect the different types of barriers.

We chose to have all the threads compute for the same time between successive barriers because balanced computations of this form are a worst-case scenario for a hardware partition scheduler (when compared to coscheduling). While balance in parallel programs is usually preferred, programs with imbalance in the computation can overlap computation with synchronization, reducing the effects of processor deprivation.

In all of our experiments, an application creates 16 threads. These applications are then run on fewer than 16 processors to measure the effects of processor deprivation.

Spinning Barriers

Figures 1 and 2 plot the performance of an application with 16 threads on 15 processors for both tree and centralized spinning barriers. The data for figure 1 were produced by parallel execution of our example program on the Butterfly; the data for figure 2 were obtained by simulation. In each figure, the vertical axis shows the average completion time of a barrier (i.e, the average time to execute the spin loop that simulates computation in our sample program plus the time to synchronize through one barrier). We use this normalized measure of running time because it does not depend on the number of barriers a program has, but only on the frequency of synchronization and the type of barrier used. The horizontal axis shows the computation time between barriers (i.e, the time to execute the spin loop that simulates computation in our sample program).

Figure 1 suggests that centralized barriers are better than tree barriers in nearly all cases. This result is somewhat surprising since, in the absence of processor deprivation, tree barriers perform better than centralized barriers when more than a few processors are involved [Mellor-Crummey and Scott, 1991]. This anomaly can be explained by considering the implementation of tree barriers. Tree barriers have more than one synchronization point, corresponding to the internal nodes of the tree. Information is passed up the tree as processes arrive at a barrier, and passed down again as processes are released from the barrier. Each node in the tree is effectively a barrier for its children,

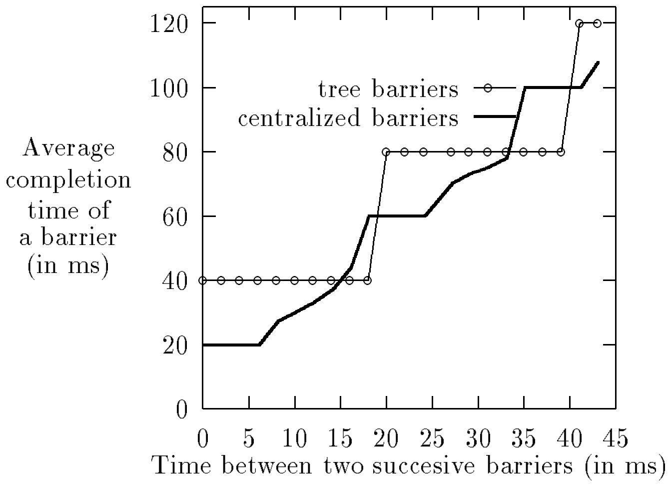

Figure 1: Measured execution time of tree versus centralized spinning barriers; 16 threads on 15 processors; quantum size = 20 ms.

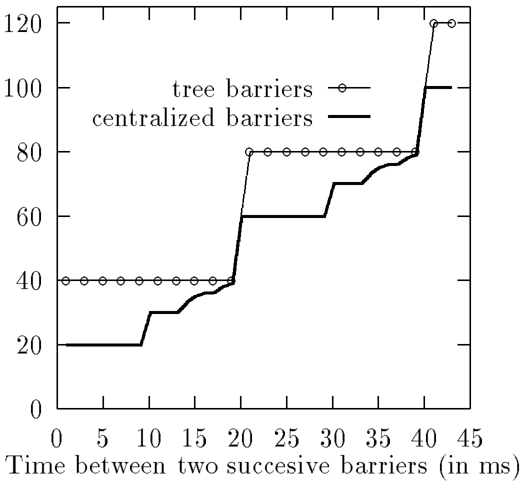

Figure 2: Simulation of tree versus centralized spinning barriers; 16 threads on 15 processors; quantum size = 20 ms.

which must be completed before a node higher up in the tree can be notified. If a process within the tree is not running due to processor deprivation, no other process on the path up to the root can participate in the barrier. Processes close to the root may have to spin while waiting for processes lower in the tree to receive a quantum. This delay, which could be as much as a whole quantum, can be introduced between every level in the tree.[3] Since we use a 4-way tree to pass information upwards, there are three levels in the tree used for our 16 thread application, and therefore two context switches can occur between levels. Figure 1 shows that even if the computation time between two successive barriers is very small, the time to complete a tree barrier is still 40 ms (two quanta). When the time between two successive barriers is less than the quantum size, most of the quantum is spent spinning. If we slightly increase the amount of useful work done between two successive barriers, then less time is spent spinning, but the completion time of the program (in terms of quanta needed) is the same. In general, if the quantum is Q and the time between barriers is T, it takes P processes $\lceil \log_4 P \rceil \times \left\lceil \frac{T}{Q} \right\rceil$ quanta to complete a tree barrier.[4]

A centralized barrier, on the other hand, takes only 20 ms (one quantum) to complete when the computation time between two successive barriers is less than half the quantum. Since a centralized barrier contains only one synchronization point, the last process to reach a barrier can continue on

[3] Adaptive tree barriers [Gupta and Hill, 1989] do not introduce a delay at every level in the tree because all processes reside at the leaf nodes. The implementation of adaptive tree barriers in [Gupta and Hill, 1989] would not be efficient on a NUMA machine however, since it involves excessive locking and remote spinning.

[4] In the case of 16 threads on 15 processors, the particular assignment of threads to processors is not important, since at most two context switches will be required to release at least one of the threads sharing a processor. However, if more threads or fewer processors are involved, the assignment of threads to processors can affect the number of context switches per barrier. To minimize context switching, interior nodes (or nodes close to the root of the tree) should not share a processor.

to the next barrier within the same quantum. After the first quantum of execution, a barrier is completed every quantum.

When the time between two successive barriers is more than half the quantum, the function that describes the average completion time of a barrier is more complicated. Let Q denote the quantum length, T the compute time between barriers, and let $R = \left\lceil \frac{1}{(Q/T)-1} \right\rceil$. When the time between two successive centralized barriers is more than half the quantum, the completion time of a program with N barriers is closely approximated by $NT\frac{(2R-1)}{R}$. The observed step-function is due to the integer ceiling function used to define R. (Details of the derivation of this formula can be found in the appendix.)

Figure 1 shows that tree barriers are better than centralized barriers when the time between two successive barriers is a little less than a multiple of the quantum size. In our simulation results, shown in figure 2, tree barriers are never better than centralized barriers, but the difference in performance is close to zero when the time between two successive barriers is a multiple of the quantum size. Our simulation results do not completely agree with our experimental results because the simulator does not model two important factors in the implementation of centralized barriers:

1. The time to complete a centralized barrier is assumed to be zero in the simulation. Thus, the simulation does not accurately model the linear-time complexity of centralized barriers.

2. The simulator does not model memory contention, which is much greater in the case of centralized barriers than with tree barriers.

Both of these factors cause centralized barriers to perform worse in our experiments than in our simulation.

Blocking Barriers

Blocking barriers can be used to avoid the unpredictable performance of spinning barriers in a multiprogrammed environment. With blocking barriers, a thread that waits at a barrier yields the processor to another thread of the same application. In our version of blocking barriers, a thread yields the processor only if there is another thread of the same application on the processor; otherwise, the thread spins. Blocking barriers are known to significantly reduce spinning time in the presence of multiprogramming [Zahorjan *et al.*, 1988], improving the percentage of time processors perform useful work, but not necessarily reducing the completion time of a program that frequently yields the processor. However, when used in combination with a hardware partitioning policy, blocking barriers can improve the completion time of a program, since the processor is always given to another thread in the same application.

Figure 3 shows the results of running our test program with 16 threads on 15 processors using blocking tree barriers. Unlike spinning barriers, the performance of blocking barriers is a smooth function of the frequency of synchronization. This figure quantifies the expected superiority of blocking over spinning under processor deprivation.

It is not surprising that blocking barriers perform much better than spinning barriers under processor deprivation. However, blocking barriers introduce overhead in the form of context switching. The context switch overhead may vary from as little as 11 μs [Massalin and Pu, 1989] to 750 μs

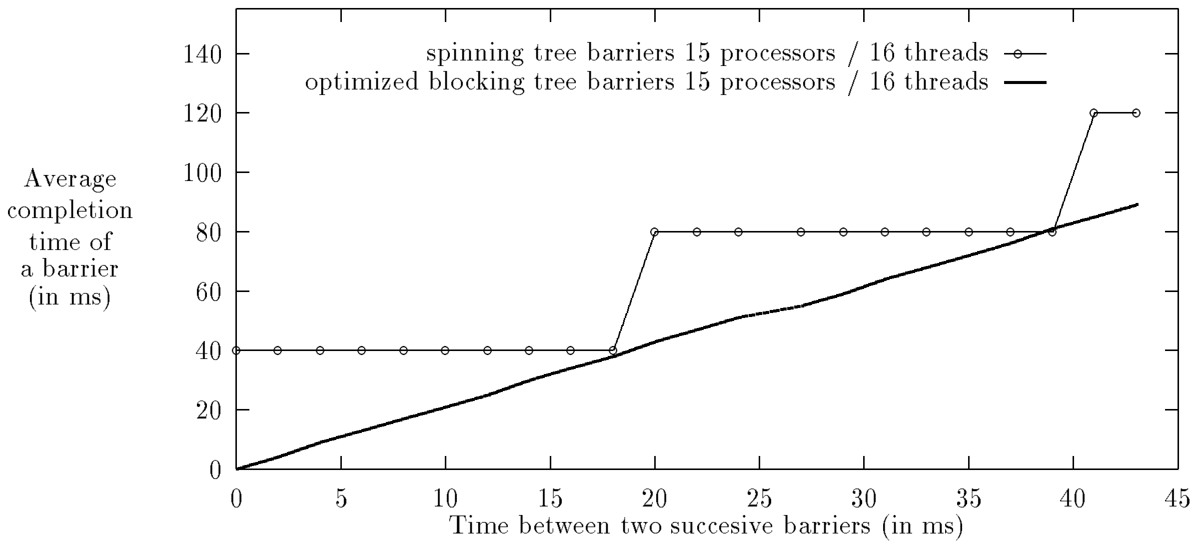

Figure 3: Measured execution time of spinning versus blocking tree barriers.

[Zahorjan and McCann, 1990] or more, depending on many factors, such as whether or not a kernel trap is required, queue manipulation overhead, the cost of saving and restoring registers, and the speed of the processors. If a single context switch is expensive, and an application synchronizes frequently, then the overhead introduced by processor deprivation could be high.

To quantify the importance of context switch overhead for blocking barriers, we ran our example program with 16 threads on 16 processors and then on 8 processors (the results for 4, 2 and 1 processor are similar). On 16 processors, no context switch overhead is incurred. On 8 processors, there is a context switch at every barrier. Absent context switch overhead, we would expect 8 processors to take twice as long as 16 processors to complete the program.[5] Any additional time can be attributed to context switch overhead. Figure 4 shows the percentage of the application's completion time attributed to synchronization overhead. The overhead is plotted as a function of the time between two successive barriers, expressed in multiples of the context switch time.

We see from figure 4 that, as expected, the overhead of synchronization is inversely proportional to the computation time between barriers (i.e., the frequency of synchronization). More precisely, the overhead is proportional to the number of barriers a program has, the number of levels in the barrier tree, and the context switch cost. As shown in figure 4, when the time between two successive barriers is more than 20 times the cost of a context switch, the overhead incurred by context switching within blocking barriers is less than 10% of the total execution time of the program. For a typical thread package with a context switch overhead of $100\mu s$ or less, this means that the time between two successive barriers has to be less than 2 ms for the overhead to be more than 10%.

[5] Our program exhibits linear speedup, and does not adjust the number of processes (or threads) to match the number of processors.

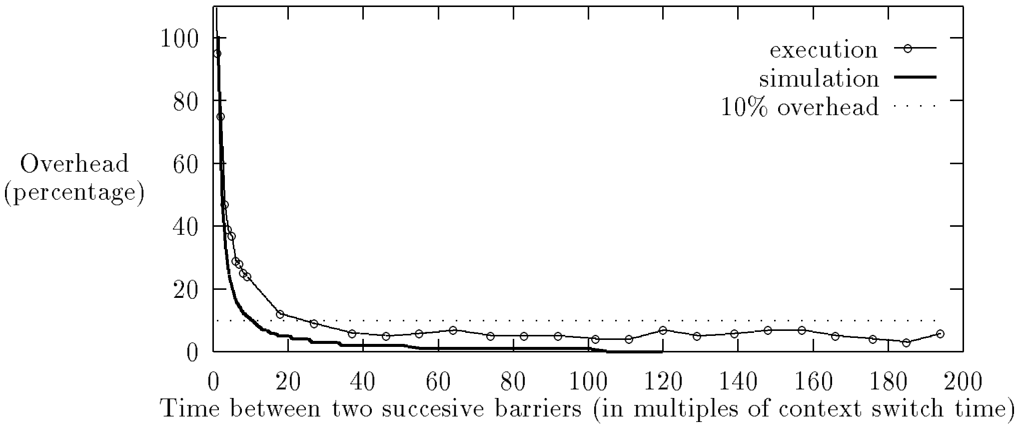

Figure 4: Measured overhead of blocking tree barriers for 16 threads on 8 processors.

Combination Barriers

The overhead introduced by blocking tree barriers is proportional to the number of levels in the tree since, in the worst case, a context switch may be required at each level. As we saw in figure 4, if synchronization is very frequent, or if the cost of a context switch is high, the overhead incurred due to context switching can be quite high, as much as 158% in extreme cases. Centralized barriers induce fewer context switches, but suffer from overhead introduced by contention. Combination barriers offer a compromise solution, without the contention problems of centralized barriers, or the additional context switching of tree barriers.

To measure the overhead of context switching with combination barriers, we again ran our example program with 16 threads on 16 processors and then on 8 processors, this time using combination barriers. As in figure 4, we plotted the percentage difference in the completion time of the application on 8 processors and 16 processors. The results are shown in figure 5.

The results are similar to those in figure 4, but the scale is quite different. Since combination barriers have many fewer context switches than tree barriers (in the worst case), the synchronization overhead of combination barriers is much smaller than the synchronization overhead for tree barriers. In fact, the overhead of combination barriers drops very quickly towards zero as the frequency of synchronization decreases. Figure 5 shows that if the time between two successive barriers is as little as 4 times the cost of a context switch, then the synchronization overhead is still less than 10%. If the time between two successive barriers is 20 times the cost of a context switch, then the synchronization overhead is practically zero. In contrast, the overhead of tree barriers is 10% when the time between barriers is 20 times the cost of a context switch, and 39% when the time between barriers is only 4 times the cost of a context switch.

Figures 4 and 5 allow us to draw some conclusions about the benefits of coscheduling, particularly in comparison to hardware partitions. In figure 5 we see that if the time between two successive barriers is less than the time to perform a context switch, then the overhead due to synchronization can be as high as 30%. In such cases coscheduling may be preferable to hardware partitions. If we

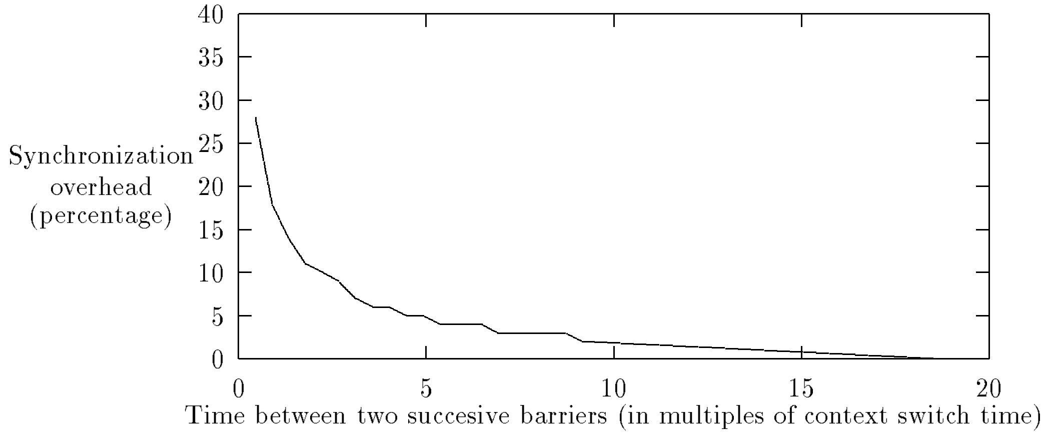

Figure 5: Measured overhead of blocking combination barriers for 16 threads on 8 processors.

use combination barriers, and the time between two successive barriers is more than 3 times the cost of a context switch, then the synchronization overhead is rather small. Therefore, we see that the synchronization overhead for medium- and coarse-grain parallel programs, which represent the majority of current parallel applications, is negligible. For example, a 512×512 Gaussian elimination program (which has 512 barriers) on 16 processors synchronizes every 45 *ms* on a BBN Butterfly Plus, which is more than 450 times the thread context switch overhead. Applications such as Dijkstra's shortest path algorithm and odd-even sort synchronize every 5-10 *ms* on a BBN Butterfly for reasonable input sizes. Thus, it seems that although there are cases where coscheduling could have an advantage over hardware partitions (namely, very fine-grain synchronizing applications), those cases are quite rare. For most applications, hardware partitions do not impose noticeable overhead, provided that combination barriers are used.

4.2 The Effect of the Number of Processors

Our results so far are based on the assumption that a program is allocated one fewer processor than it needs. This assumption results in the worst case situation where all processors except one are idle, either spinning or waiting. We now consider what happens as we vary the number of processors allocated to an application.

Since we do not allow migration, we do not expect the completion time of an application to be a smooth function of the number of processors allocated to it. For example, the completion time of an application with 16 threads running on 15 processors is about the same as the completion time of the same application running on 8 processors. Figure 6 shows the measured completion time (per barrier) as a function of the number of processors given to the application, for both blocking centralized barriers and blocking tree barriers.

Intuition suggests that the completion time of the application does not vary as we vary the number of processors allocated to the application between two successive divisors of the number of

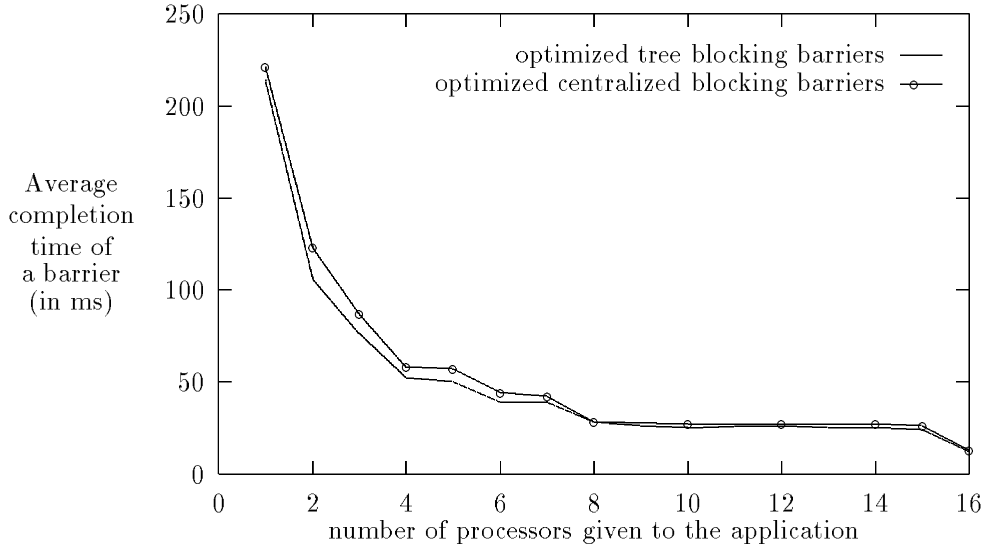

Figure 6: Effect of the number of processors on completion time (16 threads).

threads.[6] This intuition is not entirely correct however. In the absence of migration, the completion time of an application changes only when the maximum number of threads on any processor changes. Figure 6 shows there is a change in completion time for both types of barriers when the number of processors given to an application changes from 16 to 15, from 8 to 7, from 6 to 5, from 4 to 3, from 3 to 2, and from 2 to 1. This result can be phrased as follows:

> If P processors are to be given to an application that has M threads that are never migrated, then the completion time of the application would be the same if it was given P_M processors where P_M is the smallest integer less than or equal to P such that $\lceil M/P_M \rceil = \lceil M/P \rceil$.

This result suggests a need for close cooperation between the kernel and user-level thread management software in order to minimize the completion time of an application.[7] The kernel should give processors to applications that can use them, and should not take away processors from applications that really need them. Figure 6 states that if an application has 16 threads on 3 processors, an additional processor would improve completion time by 33%. The same application on 8 processors would not be helped by 7 additional processors. Given that the kernel may not be aware of thread creation and destruction, some form of cooperation between the thread library and the kernel is needed to match the number of processors allocated to an application with the needs of the application.

[6] Zahorjan, Lazowska, and Eager [1988] suggest that the number of processors given to an application that uses spinning barriers evenly divide the number of threads in the application, so as to eliminate spinning. However, perfect balance across processors only eliminates spinning when the time between two successive barriers is equal to the quantum size, and in any case, does not minimize completion time.

[7] Zahorjan, Lazowska, and Eager [1989] call for the same degree of cooperation to minimize spinning time.

5 Conclusions

We investigated the performance of coarse-grain parallel applications that synchronize frequently using barriers, under the assumption that an application is given fewer processors than it needs. We evaluated the overhead present in alternative implementations of barriers, and found that if an appropriate implementation is used (combination barriers in particular), applications need not suffer a significant performance penalty if they are given fewer processors than they have threads.

More specifically we showed that:

- *Spinning centralized barriers often outperform spinning tree barriers in the presence of multiprogramming, even though tree barriers are clearly preferable on a dedicated machine.* In general, when several threads from the same application share a processor, different implementations of barrier synchronization result in a wide variation in performance.

- *Tree barriers in the presence of multiprogramming may incur overhead proportional to the number of levels in the tree.* Barriers with more than one synchronization point in their implementation may introduce a context switch at every synchronization point.

- *The completion time of a barrier program depends on the maximum number of threads assigned to any one processor.* When processors are redistributed among applications, the scheduler should not blindly remove one processor from an application, since it may be possible to remove several more without affecting the completion time of the program. Also, to minimize completion time, the number of processors need not evenly divide the number of threads.

- *Dedicating processors to an application is an effective scheduling policy even for inflexible programs that synchronize frequently using barrier synchronization.* It is possible to achieve reasonable performance without thread migration and without dynamically adjusting the number of threads to match the number of processors. As long as threads use appropriate blocking barriers, and assuming that the time between barriers is more than four times larger than the context switch time (which is almost always true), no significant performance penalty must be paid as a result of multiplexing the threads of an application on a processor.

Although we have not considered coscheduling in our experiments, our results indicate that even programs that synchronize frequently using a barrier will not suffer a significant performance penalty if they are not coscheduled. Unless the context switch cost is very high, or the frequency of synchronization extremely high, coscheduling is not necessary. Given that coscheduling has an expected efficiency of around 80% [Ousterhout, 1982], there seem to be few situations where coscheduling would be preferred over hardware partitioning.

Finally, although our experimental work used a NUMA multiprocessor, which has very high migration costs, we believe many of our results also apply to UMA multiprocessors. Recent work has shown that migration should not be considered free in an UMA machine because the cache reload costs can be high [Mogul and Borg, 1991; Squillante and Lazowska, 1990]. The processor-local ready queues used in our experiments on a NUMA machine can also be used to avoid cache reload costs on an UMA machine, avoiding migration for short-term scheduling in both cases.

References

[Axelrod, 1986] T. S. Axelrod, "Effects of Synchronization Barriers on Multiprocessor Performance," *Parallel Computing*, 3:129–140, 1986.

[Black, 1990a] D. L. Black, *Scheduling and Resource Management Techniques for Multiprocessors*, PhD thesis, Carnegie Mellon University, July 1990.

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

[Brooks, 1986] E. D. Brooks, "The Butterfly Barrier," *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[Crovella *et al.*, 1991] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos, "Multiprogramming on Multiprocessors," *submitted for publication*, 1991.

[Gupta *et al.*, 1991] A. Gupta, A. Tucker, and S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications," In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, May 1991, to appear.

[Gupta and Hill, 1989] R. Gupta and C. R. Hill, "A Scalable Implementation of Barrier Synchronization Using An Adaptive Combining Tree," *International Journal of Parallel Programming*, 18(3):161–180, June 1989.

[Hensgen *et al.*, 1988a] D. Hensgen, R. Finkel, and U. Manber, "Two Algorithms for Barrier Synchronization," *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[Hensgen *et al.*, 1988b] Debra Hensgen, Raphael Finkel, and Udi Manber, "Two Algorithms for Barrier Synchronization," *International Journal of Parallel Programming*, 17(1):1–17, 1988.

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

[Leutenegger and Vernon, 1990] S. T. Leutenegger and M. K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies," In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 226–236, May 1990.

[Lubachevsky, 1989] B. Lubachevsky, "Synchronization Barrier and Related Tools for Shared Memory Parallel Programming," In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II:175–179, August 1989.

[Majumdar, 1988] S. Majumdar, *Processor Scheduling in Multiprogrammed Parallel Systems*, PhD thesis, University of Saskatchewan, 1988.

[Massalin and Pu, 1989] H. Massalin and C. Pu, "Threads and Input/Output in the Synthesis Kernel," In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 191–201, Litchfield Park, AZ, December 1989.

[Mellor-Crummey and Scott, 1991] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Transactions on Computer Systems*, 9(1), 1991, (to appear), Earlier version published as TR 342, University of Rochester, Computer Science Department, April 1990.

[Mogul and Borg, 1991] J.C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, CA, Apr 1991.

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

[Scott *et al.*, 1990] M.L. Scott, T.J. LeBlanc, B.D. Marsh, T.G. Becker, C. Dubnicki, E. Markatos, and N. Smithline, "Implementation Issues for the Psyche Multiprocessor Operating System," *Computing Systems*, 3(1):101–137, Winter 1990.

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

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

[Yew *et al.*, 1987] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers*, 36(4):388–395, April 1987.

[Zahorjan *et al.*, 1988] J. Zahorjan, E. D. Lazowska, and D. L. Eager, "Spinning Versus Blocking in Parallel Systems with Uncertainty," Technical Report 88-03-01, Computer Science Department, University of Washington, March 1988.

[Zahorjan *et al.*, 1989] J. Zahorjan, E.D. Lazowska, and D.L. Eager, "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Processors," Technical Report 89-07-03, Computer Science Department, University of Washington, July 1989.

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

Appendix

This appendix considers the running time of a barrier-synchronizing application that has more threads than processors. In particular, we consider the case where T threads are run on P processors, such that $T/2 \leq P < T$. In this case, each processor runs either one or two threads, and at least one processor has two threads. The analysis presented here could be extended to the case where even fewer processors are used; the effect would be to increase the maximum number of threads on a processor. The barriers are centralized, and threads spin when they wait at a barrier. For the purposes of analysis we assume that the barrier code takes no time to execute, so that all we are concerned with is computation time and time spent waiting at the barrier.[8] We model a thread as an execution of this code:

```
for (j = 0 ; j < NUM_PHASES; j++) {
        for (i = 0 ; i < DELAY ; i++)
                ;        /* this step models computation time */
        barrier();
}
```

The threads can be separated into two groups, A and B, where group A executes during odd quanta and group B executes during even quanta. Because the threads spin at a barrier, context switching only occurs due to quantum expiration. We assume for clarity of discussion that quanta expire at the same time on each processor; however relaxing this assumption would not change the results. In the first quantum, all the threads of group A are run; in the next quantum all the threads of group B are run; this continues in an alternating manner. All threads within a group advance identically, so we can refer simply to the advancement of a thread group. To quantify the progress of a group's computation, we define the function

$$t_A = \texttt{j} + (\texttt{i}/\texttt{DELAY})$$

where `i` and `j` are the values of the variables in any thread of thread group A. We define the function t_B similarly. The t function measures the progress of a thread group in units of "interbarrier times." That is, if $t_A = x$, where x is an integral value, then thread group A has just completed barrier number x. Additionally we define $t_{A,n}$ to be the value of t_A at the end of quantum number n.

We measure the quantum length Q in the same units. Thus if $Q = 1$, there is one barrier per quantum. If $Q = 2$, there are two barriers per quantum, etc. The complex portion of an application's behavior occurs when Q lies between 1 and 2, as can be seen from figure 1.

The overall computation proceeds as follows. During each quantum, the currently executing thread group moves past the currently pre-empted group. Thus if group A was executing during quantum n, then $t_{A,n} > t_{B,n}$. Since group B made no progress during quantum n, we can additionally say that $t_{A,n} > t_{B,n-1}$. The amount of progress that the current group makes depends on the progress that the pre-empted group made in the last quantum. The current group either:

1. runs to the end of its quantum, or

[8] The effect of this assumption can be seen in the difference between the simulation and the execution experiments for this situation. As shown in Figures 1 and 2, the overall behavior of the application does not change qualitatively as a result of this assumption.

2. reaches a barrier that is one past the last barrier acheived by the pre-empted group. In this case, the current group spins for the the remainder of its quantum.

Based on this observation, we can model the progress of the computation for thread group A as:

$$t_{A,n} = \min(\lfloor t_{B,n-1} \rfloor + 1, t_{A,n-2} + Q).$$

$\lfloor t_{B,n-1} \rfloor$ is the last barrier that thread group B reached in its quantum. Adding 1 to $\lfloor t_{B,n-1} \rfloor$ gives the limit imposed by the barriers on thread group A's progress. $t_{A,n-2} + Q$ is the limit imposed by quantum expiration. The min function reflects the rule that thread group A computes for the entire quantum unless it is stopped at the barrier.

The initial conditions are: $t_{A,0} = 1.0$, because A is assumed to go first, and it hits the first barrier during its quantum; and $t_{B,1} = Q$, because group B passes through the barrier (now completed) which A stopped at and then runs to the end of the quantum.

This model can be recast as the recurrence relation:

$$t_n = \min(\lfloor t_{n-1} \rfloor + 1, t_{n-2} + Q),$$

where $t_n = \max(t_{A,n}, t_{B,n})$, with initial conditions $t_0 = 1.0$, $t_1 = Q$. This recurrence relation describes the same computation as the formula using $t_{A,n}$ because the thread groups make progress in strict alternation.[9]

Solving the general recurrence for t_n will give us a formula for how many barriers can be completed in n quanta. Of course, what is actually measured in experiments such as shown in figure 1 is the running time of an application with a fixed number of barriers. That is, figure 1 shows a plot of the function:

$$f(Q) = y \mid t_y = 500.$$

$f(Q)$ is the inversion of the function t_n. Although the solution to the recurrence is not directly invertible, it can be approximated, allowing us some insight into the complex step-function nature observed in the figure.

The recurrence behaves as follows, for $1 < Q < 2$. The recurrence forms a repetition of a pattern of successive increments. The first thread group acheives a barrier, then the second thread group achieves the same barrier and runs on to the end of the quantum. It thus performs some extra computation. The first time this happens, the extra computation is $Q - 1$. It is thus $Q - 1$ "ahead" of the first thread set. The first thread set then runs to the next barrier. The second thread set then runs past the same barrier and uses up its quantum. It is then ahead by $2(Q - 1)$. When eventually the $Q - 1$ terms add up to 1, the second thread set acheives a barrier, and stops. The thread sets' roles are then reversed, and the same pattern repeats. The number of barriers completed during one repetition of this pattern is

$$R = \left\lceil \frac{1}{Q-1} \right\rceil$$

[9]This recurrence agrees with figure 1 for $Q > 2$ also. Figure 1 shows that when $Q > 2$, the running time of the application is the product only of the number of barriers and the length of the quantum in seconds; the application passes through exactly one barrier each quantum. The recurrence agrees with the figure because when $Q > 2$, $\lfloor t_{n-1} \rfloor$ is always less than $t_{n-2} + Q$; so the recurrence reduces to $t_n = t_{n-1} + 1 = n + 1$.

The value of R determines how quickly the application runs. The application always acheives R barriers every $2R-1$ quanta. For shorthand we will use L for $2R-1$.

The complete solution to the recurrence is then:

$$t_n = \left\lfloor \frac{n}{L} \right\rfloor R + \begin{cases} \frac{n \bmod L}{2} + 1 & \text{if } n \bmod L \text{ even} \\ \frac{(n \bmod L)+1}{2} Q & \text{if } n \bmod L \text{ odd} \end{cases}$$

In this expression, the first term represents the progress made by fully repeated patterns of R barriers, and the second term represents the additional progress made since the last time the thread groups reversed their roles.

This function is not directly invertible. However we can observe that

$$\lim_{n\to\infty} t_n/n = R/L,$$

since the second term of the solution goes to zero in the limit. This allows us to invert the function in an approximate manner:

$$t_n = \frac{R}{L}n,$$

so:

$$n = \frac{L}{R}t_n$$

and:

$$n = \frac{2\left\lceil \frac{1}{Q-1} \right\rceil - 1}{\left\lceil \frac{1}{Q-1} \right\rceil} t_n.$$

This function describes the behavior seen in figure 2. Here t_n is equal to 500. As Q decreases from 2 to 1, the value of $R = \lceil 1/(Q-1) \rceil$ increases in steps. The frequency of change increases as Q approaches 1, since $\lceil 1/(Q-1) \rceil$ is changing more often. Since R is increasing, $(2R-1)/R$ is increasing also, asymptotically towards the value 2. When Q reaches 1, (the interbarrier computation time exactly equals the quantum length) the term $(2R-1)/R$ is exactly equal to 2, which means that $n = 2t_n$, so the application passes through one barrier every 2 quanta.