Barrier Synchronization on a Loaded SMP using Two-Phase Waiting Algorithms

Dan Tsafrir and Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, Jerusalem, Israel

Abstract

Little work has been done on the performance of barrier synchronization using two-phase blocking, as the common wisdom is that it is useless to spin if the total number of threads in the system exceeds the number of processors. We challenge this and show that it may be beneficial to spin-wait even if the number of threads is up to double the number of processors. We also characterize the alternating synchronization pattern that applications based on barriers tend to fall into, which is quite different from the patterns typically assumed in theoretical analyses.

1 Introduction

Symmetric multiprocessors (SMPs) are the most common parallel machines on the market [2]. The two main synchronization constructs used by parallel applications on such machines are locks and barriers. Locks are used to protect shared data structures in isolation. Barriers are used to delimit phases of the computation, and ensure that all the data structures from the previous phase are up to date.

The nature of synchronization is that processes may have to wait for each other. This waiting can be done in either of two ways: a process can spin, using the CPU to repeatedly check the synchronization condition, or it can block, incurring the context switch overhead but freeing the CPU for the benefit of other processes. This choice is very important, as synchronization overhead is a major cause of performance degradation; one study of the SPLASH2 benchmarks found that some applications spend half of their time waiting for synchronization [5], while another found that some applications spend a third of their time on context switching if they always choose to block [7].

A promising solution to this dilemma is to use two-phase blocking, in which the process spins for a certain time and then blocks if synchronization is not yet achieved [10]. When waiting for a lock, spinning for a time equal to the context switch overhead is 2-competitive, meaning that it results in an execution that is at most a factor of two from that of an optimal execution in which wait times are known in advance [6]. This is the best possible result for a deterministic algorithm, but a randomized algorithm can achieve a competitive factor of $\frac{e}{e-1} \approx 1.58$. The advantage of two-phase blocking has also been demonstrated experimentally [7].

Different synchronization mechanisms, however, have different wait-time distributions. Specifically, assuming Poisson arrivals, the expected wait times at locks are exponentially distributed, whereas the wait times at a barrier are uniformly distributed. Given that the distribution is known, better spin times can be found. Specifically, for locks (exponential distribution) spinning for $\ln(e-1) \approx 0.54$ the context switch overhead leads to a competitive factor of $\frac{e}{e-1}$, and for barriers (uniform distribution) spinning for $\frac{1}{2}\left(\sqrt{5}-1\right) \approx 0.62$ that overhead results in a competitive ratio of $\frac{1}{2}\left(\sqrt{5}+1\right) \approx 1.62$ [9]. However, when the number of processes exceeds the number of processors, spinning was concluded not to be useful for barriers, and immediate blocking was preferred.

Interactions of the synchronization algorithm with the operating system kernel have also been proposed. For example, this can be used to ensure that a process holding a lock will not be preempted [8]. In the case of barriers, the main contribution of this work was the observation that if the number of processes exceeds the number of processors p, only the last p processes should spin. Thus the spinning time is selected dynamically by each process as it reaches the barrier, rather than being a pre-defined constant.

An important factor that is lacking in previous work is taking a global view of the system when it is overloaded. For example, the claim that waiting for the duration of a context switch before blocking is 2-competitive is based on a local view of synchronization, taking one operation at a time. But in a real system, the decision to spin or block may affect the evolution of the computation, and especially the waiting time at subsequent synchronization events. Consider a simple example of two identical jobs with two processes each, on a two processor system. If initially one process of each job is running, the locally optimal algorithm will always block. But a globally optimal algorithm will only block the process of one job, causing the system to move to a state in which it always scheduled both processes of the same job, rather than always scheduling one process from each job. Thus instead of paying the price of a context switch for each synchronization, it becomes essentially free.

Moreover, it turns out that assumptions such as Poisson arrivals to a barrier do not necessarily hold in practice. Our simulations show that applications using barrier synchronization tend to fall into an "alternating synchronization" pattern, in which the job's processes are partitioned into two groups that run alternatively. Due to this pattern, it is sometimes beneficial to spin even if the total number of processes in the system exceeds the number of processors. Indeed, by extending the spin duration, it is sometimes possible to nudge the system into gang scheduling all the processes of a certain application, leading to much more efficient synchronization than that achieved by always blocking.

2 Methodology

2.1 The Simulator

Throughout this work we use an event driven SMP simulator. The simulator distinguishes between *synchronizing* jobs, which perform barrier synchronizations, and *non-synchronizing* jobs, which provide a backdrop of load on the individual processors. Synchronizing jobs may vary in many parameters, among which are size (bounded by p), granularity (the amount of computation between barriers), and variability (the standard deviation of the granularity across threads in the job). A fixed-spinning waiting algorithm is used in order to perform barrier-synchronization. A thread can be in one of three states: ready, running, or blocked (waiting for a synchronization). Spinning is, of course, done in the running state.

The simulator is event based. Only one event is allowed per processor on a given time step. Typical values used in the simulation are a quantum of 100 steps, and a context switch overhead of 6 steps. Checking larger values for the quantum, in order to improve the resolution, showed that the results were practically identical. It turns out that the context switch overhead is the only important parameter, as it distinguishes between "fine grain" and "coarse grain".

Each transition between the various thread states is associated with an event. In addition, events are used to denote the end of a computation phase in synchronizing threads, and for the implementation of spinning. The contention due to synchronization was not simulated. This is a reasonable simplification when assuming that a barrier completion time (with contention) is still significantly shorter than a context switch duration.

Each simulation starts by reading a configuration file which describes the various SMP parameters (e.g. how many processors it has, the length of a context switch, etc.) and the parameters of the jobs it executes (e.g. granularity and variability, number of barriers, number of threads etc.). The output of the simulator is a table describing how well the synchronization policy performed.

2.2 The SSR Metric

Throughout this work, we will use the *successful-spin-rate* (SSR) in order to evaluate the advisability of spinning. This metric is defined to be the percentage of cases in which a process succeeds to synchronize while spinning, excluding the last one to arrive. More formally it is

$$SSR = \frac{\sum_{t \in ST} successfulSpin(t)}{\sum_{t \in ST} totalSpin(t)} \times 100$$

where ST is the set of all synchronizing threads in the simulation, $totalSpin(t)$ is the number of times thread t started to spin when waiting for synchronization, and $successfulSpin(t)$ is the number of times synchronization was achieved before t blocked. Note that this does not include the times t was the last thread of its job to reach a barrier, since no spinning was performed. As a rule of thumb, if the SSR is smaller than 50%, we will consider spinning as not worth while, because threads failed more than succeeded.

We remark that SSR is not a perfect metric and should be used carefully. For example, if the always-spin waiting algorithm is used, jobs executing on a preemptive scheduler (which is what we use in this work) will always achieve an SSR of 100%. Thus we also use elapsed time and speedup in parts of this work.

2.3 The Linux Scheduler

The performance of synchronizing jobs also depends on the scheduler, which chooses the order in which ready threads are allocated to processors. Our simulator includes a rather detailed emulation of the Linux scheduler. Linux is POSIX compliant, and supports three policies: FIFO, Round-Robin (RR), and "OTHER" (this is not defined by POSIX, but its presence is mandated, and it is the default). The latter is a priority-based preemptive scheduler, albeit somewhat different from that of other Unix variants. Additional details are given in Appendix A.

As the priority function tends to give higher priority to threads that run less, it is expected to have a strong effect on synchronizing threads that spend much of their time blocked waiting for synchronization. However, it is easier to understand the behavior of the system under RR scheduling. We therefore performed extensive simulations of all six combinations of the two scheduling schemes (RR and OTHER) and three workloads:

- A single synchronizing job against a backdrop of non-synchronizing threads that just get in the way.
- A homogeneous set of identical synchronizing jobs.
- A heterogeneous mixture of synchronizing jobs with different sizes and different granularities.

Analyzing the first workloads was instrumental in gaining insights that helped understand the latter, more realistic workload.

As tens of thousands of simulation runs were performed, the following sections only present the main findings. A much more detailed description can be found in [12].

3 Alternating Synchronization

The phenomenon of alternating synchronization is the main result discovered in the RR simulations. It explains the finding that many of the simulations led to similar performance, insensitive of the synchronizing job size and (to some degree) of the system load.

For example, simulations of homogeneous sets of synchronizing jobs converged to an SSR in the range of 36-42% when the granularity was 1% of the quantum, and in the range 25-39% when the granularity was 10% of the quantum. The exact number depended on the job sizes, but not on how many jobs were competing with each other! Thus it is clear that a model in which threads are scheduled randomly does not capture the actual system behavior.

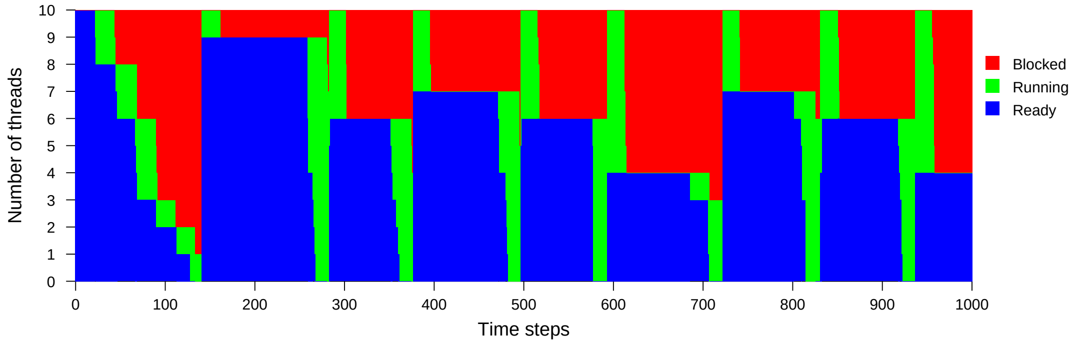

Figure 1: *Distribution of states of threads in a job showing how it enters into an alternating synchronization pattern. From a simulation of 19 homogeneous jobs of 10 threads each, with grain of 10% of the quantum.*

The answer to this puzzle is that each job's threads become partitioned into two sets, that are either running simultaneously of contiguous in the ready queue. This pattern is created by itself, within a short time, even if initially the order of threads in the ready queue is randomized.

The reason that this pattern is created is as follows (Figure 1). Consider a fine-grained job (i.e. with granularity much smaller than a quantum) composed of s threads, running on a p-processor SMP with a total load of n threads. Assuming all threads are randomly ordered at the beginning, the probability of all s threads being allocated processors at the outset is $\prod_{i=0}^{s-1} \frac{p-i}{n-i}$, which tends to zero for high loads. Thus only a subset of the job's threads will run initially. As they are fine-grained, they will complete their first iteration, spin for a while, and block. This scenario will be repeated several times, as more threads get scheduled, leading to the stair-like pattern at the left of the figure. But when the last thread arrives, the job is partitioned into two: those threads that have previously blocked move in unison to the end of the ready queue, whereas those that are spinning achieve synchronization and continue for the next iteration. When this second subset reaches the next barrier, they will all spin and block together, because the first subset is still in the ready queue. Hence we find a pattern of alternating synchronization, where the synchronization is achieved by two subsets of threads alternatively. Note, however, that the subsets are not fixed, and that threads may pass from one to the other if they are not all scheduled at close proximity.

The consequences of alternating synchronization

We've seen that jobs with relatively small granularity, i.e. jobs that perform "a lot" of synchronization, have a tendency to fall into an alternating synchronization pattern. For this type of computation the SSR has a 50% upper bound. This is true because the best we can expect from a thread is to successfully spin at the first barrier it reaches in the quantum (causing the blocked threads in its job to move to the end of the ready queue) and fail spinning at the next (causing it to be preempted and enter blocked state). For every successful spin a thread performs, it also performs an unsuccessful one. Moreover, if the barrier completes as a thread is in the process of being blocked, it may not even contribute a singe successful spin. Therefore the overall SSR is always less than 50%.

An immediate result of this scenario is dismal CPU utilization. As spinning succeeds not more than half the time, it fails more than half the time. That means that at least one in every two barriers includes the cost of unsuccessful spinning and subsequent blocking. Assuming that the spinning time is set to be equal to the blocking overhead (a context switch), this means that each barrier costs a context switch. If the granularity is very fine, i.e. the computation is shorter than this overhead, the effective CPU utilization is less than 50%.

Another interesting feature of the simulations used to study the alternating synchronization

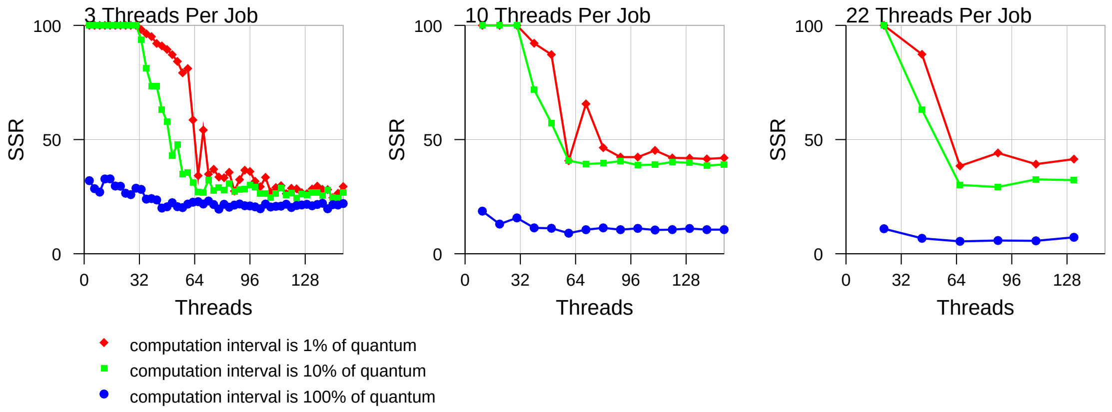

Figure 2: *Example SSR values achieved for homogeneous sets of synchronizing jobs with various sizes, as a function of the number of jobs. Reasonable SSR values are achieved up to twice the number of processors, which is 32. From simulation using RR scheduling.*

pattern is that this pattern does *not* occur immediately as the load is increased. Rather, there is as "intermediate load" range where the system is already full (more than 32 threads), but the SSR is still higher than 50% (Figure 2). This contradicts the argument made above that the SSR is bounded by 50%. This phenomenon, to be further discussed below, indicates that the simple solution of using an "always block" algorithm to reduce the overhead of useless spinning may not be advisable, at least in this load range.

4 Spin Duration and the Transition Point

The simulations used to elucidate the alternating synchronization pattern were based on an RR scheduler. This simplified matters because threads retain their order in the ready queue. But production systems typically use a priority-based scheduler, in which threads are entered into the ready queue according to their priority. Priority, in turn, is typically based on CPU usage (or lack thereof), implying that fine-grain threads may be placed higher than coarse-grain threads. The results presented from here on are based on using the Linux scheduler, as described in Appendix A.

4.1 Conditions for a Transition Point

The first simulations done with the Linux scheduler, in which a single synchronizing job ran against a backdrop of non-synchronizing threads, revealed an interesting pattern: for fine-grain synchronization the SSR was around 50% most of the time, but whenever the total number of threads in the system was a multiple of 16, it shot up to near 100%.

A detailed analysis of what happens at these loads revealed the following. Initially, the synchronizing job did not do very well. Its threads spent much of their time blocked, and typically did not manage to pass more than a single barrier. However, at some point in the simulation, everything suddenly fell into place: all the job's threads were scheduled at the same time, and they therefore completed multiple barriers in rapid succession. This caused them to accumulate CPU time, and their priority dropped. They were then all preempted more or less together in favor of other non-synchronizing threads, and moved to the ready queue. This pattern of interspersed intervals of work and waiting in the ready queue repeated until the end of the simulation (figure 3). We call the point in the simulation at which the job started to work efficiently the *transition point.*

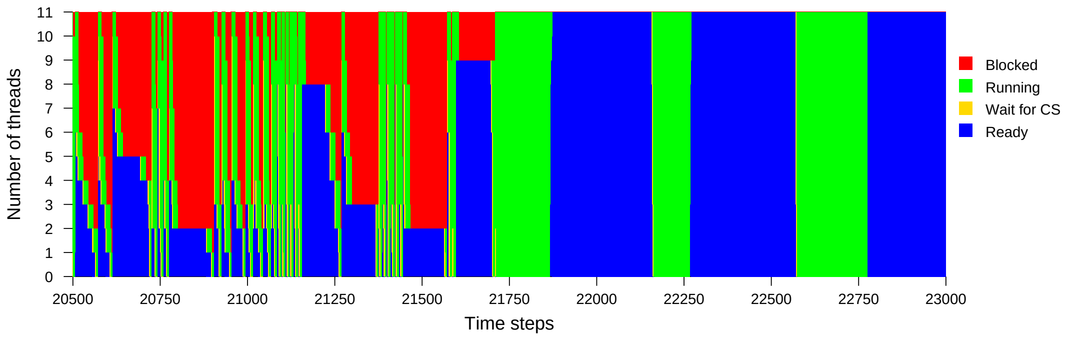

Figure 3: *The distribution of the states of a job's threads changes dramatically after the transition point, when they suddenly manage to run together and then maintain this pattern. From a simulation of an 11-thread synchronizing job (with spinning for the duration of a context switch) and 85 additional non-synchronizing threads, for a total of 96 threads.*

The characteristics of the Linux scheduler are apparent in the job's behavior before the transition point. Contrary to what might be thought, it is not exactly alternate-synchronizing. Rather, its threads are divided into many little subsets that take turns running on a small set of processors. This happens because of two things. First, all the job's threads have a similar priority, because they have a similar history. Second, they are part of the same job, so they enjoy the `SAME_ADDRESS_SPACE_BONUS` when scheduling decisions are made (see Appendix A.). Thus, when one of them blocks, there is high probability that another thread from the same job will be chosen to replace it. If other (non-synchronizing) processes are also preempted, the processes of the synchronizing job may get the freed processors as well. This happens especially towards the end of an epoch, when the non-synchronizing threads have a very low counter, but the threads in the synchronizing job have a relatively high counter, due to the time they have spent in the blocked mode. This relatively high priority allows them to overcome the `PROC_CHANGE_PENALTY` factor. They may even get into an alternate synchronizing pattern. However, the blocked intervals in this pattern are short, because when these processors are unblocked, they tend to have a higher priority than other running processes, so they cause the other processes to be preempted and run in their place.

The problem is that if this happens towards the end of an epoch, the scheduler soon finds itself in a situation in which it has no ready process with a positive counter value. It then starts a new epoch, and re-initializes the counters of all threads. The non-synchronizing threads thus suddenly gain in priority (like Popeye after eating a can of spinach [11]), and soon cannot be preempted any more by the synchronizing threads — until towards the end of the new epoch.

Another problem is that spinning for the duration of a context switch is actually not enough. Consider a process that is the last to arrive at a barrier, and unblocks its peers. It then computes for time t, reaches the next barrier, and spins for time CS before giving up. Its peers, in the mean time, take CS time to start running (this is the context switch that allocates them a processor), and then also compute for time t until they reach the next barrier. Thus they reach the next barrier exactly as the original thread gives up and decides to block.

If everything was completely synchronized, this argument implies that the job would never get all its threads running together. But in the simulator, events that occur in the same simulation time step are serialized by CPU number. Thus it sometimes (though rarely) happens that the sets of processors used by the two alternating subsets of synchronizing threads are non-overlapping. When this happens, the events may be ordered in the correct order: first all newcomers reach the barrier, and then all spinners decide whether to block. As the events happened in the correct order, they decide not to block, and from then on all the treads are synchronized — transition has been achieved.

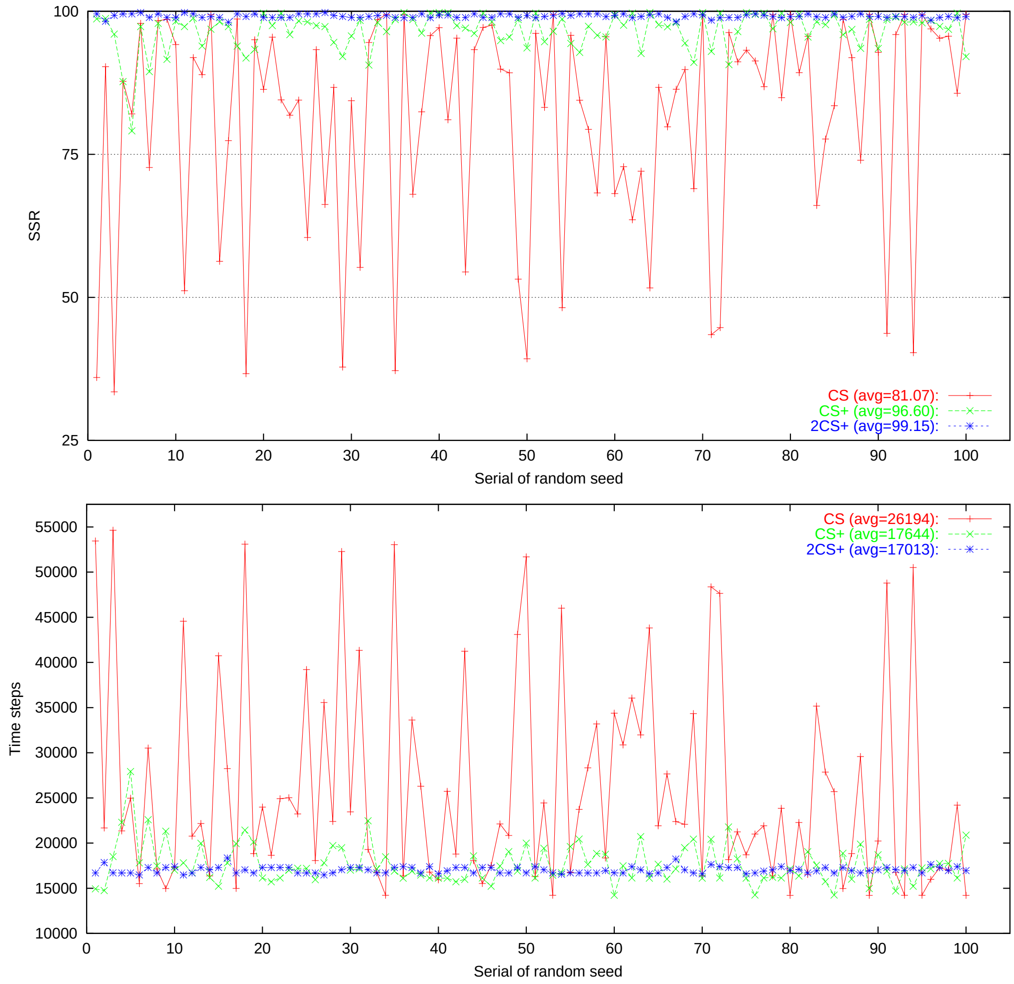

Figure 4: *Comparing the performance of spinning for CS, CS+, or 2CS+ time at a barrier. The latter provides the best combination of good performance and stability. Results from 100 independent simulations using a total of 80 threads, of which 11 are synchronizing at a fine granularity (1% of the quantum).*

Finally, we need to explain why this only happens when the total number of threads is a multiple of 16. The reason is that, for an 11-thread synchronizing job, this allows the threads in the system to be divided into groups that cleanly partition the system. For other numbers, there are always extra non-synchronizing threads that are left over and break the pattern for the synchronizing ones.

4.2 Using a Longer Maximal Spin Duration

Naturally, if transition was only an artifact of the simulation, and only for specific loads at that, it would not be interesting. But in reality it illuminates the condition needed to achieve complete synchronization regardless of the simulator or load: increase the spin-waiting duration beyond a context switch overhead!

Simulations using a spinning duration of slightly more than a context switch overhead (denoted CS+ in the following figures) show an improvement, but not much of an improvement. The reason was traced to the fact that even this is not enough. Consider a scenario in which a thread reaches a barrier and unblocks one of its peers, but that peer thread had only just recently decided to

wakeup scheme	fine grain			medium grain		
	CS	CS+	2CS+	CS	CS+	2CS+
Linux	$38.5_{\pm 8.9}$	$69.9_{\pm 10.4}$	$91.9_{\pm 4.5}$	$23.0_{\pm 5.8}$	$34.7_{\pm 11.5}$	$73.8_{\pm 9.5}$
Improved	$48.9_{\pm 10.3}$	$79.0_{\pm 8.1}$	$93.9_{\pm 3.0}$	$24.7_{\pm 6.8}$	$37.5_{\pm 12.5}$	$75.2_{\pm 9.2}$
Optimal	$54.0_{\pm 10.6}$	$83.1_{\pm 7.0}$	$94.2_{\pm 2.6}$	$28.1_{\pm 7.8}$	$41.6_{\pm 13.5}$	$76.7_{\pm 8.8}$

Table 1: *SSR achieved by a single synchronizing job using various features, given as an average ± the average absolute deviation. CS is a spin duration equivalent to a context switch.*

block. In this situation, the peer thread is still in the process of being blocked, and can therefore not start the unblocking process yet. Thus our thread must first wait for it to block, and then to unblock, for a total time that is more than *twice* the context switch overhead (denoted 2CS+).

The results of using different spin times are compared in Figure 4. While these specific results are for a load of 80 threads (a multiple of 16), similar results were obtained for other loads. In particular, when using 2CS+ spinning, fine-grain jobs almost always achieve an SSR of more than 90% (Table 1). For medium grain jobs (with a grain of 10% of the quantum) using 2CS+ makes the difference between preferring immediate blocking to preferring spinning. Thus enlarging the maximal spin duration has transformed all load conditions into the peak conditions seen initially.

4.3 Effect of the Wakeup Scheme

When a thread is unblocked and has a high priority, it can preempt another running thread. Choosing the processor to preempt is done by setting its `need_resched` flag. The way in which the victim processor is chosen is called the wakeup scheme. Our analysis of the Linux scheduler uncovered a "mis-feature" in this context: when several threads are unblocked at roughly the same time, they may all choose the same victim (see Appendix A. for details). We compared this scheme with a corrected scheme that avoids this pitfall, and with a more sophisticated scheme that takes a global view of pairing threads with processors.

The results were that in most cases the influence of the wakeup scheme on performance were relatively small. There are only few pathological cases in which the Linux mis-feature causes trouble. In these cases, our simple fix is sufficient, and using more sophisticated algorithms is unwarranted.

5 Performance Results for Different Job Collections

After reviewing the specific findings in the previous sections, we now turn to how they interact and affect the performance of various job mixes. The results reported here are for the priority-based Linux scheduler.

5.1 Synthetic Job Mixes

As noted in Section 2.3, we used three types of job mixes: a single synchronizing job with a backdrop of non-synchronizing threads, a homogeneous mix of synchronizing jobs, and a heterogeneous mix of synchronizing jobs.

The results for a single synchronizing job were strongly dependent on the spin duration, and to some degree also on the wakeup scheme (Table 1). The important thing to notice is that when the spin duration is long enough, i.e. a bit more than twice the context switch overhead, an SSR of 94% is achieved for fine-grain jobs, and 75% for medium-grain jobs. These numbers are an average over different load conditions, when the 11-thread synchronizing job competes with up to 200 non-synchronizing threads! Thus we see that the scheduler gives this job's threads a higher priority than the others, which allows them to make good progress regardless of the competing load.

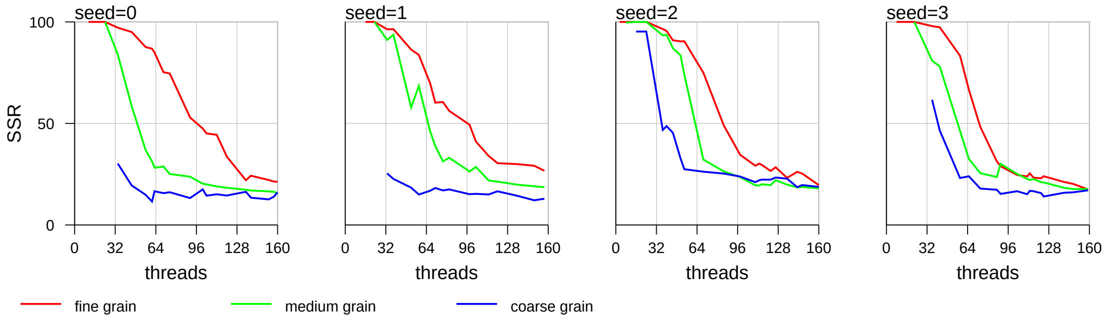

Figure 5: The SSR achieved by different job types in a heterogeneous mix, using 2CS+ spinning. From simulation based on the Linux scheduler. Results for four different random seeds are shown.

wakeup scheme	fine grain			medium grain		
	CS	CS+	2CS+	CS	CS+	2CS+
Linux	23.4 ± 3.7	30.1 ± 5.1	52.2 ± 13.9	10.1 ± 4.2	14.2 ± 5.5	23.1 ± 6.6
Improved	24.8 ± 4.3	30.5 ± 4.7	52.2 ± 19.1	9.9 ± 5.8	13.0 ± 6.0	24.0 ± 5.6
Optimal	24.7 ± 4.0	30.5 ± 4.8	48.5 ± 13.4	9.7 ± 5.2	12.4 ± 4.9	24.9 ± 6.3

Table 2: *Extra threads tolerated while still achieving an SSR above 50%, from simulations of a heterogeneous job mix under a priority-based scheduler. The average is shown together with the average absolute deviation. The system size is 32.*

The case of homogeneous jobs is exactly the opposite: all the competition is composed of threads with an identical profile in terms of synchronization activity. Therefore none will have a distinct advantage over the others, and the scheduler will fall into a pattern similar to that of RR scheduling. Indeed, simulation results turn out to be quite similar to those shown in Figure 2. Again, reasonable SSR values (above 50%) are achieved in the intermediate load range, when the total number of threads is up to twice the number of processors in the system.

Based on this, we would expect that for a heterogeneous job mix the priority scheduler will again come into play. Given jobs with different granularities, the fine-grain ones may be expected to suffer more from blocking, as each synchronization event is amortized by less useful computation. The threads in these jobs will then be given a higher priority, which should enable them to make up for the loss. They will not get above 90% SSR as when running alone, but they should do better than when all competing threads are identical.

The simulation results indeed corroborate these expectations (Figure 5). For example, the system load that can be tolerated while still maintaining an SSR above 50% for fine-grain jobs sometimes reaches *three* times the number of processors in the system. This comes at the expense of coarse-grain jobs, that only manage to synchronize well under lower loads. The extra number of threads tolerated under different conditions is summarized in Table 2.

5.2 Spin vs. Always-Block

So far, most of our work has been concerned with understanding the behavior of spinning jobs, and with finding conditions under which the SSR is improved. In this section we compare our best spinning algorithms with the obvious alternative of always blocking, which has been suggested by others. We also use the actual completion time as a metric rather than the SSR.

Sample results are shown in Figure 6. The y axis here is the relative reduction in completion time when using two-phase spinning as opposed to immediate blocking. Thus the initial value of around 60% means a 60% cut in the completion time, which is equivalent to a speedup by a factor of 2.5; negative values indicate a slowdown. Note that spinning for the duration of a context

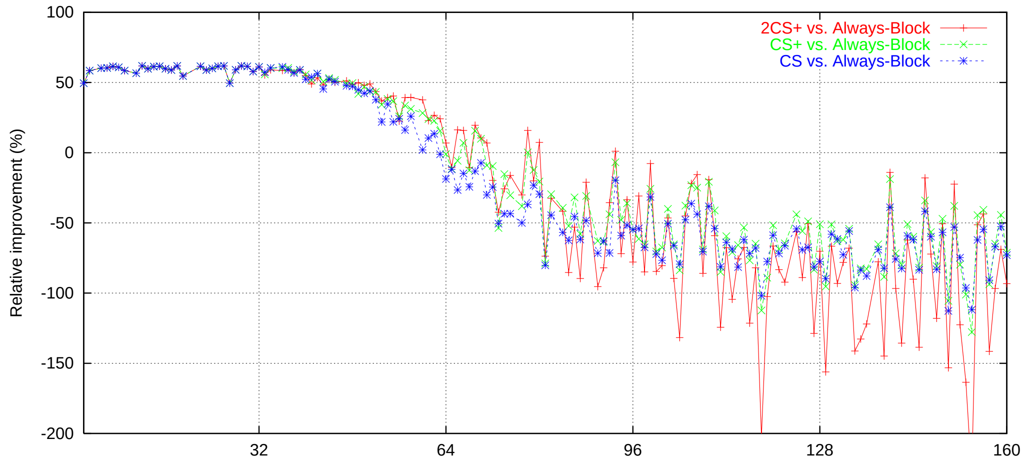

Figure 6: *Spinning cuts around 60% of the runtime relative to always blocking, for all spin durations, up to a load of about twice the number of processors. Results for fine-grain jobs from simulations of heterogeneous job mix.*

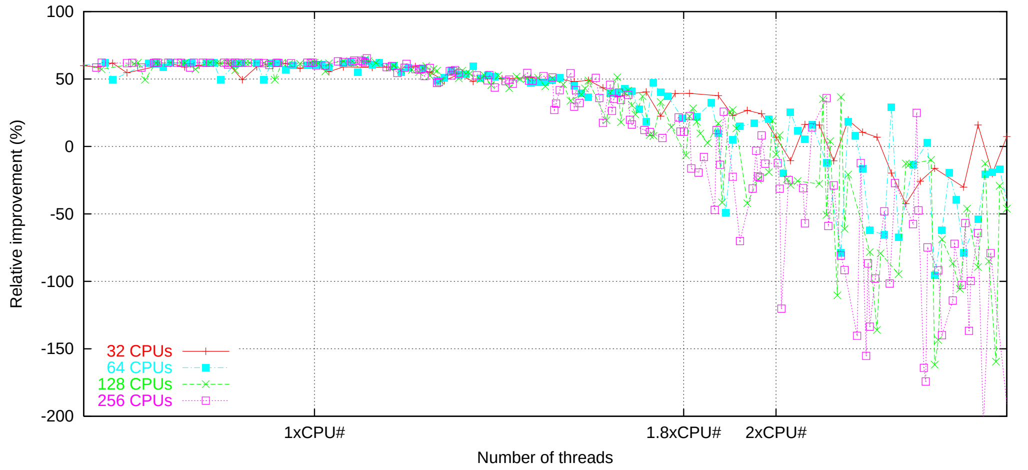

Figure 7: *Larger machine sizes slightly reduce the load threshold beyond which spinning is no longer useful.*

switch (CS) produces similar results to the longer spinning duration (2CS+), and both provide a benefit up to loads of twice the number of processors. This is due to the preferential treatment of the priority scheduler.

5.3 Effect of Machine Size

The final point we will discuss is what happens when we increase the machine's size. Figure 7 displays the results comparing between the completion time achieved by spinning for 2CS+ and always-block (as was done in figure 6) for different sizes. The x axis shows the number of threads relative to the machine size, rather than absolute numbers. Evidently, as we enlarge the machine, the intermediate range in which it is preferable to spin shrinks a bit. Nevertheless, for larger machines in the magnitude of 128 and 256 processors, it is clear that while the load is smaller than 1.8 times the number of processor spinning will still achieve better performance then blocking.

6 Discussion and Conclusions

Our goals in this research were to gain a better understanding of parallel barrier-based applications operating in a multitasking environment, and check the implications of high loads on such applications. We hope these understandings will serve in the design and implementation of barrier synchronization algorithms.

A main contribution of this work is identifying that in the context of barrier synchronization, **load should be a dominant factor** in the decision of whether to spin or block. Most of our empirical results have shown that when the total number of threads in the system exceeds twice the number of processors, most spins will fail and therefore are best avoided. We have shown that any fixed spin algorithm is inferior to the always block algorithm for loads higher than twice the number of processors. On the other hand, in the *intermediate load* range, namely when the surplus in threads is smaller than the number of processors, spinning can be highly beneficial.

Another requirement for successful spinning is doing it for the right time. We have shown that the very popular fixed duration of **spinning for the overhead of a context switch is not enough** for a fine grain parallel job attempting to complete a barrier. Indeed, this duration gives an awakened thread enough time to resume its execution. But, it denies the possibility to actually complete the short computation phase and reach (in time) the synchronization point at which its peer threads are waiting (while spinning). Our findings indicate that a longer duration, of spinning for somewhat more than *twice* the context switch overhead, is required. This duration maximizes the probability that all the threads of a job execute simultaneously, leading to reduced context switches, and to actual spin times that are much smaller than the maximum. This is similar to the result of Arpaci-Dusseau et al. [1] who have shown that in a cluster of workstations spinning for a duration five times the context switch overhead is optimal.

Another important contribution of this work is the identification of the **alternating synchronization pattern**: When jobs do not manage to synchronize, they tend to fall into this computation pattern, in which the job's threads form two contiguous groups in the ready queue. The first group reaches the barrier, spins, and blocks. When the second group runs, the barrier is completed, and all the processes in the first group are released again into the ready queue. Almost all our findings are related to and can be explained based on this phenomenon. This refutes the common assumption that the occurrence of synchronization events obeys some time invariant canonical probability distribution (e.g. the Poisson arrivals of [9]).

The importance of alternate synchronization is evident when considering the effect of granularity on spin success. All the **positive results regarding spinning are for fine-grain**, or maybe medium-grain jobs; we would like coarse grain jobs not to spin. But in general the granularity of a job is not known in advance, and a bad decision may cause a loaded system to waste many cycles on hopeless spinning. It is therefore reasonable to consider some sort of granularity classification mechanism. But due to the prevalence of the alternate synchronization pattern, using the near past as an indication for the future (as in the variable-competitive-algorithms presented in [7]) is not a good option: before the transition point failures are common, and the transition cannot be anticipated based on previous successes. A possible alternative to these methods is for the barrier mechanism to maintain (for each thread) a direct measure of the elapsed time between its few recent synchronization trials (within the same quantum!). This can be done relatively efficiently using hardware devices such as the cycle counter on Pentium processors [4].

An important observation deriving from all the above is that **barriers are quite different from locks**. In the context of (mutex) lock synchronization, Karlin et al. [7] have considered spinning as worth while only when the lock is currently held by a running thread. But in barriers, when a thread reaches a synchronization point, its very own arrival probably means that the awaited threads (in the consecutive synchronization point) are now being scheduled to run. The alternating synchronization computation pattern implies that the practical meaning of following the policy suggested by Karlin et al. (in barrier context) would be to always block. This is contrary to our findings that within the intermediate load, always block is inferior to the fixed spinning policy.

Finally, our work on implementation of barriers also exposed an issue related to the underlying scheduler. When the last thread of a parallel job completes a barrier, many other threads become unblocked at once. The scheduler then checks whether they can be scheduled to run at once. It turns out that the Linux scheduler tries to find the "best" processor for each such thread, and may end up assigning all of them to the same processor! We have shown that a simple improvement, which prevents the scheduler from stumbling over its own feet (by simply noting which processors have already been assigned), produces better results at practically the same cost; more sophisticated approaches seem unwarranted.

Acknowledgement

This research was supported in part by the Israel Science Foundation (grant no. 219/99).

A. The Linux Scheduler

While Linux supports FIFO and Round-Robin scheduling, the default scheduler is priority based. We remark that in the Linux kernel, thread and process entities are indistinguishable; the conventional term used to represent them both is a *task*. The scheduler described here has been in effect essentially unchanged since version 2.2 of the kernel.

Unlike other versions of Unix, Linux scheduling is based on the notion of an *epoch*. In a single epoch, every task has a certain CPU time allocation, which was set at the beginning of the epoch. The default allocation is equal for all tasks, unless they have different nice values. When a task exhausts its allocation it is preempted in favor of another runnable task. However, the task can block and then continue to run if its allocation has not been exhausted. An epoch ends when all the ready-to-run tasks have exhausted their allocations, but blocked tasks may still have part of their allocation. To start a new epoch, all tasks receive new allocations. This is computed as the default allocation plus half of what was left of the previous allocation. Thus the maximal possible allocation is twice the default allocation.

Within an epoch, runnable tasks are selected for execution based on their priority. The priority has a dynamic part, which is simply the remaining time allocation. This is measured in "ticks", typically 0.01 of a second (10 milliseconds). The default allocation was 20 ticks in Linux 2.2, and was changed to 5 in 2.4[1]. The dynamic priority is also called the "counter value", as it is stored in a variable called the counter, and essentially counts down the CPU usage of the task in this epoch; when it reaches 0 the task will not run any more.

The actual scheduling algorithm is not based directly on a task's priority, but on its *goodness* relative to different processors. The goodness is based on the counter value; if this is zero the goodness is also zero. But for tasks that have not exhausted their allocation, two modifications are made. First, if the considered processor is different from the one on which the task last ran, the goodness is reduced by the `PROC_CHANGE_PENALTY`, which is equivalent to 15 ticks[2]. Second, if the previous task to run on this processor had the same address space as this task, the goodness is improved by the `SAME_ADDRESS_SPACE_BONUS`, equivalent to 1 tick.

In the context of our work, it is important to understand what happens when tasks become unblocked (as when a barrier synchronization is achieved). Such tasks are moved to the ready queue, and the `reschedule_idle` function is called for each one of them in turn. This function tries to find a suitable processor for the awakened task, giving priority to the one it ran on previously or to the longest idle processor. But if these two options do not work out, the goodness of the new task is compared with the goodness of the current task on all the processors. The processor with the largest difference is then chosen, provided the difference is larger than the `PREEMPTION_THRESHOLD` (1 tick). The selected processor (if any) is then marked as `need_resched`, which means that the scheduler will be called when the system returns to user mode, or at the next clock tick.

[1]This means that the scheduler has rather poor resolution when it tries to distinguish between different jobs. We used the 2.2 value which is slightly better. A still better solution would be to reduce the tick interval [3].

[2]Making a migration from one processor to another practically impossible in 2.4. This is another reason to use the 2.2 values.

Note, however, that the `reschedule_idle` function does not actually map tasks to processors — it just verifies that a suitable candidate exists, and marks it. Thus when many tasks are awakened at once, it is possible that they all find the mark the same processor, e.g. the longest idle one. Such a situation may end up leaving high-priority tasks in the ready queue despite the fact that they could have preempted other tasks on other processors. In our work we therefore also considered other wakeup schemes, that avoid this pitfall.

Finally, a somewhat surprising find is that the overhead of a context switch in Linux is linear in the number of tasks in the ready queue. The actual numbers are typically between 140 and 400 cycles per task (with the bulk at less than 180). For a 550 MHz machine with 32 processors and say a hundred ready tasks, the overhead can therefore be 25–73 μs. This is a result of the fact that the goodness of all tasks has to be calculated anew each time, because it is different when different processors are being targeted. It is surprising because better data structures could be used to make this constant, based on the rather limited number of possible priority levels. In our simulations we used a rather high constant overhead value of 6% of the quantum duration. This is a reasonable upper bound for the overhead in a real system, and in fact the specific value is not very important — only that spinning is done for the desired multiple of this value.

References

[1] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring, "*Scheduling with implicit information in distributed systems*". In *SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 233–243, June 1998.

[2] D. E. Culler and J. P. Singh, *Parallel Computer Architecture*. Morgan Kaufmann Publishers Inc., second ed., 1999.

[3] Y. Etsion, *How Should the UNIX Scheduler Handle Modern Interactive Processes*. Master's thesis, School of Computer Science and Engineering, the Hebrew University of Jerusalem, 2002. In preparation.

[4] Y. Etsion and D. G. Feitelson, *Time Stamp Counters Library Measurements with Nano Seconds Resolution*. Technical Report, School of Computer Science and Engineering, the Hebrew University of Jerusalem, July 2001. http://www.cs.huji.ac.il/labs/parallel/tsclib.ps.

[5] D. Jiang and J. P. Singh, "*Scaling application performance on a cache-coherent multiprocessor*". In *Proc. 26th Ann. Int'l Symp. Computer Architecture*, pp. 305–316, May 1999.

[6] A. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, "*Competitive randomized algorithms for non-uniform problems*". In *Proc. 1st ann. ACM-SIAM symp. Discrete Algorithms*, pp. 301–309, January 1990.

[7] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki, "*Empirical studies of competitive spinning for a shared-memory multiprocessor*". In *Proc. 13th ACM Symp. Operating Systems Principles*, pp. 41–45, October 1991.

[8] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott, "*Scheduler-conscious synchronization*". *ACM Trans. Computer Systems* **15(1)**, pp. 3–40, February 1997.

[9] B-H. Lim and A. Agarwel, "*Waiting algorithms for synchronization in large-scale multiprocessors*". *ACM Trans. Computer Systems* **11(3)**, pp. 253–294, August 1993.

[10] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In *Proc. 3rd Int'l Conf. Distributed Computing Systems*, pp. 22–30, October 1982.

[11] E. Segar, *Thimble Theatre, Popeye the Sailor Man*. King Features Syndicate, 1929.

[12] D. Tsafrir, *Barrier Synchronization on a Loaded SMP using Two-Phase Waiting Algorithms*. Master's thesis, School of Computer Science and Engineering, The Hebrew University, Sep 2001.