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, 91904 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, especially if the waiting time is at least twice the context switch overhead (rather than being equal to it). 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 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 (denoted *CS*) 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 [9]. When waiting for a lock, spinning for a time equal to CS 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 $\ln(e-1) \approx 0.54$ of CS 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 $\frac{1}{2}\left(\sqrt{5}+1\right) \approx 1.62$ [8]. 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.

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 CS 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 which is typically 32), and granularity (explained below). A fixed-spinning waiting algorithm is used to perform barrier-synchronization (unless stated otherwise, the maximal spin duration used is CS). A thread can be in one of three states: ready, running, or blocked (waiting for a synchronization). Spinning is, of course, done in running state.

Typical values used in the simulations are a quantum of 100 steps, and a context switch overhead (CS) of 6 steps. The latter is probably too long. However, aside from being considerably shorter than a quantum duration, its only importance lies in the manner in which we classify granularity of jobs. Checking larger values for the quantum, in order to improve the resolution, showed practically identical results.

All computation intervals of threads are normally distributed (i.e. they are not deterministic). Granularity is expressed based on the mean and standard deviation of this distribution. Let X_J denote a random-variable representing the duration of computation intervals between consecutive barriers of threads from job J. We classify J as being *fine-grained* if around 90% of X_J's values are smaller than CS. J is categorized as *medium-grained* if around 90% of X_J's values are smaller than 5CS. Otherwise, J is considered to be *coarse-grained*.

The simulator is event based. Only one event is allowed per processor on a given time step. 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 shorter than CS.

Each simulation starts by reading a configuration file which describes the various SMP parameters (p, CS etc.) and the parameters of the jobs it executes (e.g. granularity, sizes, number of barriers, etc.). The simulator's output describes how well the synchronization policy performed.

2.2 The SSR Metric

In order to evaluate the advisability of spinning, we will use the *successful-spin-rate* (SSR). 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: $SSR = \frac{\sum_{t \in S} successfulSpin(t)}{\sum_{t \in S} totalSpin(t)} \times 100$ where S 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'll 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". OTHER is not defined by POSIX, but its presence is mandated, and it is the default. In Linux it is a priority-based preemptive scheduler. 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 [11].

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 jobs size and (to some degree) of the system load. For example, simulations of homogeneous sets of fine and medium grain jobs converged to an SSR in the range of 25-42% (Figure 1). The exact number depended on the job sizes and granularity, but not on how many jobs were competing with each other!

The answer to this puzzle is that each job's threads become partitioned into two sets, that are either running simultaneously or 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 this pattern is created is as follows (Figure 2): Consider a fine-grain job 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 Figure 2. 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 , 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 (causing the blocked threads in its job to move to the ready queue) and fail spinning at the next barrier (thus entering blocked state). We get that for every successful spin a thread performs, it also performs an unsuccessful one.

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 CS, 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%.

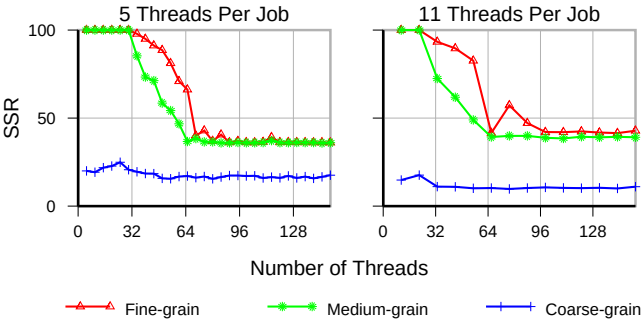

Figure 1. *Example SSR values achieved for homogeneous sets of synchronizing jobs with various sizes, as a function of the number of threads. From simulation using RR scheduling. Reasonable SSR values are achieved up to twice the number of processors, which is 32. However after this point, SSR is below 50% and is load invariant.*

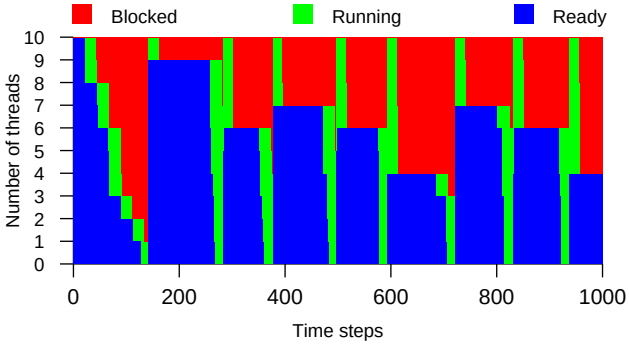

Figure 2. *Distribution of states of 10-thread-job showing how it enters into an alternating synchronization pattern. From a homogeneous simulation of 19 medium grain jobs.*

Another interesting feature of the simulations used to study the alternating synchronization 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 1). 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 Wakeup Scheme

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 Transition Point

The first simulations done with the Linux scheduler, in which a single synchronizing job composed of 11 threads 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% (Figure 3). The maximal spin duration used in these simulations was CS.

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 4). We call the point in the simulation at which the job started to work efficiently the *transition point*.

The characteristics of the Linux scheduler are apparent in the job's behavior before the transition point. Let J denote the synchronizing job. Let S denote the native processors set of J's threads. Initially, S is usually small. J's computation pattern is a variation of alternating synchronization on S: Since J's threads enjoy the `SAME_ADDRESS_SPACE_BONUS` (see Appendix A.), then when one of them blocks, there is high probability that another thread from J will immediately be chosen to replace it. Towards the end of the epoch the priorities of the non synchronizing threads are very low (by definition) while the priorities of their synchronizing counterparts are relatively high (since they spent a lot of time in blocked mode). This difference allows J's threads to overcome the `PROC_CHANGE_PENALTY` factor and preempt low priority threads even when migration is involved. Consequently S grows until $|J| = |S|$ causing J to perform *rapid alternating synchronization* where every thread from an unblocked group is immediately assigned a processor. The scheduler soon finds itself in a situation in which it has no ready process with a positive counter value. It then starts

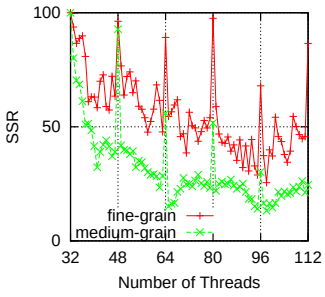

Figure 3. *SSR achieved by an 11-thread synchronizing job running against a backdrop of non synchronizing threads and spinning for the duration of CS. SSR is very high whenever the load is a multiple of 16.*

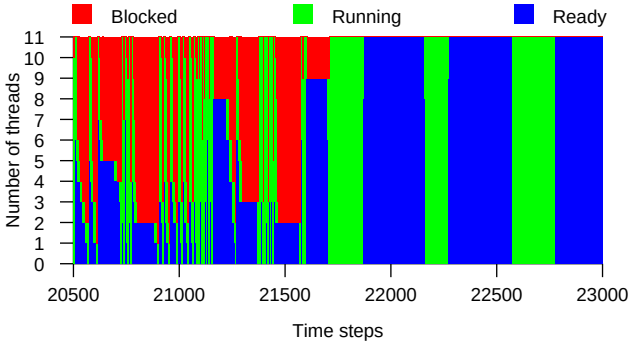

Figure 4. *The distribution of the states of a job's threads changes dramatically after the transition point, when they suddenly manage to run together and then maintain this pattern. From a simulation presented in Figure 3 associated with load of 96 threads (backdrop of 85 non sync threads).*

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 [10]), and soon cannot be preempted any more by J's threads — until towards the end of the new epoch.

The immediate question that follows is why does J continue to perform alternating synchronization even when $|J| = |S|$ as each of its threads has its own "dedicated" processor. The answer is simple: spinning for the duration of CS is actually not enough. Consider a process that is the last to arrive at barrier b_i, and unblocks its peers. It then computes for an expected time of μ, reaches the next barrier b_{i+1}, and spins for time CS before giving up. Its blocked peers, in the mean time, take CS time to start running (context switch that allocates them a processor), and then also compute for an expected time of μ until they reach b_{i+1}. They therefore reach b_{i+1} more or less at the same time the original thread gives up and decides to block.

However, on rare occasions it happens that the two alternating sets reach b_{i+1} in the correct order: first all newcomers reach the barrier, and then all spinners decide whether to block. As all newcommers have already arrived, they decide not to block, and from then on all the treads are synchronized — transition has been achieved.

Finally, we need to explain why this only happens when the total number of threads is a multiple of 16. The reason is that 16 is the only divisor of 32 (=system size) which is big-

ger than $|J|$ (=11). 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 and so even if transition is achieved, it lasts only during the epoch in which it was established.

4.2 Using a Longer Maximal Spin Duration

Naturally, if transition occurred only within specific loads, it wouldn't be interesting. But in reality it illuminates the condition needed to achieve complete synchronization regardless of the 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+*) show an improvement, but not an optimal 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 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+*).

Figure 5 shows that enlarging the maximal spin duration has indeed transformed all load conditions into the peak conditions seen initially. Note that for medium grain jobs, using 2CS+ makes the difference between preferring immediate blocking to preferring spinning.

4.3 Effect of the Wakeup Scheme

When a job completes a barrier, the priority based scheduler checks whether consequently awakened threads (if exist) can be immediately scheduled to execute (possibly by preempting lower priority threads). It is therefore faced with the problem of determining which awakened thread would be assigned to which processor. The algorithm that makes this decision is called the *wakeup-scheme*. The question that follows is how much computational resources should a scheduler invest in this decision. Our analysis of the Linux scheduler uncovered that unfortunately, it doesn't invest enough: the scheduler iterates through the awakened threads and tries to find the "best" processor for each such thread; however each iteration has no recollection of previous iterations' decisions and therefore two or more (even all) awakened threads may be assigned to the same processor! (see Appendix A. for details). We compared this scheme with a corrected scheme that avoids this pitfall (denoted *AP*), and with a more sophisticated (probably impractical) scheme that takes a global view of pairing threads with processors [11, chapter 6] (denoted *GV*).

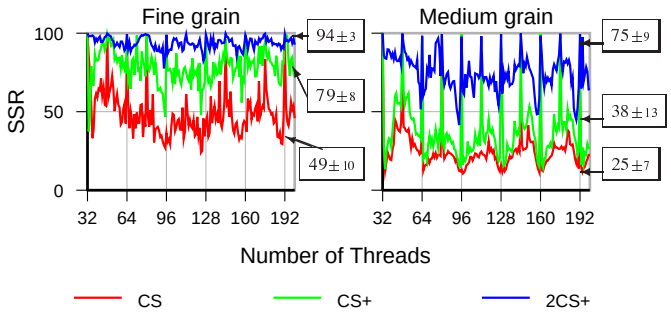

Figure 5. *Results from simulations similar to those presented in Figure 3. Comparing the performance of spinning for CS, CS+, 2CS+. The latter provides the best combination of good performance and stability. Numbers in boxes are the average ± the average absolute deviation.*

Our findings indicate that for some job mixes (like a single synchronizing job running against a backdrop of CPU-bound threads, or mixes composed from two jobs with size bigger than $\frac{p}{2}$), AP resulted in a speedup of up to 3.3 in comparison to the original wakeup scheme. We found that using a more sophisticated algorithm is unwarranted as the difference between AP and GV was minor.

5 Performance of 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 3 types of job mixes: The results for **a single synchronizing job** were strongly dependent on the spin duration (Figure 5), and to some degree also on the wakeup scheme. The important thing to notice is that when the spin duration is long enough (2CS+), an average SSR of 94% is achieved for fine-grain jobs, and 75% for medium-grain jobs. These number are an average of different load conditions, when the 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.

The case of **homogeneous jobs mix** 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 1. Again, reasonable SSR values (above 50%) are achieved in the intermediate load range, when the total number of threads is up to twice the

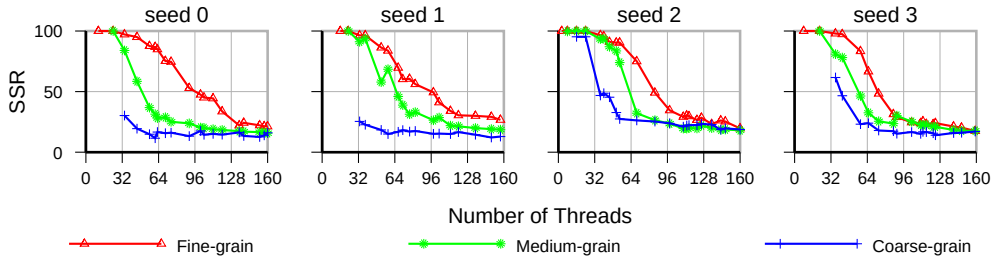

Figure 6. *The SSR achieved by different job types executing simultaneously in heterogeneous mix, using a spin duration of 2CS+. Results for four different random seeds are shown.*

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 would 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 6). 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. The average extra number of threads tolerated under different conditions (like jobs' number, granularity and size) is $52.2_{\pm 19.1}$ and $24_{\pm 5.6}$ for fine and medium grain jobs, respectively.

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 finally compare our spinning algorithms with the obvious alternative of always blocking as suggested by others. We do that by using the actual completion-time as a metric, rather than the SSR.

Sample results shown in Figure 7 and Figure 8 confirm that spinning is preferable to blocking, at least within the intermediate load. Figure 8 shows that lengthening the spin duration beyond CS plays a minor role within job mixes that don't contain non-synchronizing threads (as all spin durations produced similar results). The major factor in these mixes is actually idle processors. These exist due to blocked threads which create a gap between the total load and the effective load (number of runnable threads).

5.3 Effect of Machine Size

The final point we will discuss is what happens when we increase the machine's size (Figure 9). Evidently, 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's clear that spinning will still achieve better performance then blocking while the load is smaller than 1.8 times the number of processors.

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. 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 fine grain parallel jobs 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 their threads form two groups. When one group is computing, the other is either blocked or ready. Almost all our findings are related to and can be explained based on this phenomenon. This refutes

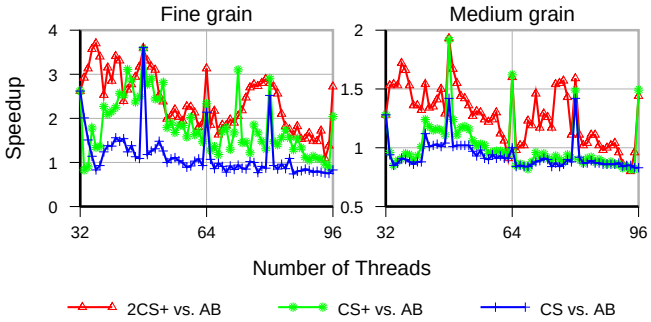

Figure 7. *Comparing the performance of spinning for CS, CS+, or 2CS+ against the always-block policy (denoted AB). 2CS+ provides the best combination of good performance and stability with an average speedup of $2.4_{\pm 0.4}$ for fine-grain jobs and $1.3_{\pm 0.2}$ for medium-grain jobs. From simulations of a single synchronizing job similar to those conducted in section 4.1.*

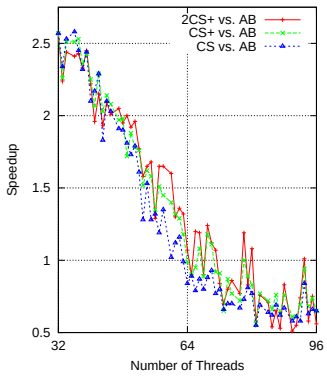

Figure 8. *Results for fine-grain jobs from simulations of a heterogeneous job mix. Spinning is beneficial for loads up to about twice the number of processors. The average speedup within this domain is $1.9_{\pm 0.3}$ when 2CS+ is used as a maximal spin duration.*

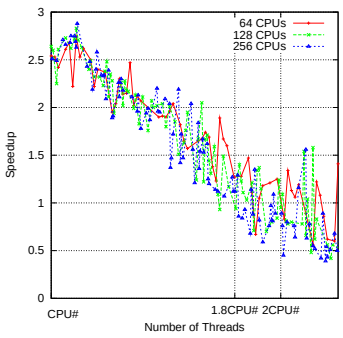

Figure 9. *Results for mixes similar to those described in Figure 8 using 2CS+ as maximal spin duration and executing on larger systems. The x axis shows the number of threads relative to the machine size, rather than absolute numbers.*

the common assumption that the occurrence of synchronization events obeys some time invariant canonical probability distribution (e.g. the Poisson arrivals of [8]).

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 sometimes 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 of a fine-grain job 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 while the Linux (2.4) scheduler tries to find the "best" processor for each such thread, it may end up assigning all of them to the same processor! Our experiments show that a simple improvement, which prevents the scheduler from stumbling over its own feet (by simply remembering 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 is of Linux-2.4.5 (essentially unchanged since version 2.2).

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 initial

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 yet been exhausted. An epoch ends when all the ready-to-run tasks have exhausted their allocations (though blocked and running 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 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 be preempted.

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 (i.e. from the same job), the goodness is improved by 1 tick which *we* named `SAME_ADDRESS_SPACE_BONUS`.

In the context of our work, it is important to understand what happens when tasks become unblocked (as when a barrier is completed). 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 (if it's idle) or to the longest idle processor. If there are no idle processors, the goodness of the awakened 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` and interrupted (which means that very soon the scheduler will run in its context).

Unfortunately, `reschedule_idle` is invoked in a serial manner independently for each awakened task, and each invocation disregard previous invocations' decisions. Thus when many tasks are awakened at once, it is possible that some (or all) of then will trigger the marking of the same processor. Consequently, this wakeup scheme can (a) end up leaving high-priority tasks in the ready queue despite the fact that they could have preempted other tasks on other processors, or (b) even worse: leave processors idle, even when there exist (newly awakened) ready to run tasks!

The following is a simple example that demonstrates this. Let c be the longest idle processor. Changing c's state from idle to non-idle takes time, leading to a race between this event and the `reschedule_idle` iteration. If the iteration finishes before c's state was changed, only c will be marked. In our work we therefore also considered alternative wakeup schemes, that avoid this pitfall.

[1] This means 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 non idle processor practically impossible in 2.4. This is another reason to use the 2.2 values.

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*, pages 233–243, June 1998.

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

[3] Y. Etsion and D. G. Feitelson. Clock Resolution and the Scheduling of Interactive Processes. Technical report 2001-14, School of Computer Science and Engineering, the Hebrew University of Jerusalem, Nov 2001.

[4] Y. Etsion and D. G. Feitelson. Time stamp counters library measurements with nano seconds resolution. Technical report 2000-36, School of Computer Science and Engineering, the Hebrew University of Jerusalem, Aug 2000. 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*, pages 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*, pages 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*, pages 41–45, October 1991.

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

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

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

[11] 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.