Impact of Load Imbalance on the Design of Software Barriers*

Alexandre E. Eichenberger

Advanced Computer Architecture Laboratory
EECS Department, University of Michigan
Ann Arbor, MI 48109-2122
alexe@eecs.umich.edu

Santosh G. Abraham

Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303
abraham@hpl.hp.com

Abstract

Software barriers have been designed and evaluated for barrier synchronization in large-scale shared-memory multiprocessors, under the assumption that all processors reach the synchronization point simultaneously. When relaxing this assumption, we demonstrate that the optimum degree of combining trees is not four as previously thought but increases from four to as much as 128 in a 4K system as the load imbalance increases. The optimum degree calculated using our analytic model yields a performance that is within 7% of the optimum obtained by exhaustive simulation with a range of degrees. We also investigate a dynamic placement barrier where slow processors migrate toward the root of the software combining tree. We show that through dynamic placement the synchronization delay can be reduced by a factor close to the depth of the tree, when sufficient slack is available. By choosing a suitable tree degree and using dynamic placement, software barriers that are scalable to large numbers of processors can be constructed. We demonstrate the applicability of our results by performing measurements on a small SOR relaxation program running on a 56-processor KSR1.

KEYWORDS: Synchronization barrier, fuzzy barriers, combining tree, shared-memory multiprocessors, parallel processing

1 Introduction

Synchronization barrier constructs are used by programmers to ensure that all processors have reached a particular point in a computation before any processors are allowed to advance beyond that point. Parallel supercomputer applications typically use data-parallel programming techniques where large data structures are updated in parallel by all the processors. Barriers are used in such programs to separate the phases of the computation and to ensure that all processors have finished updating a data structure in step t before any processor uses the updated values as input in step $t+1$. The simplest implementation of a barrier uses a counter protected by a lock. The overhead of such barriers increases linearly with the number of processors and can dominate overall execution time.

In response to the potential overhead of software barrier synchronization schemes, several hardware schemes have been proposed. The NYU Ultracomputer [1] and the IBM RP3 [2] employ combining networks which combine accesses to the same memory location, thus alleviating contention on the counters used to implement synchronization barriers. Other machines such as the Sequent, SGI, and Alliant have provided special synchronization buses. Vector supercomputers such as the Cray and the Convex provide a set of communication registers which are used for fast barrier synchronization. The Cray T3D multiprocessor has a fast synchronization network for barrier synchronization [3]. The hardware support for efficient synchronization is indicative of the potential impact of synchronization on overall performance.

Software barrier synchronization schemes that can approach the performance of hardware techniques are extremely attractive because hardware schemes have several disadvantages. Hardware synchronization schemes are expensive; for instance, the combining network is at least six times as expensive as a non-combining network [4]. Also, hardware schemes employ special-purpose logic that has a large design cost, especially significant for low-volume parallel systems. Thirdly, software barriers are more flexible and can be adapted to suit the application or to exploit advances in synchronization techniques such as fuzzy barriers. Finally, software barrier schemes are more easily portable to different platforms.

The disadvantages of hardware synchronization schemes have motivated the study of software barriers using software combining trees. In some cases, these studies have demonstrated that the synchronization performance of software schemes approach that of hardware synchronization [5] [6]. In a software combining tree, a tree of counters is used for synchronization. Processors are divided into groups and a group is assigned to each leaf of the combining tree. Each processor updates the counter and if it finds it is the last one to reach the counter it proceeds to the parent of the counter. The last processor to reach the counter at the root of the tree releases all the processors by updating a shared variable.

In previous work, the performance of synchronization barriers is evaluated and optimized for the case where all processors arrive at the barrier simultaneously. This assumption is motivated by the perception that synchronization mechanisms and load imbalance are two different issues that can be solved independently. As a result, synchronization mechanisms are developed assuming the case of zero load imbalance and consequently maximal contention.

However, experiments on parallel machines have demonstrated that processors typically fail to reach a synchronization

*Appeared in the *Proceedings of the 1995 International Conference on Parallel Processing*, pp 63–72, August 14-18, 1995.

point at the same time for several reasons. Firstly, the workload may be unevenly partitioned among the processors. As a result, certain processors consistently arrive late at a synchronization point thus idling other processors and resulting in *systemic load imbalance*. Though, systemic imbalance can be handled by effectively partitioning the workload, sufficient information may not be available to perform the best partitioning. In *non-deterministic* imbalance, processors fail to reach a synchronization point at the same time but typically the processor arriving last changes on each iteration. Non-deterministic load imbalance is generated by several factors: the workload associated with a processor may change from cycle to cycle; the interprocessor communication may incur random delays due to contention; or there may be contention for hardware or software resources.

Theoretically, the earliest a processor can leave a barrier is when the last processor arrives at the barrier. In practical implementations such as combining trees, processors have to wait till this last processor updates the root counter. Thus, we define *synchronization delay* of combining trees as the difference between the release time (time when last processor updates root counter) and the arrival time of the last processor. There are two components to this delay: *update delay* of updating counters and *contention delay* for locks that govern access to the counters. The first component is determined by the tree depth and the second by the number of processors simultaneously attempting to gain access to a particular counter.

The first contribution of this paper is to investigate the degree of a software combining tree that minimizes the synchronization delay as a function of the load imbalance. On the one hand, if all processors arrive simultaneously, contention delay is maximum and a deep tree is desirable. On the other hand, if one of the processors arrives significantly later than the others, the main issue is to reduce the update delay and a wide tree is desirable. Thus the requirements for the two cases, viz. deep tree (small degree) and wide tree (large degree), are conflicting and the best degree of the combining tree is a function of the amount of load imbalance. In this paper, we develop an approximate analytic model for estimating synchronization delay as a function of load imbalance and combining tree degree. Simulation results show that the performance of the combining tree obtained using the analytic model is within 7% of the performance of the optimum combining tree obtained through exhaustive simulation, for normally distributed processor execution times.

The second contribution of this paper is a novel software barrier where late arriving processors migrate toward the root of the combining tree to minimize the synchronization delay. In the normal software combining tree, late arriving processors tend to update more counters on average. In presence of systemic load imbalance, some processors tend to be consistently late and are consistently delayed further by the synchronization barrier. We explore the possibility of giving less synchronization work to these processors by modifying a tree structure first proposed in Mellor-Crummey and Scott [7]. In their tree structure, one processor is statically attached to each non-leaf counter in the software combining tree. The rest of the processors are split into groups and assigned to the leaf counters. A processor updates the counter it is attached to as well as its parent if it is the last processor to arrive at the counter. In contrast to the static assignment in Mellor-Crummey and Scott, we attach processors dynamically to the non-leaf counters. Processors that tend to arrive late are attached to counters closer to the root to reduce the latency delay of synchronization. We compare the performance of the dynamic placement scheme with the static placement scheme used by Mellor-Crummey and Scott and obtain significant performance improvement.

This paper is organized as follows. First, we present a summary of the related work in Section 2. We present an analytical model for approximating the synchronization delays in Section 3. Using this model, we estimate the optimal combining tree degree of a barrier in Section 4. We investigate a dynamic placement barrier in Section 5. In Section 6, we present a quantitative comparison of the simulation results of the two previous sections. Section 7 presents our measurements on a parallel machine (KSR1). Finally, we conclude in Section 8.

2 Related Work

Performance degradations due to busy wait synchronization are widely regarded as a serious performance problem. Pfister and Norton [4] showed that the presence of hot spots can severely degrade performance for all traffic in multistage interconnection networks. Agarwal and Cherian [8] investigated the impact of synchronization on overall program performance and showed that cache line invalidations due to synchronization references can account for more than half of all invalidations.

In response to performance concerns, hardware support has been designed and implemented in several parallel machines. Combining networks [1] [2], that combine concurrent accesses to the same memory location, have been advocated as a technique that significantly reduces the impact of busy waiting. Similarly, special purpose cache protocols [9] [10] have been designed to include synchronization primitives that reduce communication due to synchronization.

New software synchronization mechanisms have been developed to approach the synchronization performance of dedicated hardware at lower cost. Yew, Tzeng and Lawrie [6] investigated the use of software combining trees to distribute hot spots in large scale multiprocessors. Their analysis indicates that combining trees effectively decrease memory contention. Furthermore, they showed that the optimal degree of a combining tree (fan in) is around four. Mellor-Crummey and Scott [7] refined this technique further, presenting an algorithm that generates the theoretical minimum number of communications on machines without broadcast. Michael and Scott [5] showed that a software implemented exclusion mechanism could outperform naive hardware locks, even under heavy contention.

Alternatives to the usual synchronization barriers have also been investigated. Gupta [11] developed and investigated Fuzzy Barriers. He measured significant performance improvements with software implemented Fuzzy Barriers on a four processor Encore Multimax. He presents techniques [11] [12] that detect and increase the number of independent operations, and hence the slack time. Eichenberger and Abraham [13] characterized the performance improvements due to fuzzy barriers and showed that the expected idle time at a fuzzy barrier is inversely proportional to the slack time. Finally, Nguyen [14] investigated compiler techniques that transform synchronization barriers into point

to point synchronizations, showing encouraging performance improvements.

The source and extent of variation of thread (processor) execution times have been investigated in a few studies. Adve and Vernon [15] have measured the fluctuations of parallel execution times for a large number of applications and observed that the empirical execution time distribution very closely tracks the normal distribution. Dubois and Briggs [16] obtained an analytical formula describing the expected number of cycles and its variance for memory references in tightly coupled systems. Sarkar [17] provided a framework to estimate the execution time and its variance based on the program's internal structure and control dependence graph. Finally, Eichenberger and Abraham [13] analyzed the fluctuation of processor execution time due to random replacement caches and communication contention. We derived an analytical formula describing the expected variance for programs with simple memory and communication access patterns. In all four papers, the variation in execution times was found to approximate a normal distribution and we assume that thread execution times are normally distributed in this paper.

The effects of load imbalance on idle times, assuming a perfect barrier with zero synchronization delay, have been investigated in several articles. Kruskal and Weiss [18] have investigated the total execution time required to complete k tasks for various distributions. The performance of parallel algorithms that have regular control structures and non-deterministic task execution times is quantified by Madala and Sinclair [19]. Durand *et al* provide experimental measurements on the impact of memory contention in NUMA parallel machines [20].

Axelrod [21] has considered both the effects of load imbalance and synchronization costs and derived an analytical result that takes both load imbalance and synchronization costs into consideration. However, while considering the synchronization costs, he assumed that processors arrive simultaneously at the synchronization point thus overestimating the effects of contention. We determine synchronization delays as a function of both the particular synchronization structure used and the load imbalance.

Beckmann and Polychronopoulos [22] have investigated the effects of barrier synchronization and dynamic loop dispatch overhead. They classify loops as synchronization bound or arrival-time bound, depending on the spread of processor arrival time at the barrier. They derived an analytical result for these two cases for shared-bus multiprocessors and present theoretical and simulated speedup curves.

Definitions

In this article, we distinguish two synchronization phases: the release and the enforce phase. During the *release* phase, a processor signals its arrival at the synchronization point by incrementing counters in a combining, or synchronization, tree. A synchronization tree consists of L levels of counters, where each counter is connected to at most d other counters, where d is the tree degree. During the *enforce* phase, a processor checks if all processors have completed their release phase.

We furthermore define the *arrival* time as the time at which a processor arrives at the release phase and the *release* time as the time at which a processor completes the release phase. The time needed by a processor to update one counter is defined as t_c. This time includes the communication time to fetch the counter and to execute an atomic operation.

3 Analytic Model for Estimating Synchronization Delays

In this section, we will first derive the synchronization delay assuming simultaneous processor arrivals and then estimate this delay for general processor distributions.

When assuming simultaneous processor arrivals, the resulting synchronization delay for a combining tree of degree d with L full levels is obtained as follows. At the lowest level of the combining tree, d processors will simultaneously attempt to increment their counter. Since only one processor can update its counter at a time, the d processors are serialized. The last processor completes incrementing of its counter after $d \cdot t_c$. Since all processors arrive simultaneously, the same process occurs at each level of the combining tree. Therefore, the total synchronization delay for a combining tree with L full levels is defined as follows:

$$T_{sync,\,0}(L) = L\; d\; t_c \tag{1}$$

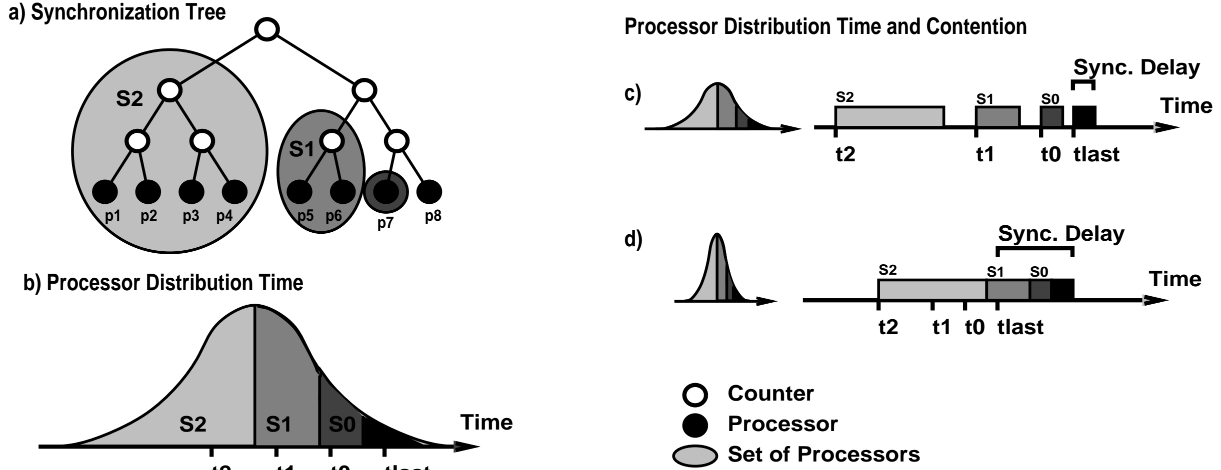
Given that the number of levels in a combining tree for p processors is defined as $L = \log_d p$ (p chosen such that it results in full levels), Equation (1) yields a minimum synchronization delay for a combining tree of degree $d = e \simeq 2.71$.

Two problems arise when extending the previous model to processors that do not arrive simultaneously. The first problem is that only the slowest processors propagate upward in the tree, requiring the use of order statistics [23] at each level of the tree. The second problem is that contention at one level changes the distribution of the processors that propagate to the next level in the combining tree. As a result, a direct solution of the synchronization delay in the presence of load imbalance would require expensive numerical computations. Therefore, we introduce three assumptions and find an approximate analytical solution that takes the load imbalance into account.

First, we partition the processors into subsets and assume that all processors of a subset arrive simultaneously. Second, we assume a specific ordering of the arrival time of each subset of processors: the closer a subset is to the last processor, the later it arrives. Third, we relate the arrival and release times of each subset to its respective position in the combining tree.

In Figure 1a, we selected processor `p8` to be the last one. Along its path to the root, it will experience contention with each of the three subsets `S0` through `S2`. The first assumption of our approximation states that all processors in subsets `S0`, `S1`, and `S2` arrive respectively at time `t0`, `t1`, and, `t2`. The ordering of these arrival times is illustrated in Figure 1b. Figure 1c and 1d illustrate how subset arrival times and contention delays are merged together. In Figure 1c, the distribution is wide enough to prevent the last processor from being slowed down by the contention of earlier processors. In Figure 1d, however, the distribution is narrower and contention from previous processors affects the last processor.

We compute the subset arrival time by first defining the processor subsets in a combining tree, then computing the percentage



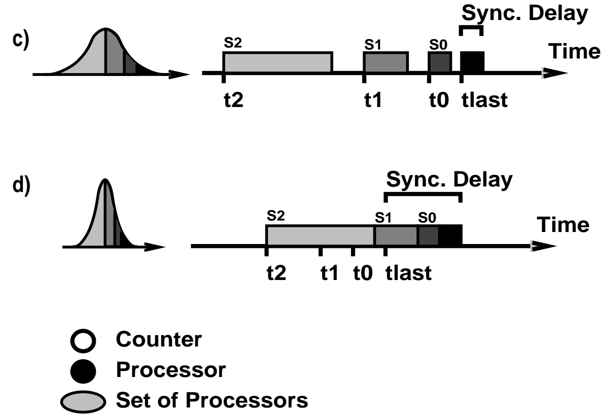


Figure 1: Estimating the synchronization delay.

of processors that arrive earlier, and finally applying the density function of the processor arrival time.

We define a subset S_l as the subset that includes all the subtrees of exactly depth l that are connected to the counters along the path from the last processor to the root of the combining tree. For example, the subset S1 in Figure 1a contains all the subtrees of exactly depth 1, and therefore consists of processors $p5$ and $p6$. In general, we see that subset S_l consists of $d-1$ subtrees of depth l and therefore contains $(d-1)d^l$ processors. Furthermore, we know from our second assumption that all the processors in subsets $S_{l+1} \dots S_{L-1}$ arrived before the ones in subset S_l. Therefore, the expected fraction of processors that arrived before the processors in subset S_l is defined as follows:

$$P_{before}(S_l) = 1 - P_{in\,/\,after}(S_l) = 1 - d^{l-L+1} \quad (2)$$

Now that we have obtained the expected fraction of processors to arrive before the processors of subset S_l, we are able to determine the expected arrival time of each subset:

$$T_{arr}(S_l) = F^{-1}(P_{before}(S_l)) \quad (3)$$

where F^{-1} is the inverse of the distribution function of the processors. If the processors are normally distributed with parameters μ and σ, the expected arrival time of subset S_l is

$$T_{arr}(S_l) = \sigma\,\Phi^{-1}(P_{before}(S_l)) \quad (4)$$

where Φ^{-1} is the inverse of the normal distribution function[1]. Since we consider here only the arrival time relative to the mean, we omitted the μ term in the preceding equation. Finally, we can asymptotically estimate the arrival time of the last processor with the help of order statistics [23]:

$$T_{arr}(last) = \sigma \left(\sqrt{2\log p} - \frac{\log\log p + \log 4\pi}{2\sqrt{2\log p}} \right) \quad (5)$$

[1] Since $P_{before}(S_{L-1}) = 0$ and $\Phi^{-1}(0) = -\infty$ we approximate $P_{before}(S_{L-1})$ as $P_{before}(S_{L-2})/2$.

Furthermore, we compute the release times as follows. The release time (T_{rel}) of a subset S_l corresponds to the sum of its arrival time, contention delays, and propagation time from the subset root counter to the combining tree root counter,

$$T_{rel}(S_l) = T_{arr}(S_l) + T_{sync,\,0}(S_l) + (L-l)t_c \quad (6)$$

The release time of the last processor corresponds to the sum of its arrival time and its propagation time to the root counter through all levels,

$$T_{rel}(last) = T_{arr}(last) + Lt_c \quad (7)$$

Since the last processor experiences contention with the processors of each subset, its release time corresponds to the maximum of all release times. The synchronization delay is defined as the difference between its release and arrival times:

$$T_{sync,\sigma} = \max_{s=S_0 \dots S_{L-1}} (T_{rel}(last),\ T_{rel}(s)) - T_{arr}(last) \quad (8)$$

The steps required to compute Equation (8) are summarized in Algorithm 1. This approximation is useful in estimating the optimal degree of a combining tree, since we can approximate the synchronization delay associated with a given number of processors, processor distribution, and degree of its combining tree.

Algorithm 1 *Given a combining tree with L levels and degree d, the synchronization delay is computed as follows.*

1. *The release time of each subset $S_0 \dots S_{L-1}$, with respective levels $0 \dots L-1$, are computed by using Equations (1), (2), (4) and (6).*
2. *The release time of the last processor is computed by using Equations (5) and (7).*
3. *The synchronization delay is computed by using Equation (8).*

To characterize the accuracy of Equation (8), we compared its results against simulation results. We assumed the processors to be normally distributed and obtained data for various processor numbers, combining tree degrees and standard deviations of

processor distributions. The time to update a counter, t_c, was experimentally measured on a KSR1 and that value[2] was used in our set of simulations.

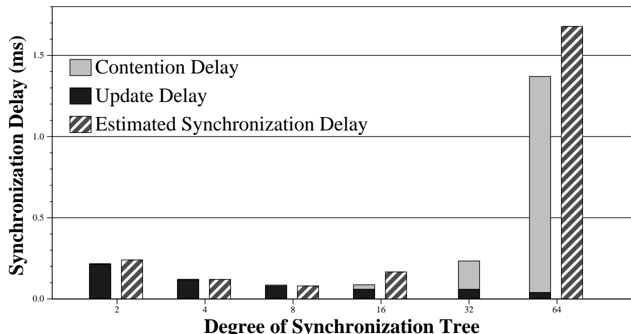


Figure 2: Synchronization delay for various degrees. (4K Processors, $\sigma = 250$ ms, $t_c = 20\mu$s)

Figure 2 illustrates this comparison for 4K processors and for six different degrees of combining trees. Degrees 2, 4, 8, 16, 32, and 64 resulted in tree depths of 12, 6, 4, 3, 3, and 2, respectively. In each pair, the right bar represents the approximated synchronization delay. Since we assumed full trees when deriving Equation (8), there is no data for the degree 32. The left bar represents the simulated results and consists of two components: update delays and contention delays. The update delay is directly proportional to t_c and the tree depth. The contention delay increases dramatically after a threshold tree degree (16 in this figure).

Despite the strong assumptions used to obtain Equation (8), namely the simultaneous arrivals within subsets of processors and the ordering of the subset arrival times, we see that this approximation still captures the behavior of synchronization under workload imbalance.

4 Optimal Degree of Combining Tree

In this section, we investigate the optimal combining tree degree of a synchronization barrier for various numbers of processors and processor distributions. We assumed the processors to be normally distributed, an assumption supported by measurements in [13] and [15]. First, we will determine the optimal degree experimentally. Then, we will use the approximation presented in Section 3 to estimate this degree and compare the performance improvement between the experimental and estimated optimal degree.

Figure 3 presents the optimal combining tree degree for synchronizing 64, 256, and 4K processors for various standard deviations defined in units of t_c. The optimal degree corresponds to the degree that resulted in the smallest synchronization delay, as defined in Section 1. The horizontal axis lists a range of standard deviations for the distribution of processor execution times. The vertical axis lists the number of processors simulated. The first number in each entry is the optimal degree obtained by exhaustive simulation. The second number, in parenthesis, is the synchronization speedup obtained when using a combining tree with optimal degree, expressed as a ratio of the synchronization delay for a combining tree of degree four to that of the optimal degree. The performance of a combining tree of optimal degree is compared to a combining tree of degree four because degree four was previously considered as optimal [6] [7].

Processors	Optimal Degree (Sync. Speedup)		
	$\sigma = 0t_c$	$\sigma = 25t_c$	$\sigma = 500t_c$
64	4 (1.00)	64 (2.87)	64 (2.99)
256	4 (1.00)	32 (1.97)	256 (4.00)
4K	4 (1.00)	32 (1.98)	128 (2.99)

Figure 3: Simulated optimal degree of combining trees.

To obtain these optimal degrees we used a conventional event driven simulator. The time for updating a counter (t_c) was set to 20μs and the contention for updating the counters was accounted for in the simulation. The optimal degree was determined by carrying a simulation for all feasible degrees and choosing the one that yields the smallest synchronization delay.

There are several conclusions that can be drawn from this experiment. First, this experiment confirms the fact that combining trees of degree four are optimal when processors arrive simultaneously, or when the distribution of processors is small compared to t_c. Second, this experiment confirms our assertion that with a wide distribution of processors, large degrees yield smaller synchronization delays. For example, when 64 processors are distributed with a standard deviation of $25t_c$, a single counter yields the smallest synchronization delay. Finally, the synchronization speedup gained by synchronizing processors with a combining tree of optimal degree compared to a combining tree of degree of four ranges from 30 percent faster with a degree of eight up to 300 percent faster with a degree of 256.

One could argue that if there is substantial load imbalance relative to the counter update time t_c, that the overall parallel application is inefficient and any improvements in synchronization performance will have only a small impact on the overall performance. However, when fuzzy barriers are employed, load imbalance does not necessarily translate into idle times [13] and an application could have substantial load imbalance and still be efficient, provided synchronization delays are not excessive.

We now investigate the use of the approximated synchronization delay of Equation (8) to estimate the optimal degree of a combining tree. Figure 4 presents the estimated optimal degree for synchronizing 64, 256, and 4K processors for various standard deviations defined in units of t_c. These estimated optimal degrees are found in the rows labeled “est”. For comparison, the optimal degree obtained by exhaustive simulation are shown in the rows labeled “opt”. Results in bold indicate the cases where the estimated degree differs from the simulated optimum. As in the previous figure, we present the synchronization speedup gained by synchronizing processors with combining tree of optimal degree compared to a degree of four. The difference between the speedup associated with the optimal and the estimated optimal degree is particularly interesting, because it is a metric of how accurate our approximation is.

As indicated in Figure 4, the approximated synchronization delay is useful in determining the optimal degree. Moreover, when the estimation fails to identify the optimal degree, the

[2] On the KSR1, it takes on average 20μs to acquire a subpage in atomic mode and to increment its value.

Processors		Optimal Combining Tree Degree (Synchronization Speedup)											
		$\sigma = 0t_c$		$\sigma = 6.2t_c$		$\sigma = 12.5t_c$		$\sigma = 25t_c$		$\sigma = 50t_c$		$\sigma = 500t_c$	
64	opt.	4	(1.00)	8	(1.31)	**16**	**(1.47)**	**64**	**(2.87)**	64	(2.94)	64	(2.99)
	est.	4	(1.00)	8	(1.31)	**8**	**(1.46)**	**8**	**(1.48)**	64	(2.94)	64	(2.99)
256	opt.	4	(1.00)	**8**	**(1.26)**	**32**	**(1.89)**	**32**	**(1.97)**	**64**	**(1.99)**	256	(4.00)
	est.	4	(1.00)	**4**	**(1.00)**	**16**	**(1.78)**	**16**	**(1.96)**	**16**	**(1.99)**	256	(4.00)
4K	opt.	4	(1.00)	4	(1.00)	8	(1.43)	**32**	**(1.98)**	**128**	**(2.97)**	**128**	**(2.99)**
	est.	4	(1.00)	4	(1.00)	8	(1.43)	**16**	**(1.96)**	**64**	**(2.94)**	**64**	**(2.99)**

Figure 4: Simulated and estimated optimal degree of combining trees.

speedup associated with the estimated optimal degree is usually not significantly smaller than the optimal speedup. Indeed the optimal degree combining trees are only 7% faster on average than the estimated degrees.

We also investigated the combining trees proposed by Mellor-Crummey and Scott [7], as described in Section 1. We simulated these trees and obtained results similar to the one of Figure 4. Comparing these results with the ones of Figure 4, we noticed performance improvements of 5%, on average, for all combining trees with an optimal degree of four. However, this performance improvement vanishes when the optimal degree is larger than four. For degree four, this performance improvement is due to the fact that the average depth seen by the processors is smaller, since some of the processors are attached at higher levels of the combining tree; however, for larger degrees this improvement decreases as the proportion of processors at the higher levels also decreases.

We note that if one could guess which of the processors will arrive late at a synchronization point, one could place those processors near the top of the combining tree, and therefore reduce the synchronization delay. The next section will investigate the feasibility of this technique.

5 Dynamic Placement in Combining Trees

In this section, we use the combining tree presented by Mellor-Crummey and Scott [7]. As mentioned in Section 1, this technique uses combining trees of degree d where each counter in the tree is connected to at least one processor, and where leaf counters are connected statically to at most $d+1$ processors. We will investigate the feasibility of a scheme that positions late processors near the top of the tree, thus reducing the synchronization delay of the slower processors. This technique can be viewed as the shifting of the synchronization cost from the slower to the faster processors. We use a prediction scheme based on recent history that is expected to work well in two situations: with systemic workload imbalance and with fuzzy barriers.

With systemic workload imbalance, the workload is unevenly partitioned among processors, and therefore processors that were attributed a larger amount of work will systematically arrive late at synchronization points. A similar situation arises with evolving workload imbalance, where the workload slowly fluctuates from iteration to iteration. In both cases, recent history is a good indication of future processor arrival order.

With fuzzy barriers [11], independent operations are inserted between the release and the enforce phase, thus significantly reducing the expected idle time due to non-deterministic work-

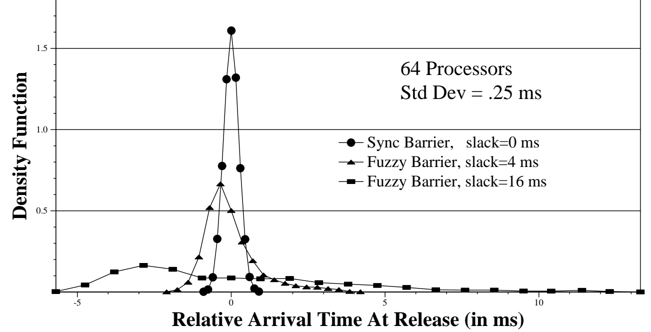


Figure 5: Distribution of processor arrival time for fuzzy barriers. ($\sigma = 0.25$ ms, $p = 64$)

load imbalance. The independent operation execution time corresponds to the *slack* of a fuzzy barrier. In [13], we showed that the expected idle time is inversely proportional to slack.

Here, we will consider another interesting property of fuzzy barriers. With increasing slack, some processors can be significantly slower than others without requiring faster processors to wait for them. As a consequence, processors that arrive late at a synchronization point are likely to arrive late in the near future.

Figure 5 illustrates the processor arrival time distributions for 64 processors and a range of slacks after 500 iterations. In this figure, the arrival times are relative to the mean: negative times correspond to processors faster than the average and vice versa. The curve labeled "Sync Barrier" corresponds exactly to the processor arrival time distribution after one single iteration since all processors were strongly synchronized at the previous synchronization barrier. We see that the larger the slack is, the wider the distribution is. This effect is explained by the fact that with large slack, processors are allowed to desynchronize themselves and are eventually distributed over the entire slack.

Several conclusions can be drawn from this experiment concerning the processor distribution synchronized with fuzzy barriers. First, we see that the processor distributions become wider with increasing slack. Second, we see that the distributions are not symmetric on their mean, resulting in an increasing number of fast processors and a decreasing number of slow processors. Finally, we see that the slowest processors are far away from the mean and are likely to remain far away during the next few iterations. Since the curve labeled "Sync Barrier" indicates the distribution after a single iteration, we know from the graph that it is unlikely that a processor changes its relative position by more than a half millisecond in this graph. As a result, processors that are ten milliseconds or more away from the mean are very likely to

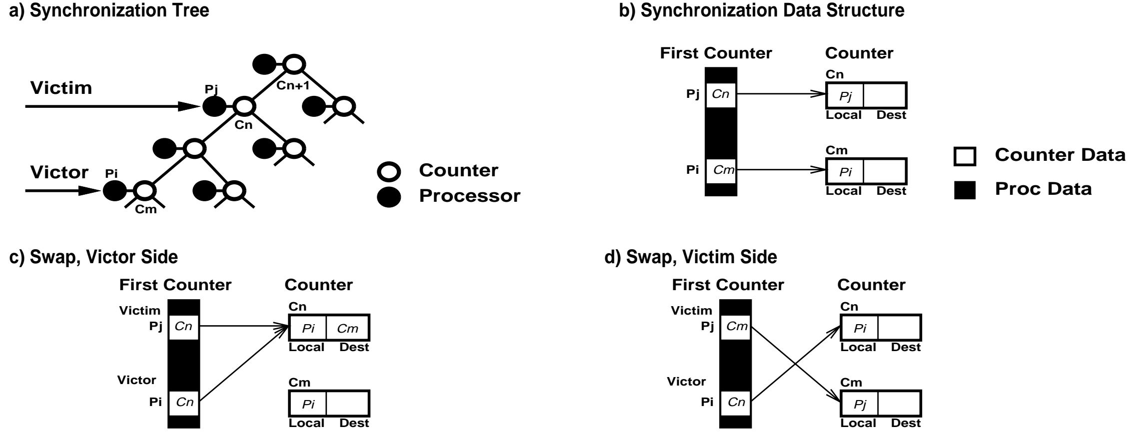


Figure 6: Dynamic placement barrier mechanism.

remain significantly slower for the next 20 iterations. This means, in turn, that a dynamic placement scheme is feasible with fuzzy barriers when the slack is larger than the distribution of processors after one iteration.

5.1 Dynamic Placement Barrier Algorithm

This section presents a *Dynamic Placement Barrier* algorithm that predicts the ordering of the processor arrival time based on previous history and places slower processors near the top of the combining tree. This algorithm is based on the combining tree presented in [7]. However, we expect to reduce the critical path from $O(\log p)$ to $O(1)$ when the prediction is successful.

The dynamic placement barrier proceeds as follows. When a processor propagates toward the top of the tree, it positions itself at the highest level counter where it arrived last. Figure 6a illustrates this scheme: processor `Pi` starts incrementing up the tree from the counter `Cm`, its initial counter, up to the counter `Cn+1`. We can deduce from this fact that processor `Pi` arrived last in the whole subtree attached to the counter `Cn`. Our dynamic placement scheme swaps processor `Pi` with the processor associated with `Cn`, namely `Pj`. For the remainder of this section, we name the processor that is swapped to a higher level in the combining tree the *victor* processor, and the one that is swapped to a lower level the *victim* processor.

This scheme requires two data structures, as illustrated in Figure 6b. The first one, `First-Counter`, is private to each processor and provides a pointer to the first counter associated with each processor. The second data structure, `Counter`, is associated with each counter and consists of two entries, `Local` and `Destination`. Its first entry allows us to locate the processor that is currently attached to a counter and the second entry provides an index to the new initial counter of the victim.

The swapping of a pair of processors occurs in two phases: the first phase occurs on the victor side and is followed by a second phase on the victim side. During the first phase, shown in Figure 6c, the victor processor checks if the conditions are fulfilled for a swap. If a swap is indicated, it updates its `First-Counter` pointer to the counter of the victim and modifies the two entries of the victim's initial `Counter` as follows: it writes its processor ID in the `Local` entry and its old `First-Counter` value in the `Destination` entry. During

```
TYPE Counter = RECORD
       count, init: INTEGER;    (* #children   *)
       local: [0..P-1];         (* current id  *)
       dest, parent: *Counter;
     END;
VAR sense: BOOLEAN;             (* shared var  *)
    first_counter: *Counter;    (* private var *)
    private_sense: BOOLEAN;     (* private var *)

PROCEDURE Release(id: [0..P-1]);
  private_sense := NOT private_sense;
  IF (first_counter->local != id) AND
      NOT Leaf(first_counter) THEN
      (* swap, victim size *)
    first_counter := first_counter->dest;
    first_counter->local := id;
  END;
  curr := first_counter; prev := NULL;
  LOOP (* through combining tree levels *)
    c := FetchAndDecrement(curr->count);
    IF (c != 0) THEN EXIT END;
      (* last at this level: reinit counter *)
    curr->count := curr->init;
    prev := curr; curr := curr->parent;
    if (curr = NULL) THEN EXIT END;
      (* last in last level: exit *)
  END;
  IF not Leaf(prev) AND (prev->local != id) THEN
    prev->local := id;   (* swap, victor size *)
    prev->dest := first_counter;
    first_counter := prev;
  END;
  (* last in last level reverse sense *)
  IF (curr = NULL) THEN sense := NOT sense END;
END

PROCEDURE Enforce()
  (* wait for last processor *)
  WHILE (private_sense != sense) DO END;
END
```

Figure 7: Dynamic placement barrier algorithm.

the next synchronization phase, the victim processor detects that it has been swapped by inspecting its initial counter `Local` field and by noticing that it is no longer local to its initial counter. As illustrated in Figure 6d, the victim uses the `Destination` entry of its initial counter to find out its new initial counter and updates its `First-Counter` entry accordingly. Figure 7 presents the detailed operations of this algorithm.

Assuming that the cache line is large enough to accommodate

a counter, local, and destination field, the communication overhead of the dynamic placement algorithm is one communication per swap, since one additional communication occurs on the victim side to acquire its new initial counter. Fortunately, this overhead occurs on the victim side, which was the faster of the two processors. Since there is at most one such swap among a counter and its direct children, the communication overhead is bounded by $1/(d+1)$ additional communications per processor. As a result, we can limit the upper bound communication overhead of this algorithm by choosing an appropriate degree of the combining tree.

	Slack				
	0ms	1ms	2ms	4ms	16ms
Degree: 4					
Last Proc Depth	5.85	3.34	1.88	1.44	1.24
Sync. Speedup	1.00	1.73	3.07	3.98	4.71
Comm. Overhead	1.09	1.08	1.07	1.04	1.01
Degree: 16					
Last Proc Depth	2.99	2.16	1.59	1.36	1.21
Sync. Speedup	0.99	1.34	1.85	2.21	2.45
Comm. Overhead	1.04	1.03	1.02	1.01	1.00

Figure 8: Performance of the dynamic placement barriers.

Figure 8 presents the performance improvement obtained with the dynamic placement barrier for 4K processors, normally distributed with a standard deviation of 0.25 ms for various slacks. Each set of measurements presents the average depth of the combining tree seen by the last processor releasing the barrier, the synchronization speedup of the dynamic placement scheme relative to the static placement scheme, and the fractional increase in communication occurred in the dynamic placement scheme.

First, we notice that as the slack increases the average depth of the last processor is effectively reduced from 5.85 to 1.24 and from 2.99 to 1.21 for barriers of degree 4 and 16, respectively. Second, we see that the synchronization speedup, relative to the performance of a static placement scheme, increases from 1 to 4.71 and from 0.99 to 2.45 for barriers of tree degree 4 and 16, respectively. It is interesting to notice that dynamic placement does not improve the performance with a slack of zero, in which case a static placement is as relevant a placement as the one of the previous iteration.

6 Quantitative Comparison

The performance improvements due to individual techniques were presented in earlier section. This section attempts to combine these techniques together and presents the cumulative performance effect.

Figure 9 presents the performance improvements due to synchronization with an optimal degree combining tree. First, we see that, for the curves corresponding to combining trees of degree four, the synchronization delay exactly corresponds to the depth of the tree, indicating that there is no contention. Therefore, the smallest standard deviation presented is sufficient to remove all contention problems for combining trees of degree four. Second, we see the performance improvements due to combining trees of optimal degree. Their synchronization delay is consistently less

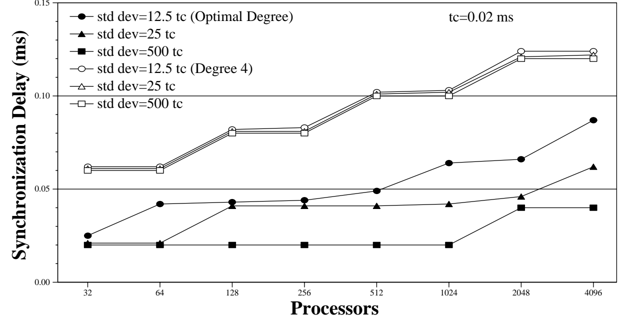


Figure 9: Degree four versus optimal degree.

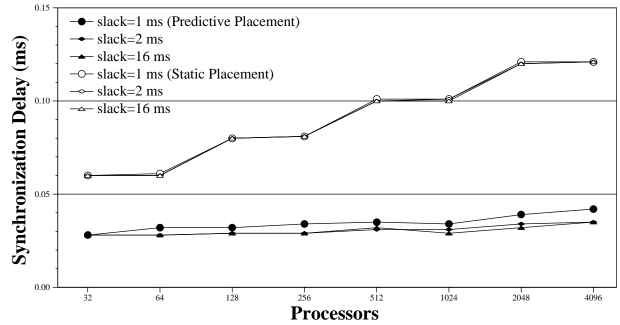


Figure 10: Static versus dynamic placements (degree 4).

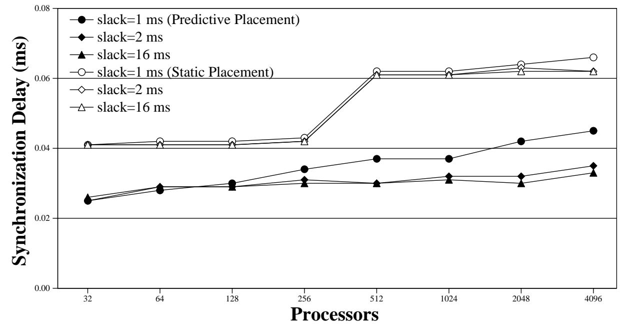


Figure 11: Static versus dynamic placements (degree 16).

than combining trees of degree four. Furthermore, the slope of these curves decreases with increasing standard deviation. Thus, the synchronization delay is relatively insensitive to the system size when load imbalance is sufficiently large.

Figure 10 illustrates the benefits of dynamic placement for execution times that are distributed with a very small standard deviation ($3.1t_c$). First, we see that the curves that correspond to a synchronization degree of four are similar to the curves of the previous figure, again indicating that there is no contention problem. Second, we see the performance improvement due to dynamic placement. The dynamic placement scheme almost neutralizes the tree depth in larger systems, and the synchronization delay is nearly constant.

Finally, Figure 11 illustrates the combined effects of higher tree degrees and dynamic placement. As in the previous figures, the curves that correspond to a static placement with a combining tree degree of 16 result in curves increasing stepwise with the number of processors, again indicating that there is no con-

Optimal Tree Degree (Synchronization Speedup)			
$d_y = 52$ $\sigma = 42\mu$s	$d_y = 210$ $\sigma = 110\mu$s	$d_y = 420$ $\sigma = 220\mu$s	$d_y = 840$ $\sigma = 421\mu$s
4 (1.00)	8 (1.04)	32 (1.22)	32 (1.23)

Figure 12: Measured optimal degree on the KSR1 for 56 processors.

tention problem. The curves associated with the dynamic placement present a synchronization delay that is smaller and increasing at a slower rate. We note that the curves associated with the dynamic placement scheme are more sensitive to the variations in slack than the corresponding curves of Figure 10. This effect is due to the fact that flatter trees have proportionately fewer processors near the top of the tree, making the choice of the slower processors more critical.

7 Measurements

In order to test the performance improvements due to optimal tree degree and dynamic placements on the KSR1 [24], we used a program that has a well defined computation and communication pattern. We used a relaxation algorithm (SOR) where each element is averaged with its four neighbors. The relaxation is performed in two alternating arrays, thus avoiding additional communication due to race conditions among processes. The two-dimensional data of size (d_x, d_y) is partitioned along the x-dimension, resulting in $4\lceil d_y/16\rceil$ communication events per processors[3]. By varying the y-dimension, we change the total number of communications and therefore the variation of execution time [13]. All measurements consist of 200 relaxations on 56 processors[4] with d_x sets to 60 data points per processor.

In the first set of measurements, we investigated the optimal combining tree degree for various numbers of data along the y-dimension. Figure 12 shows the optimal combining tree degree as well as the synchronization speedup achieved by using an optimal degree as opposed to a degree of four. As the data size along the y-dimension increases, the variance of the execution time also increases. The experimentally determined standard deviation σ, is given for each data size. As the standard deviation increases, the optimal degree increases from 4 to 32 and the resulting speedup increases from zero to 23 percent.

In the second set of measurements, we evaluate the performance improvements due to the dynamic placement barriers. We used the same SOR program with 56 processors and 210 data points along the y-dimension, resulting in an execution time of 9.5 ms and a statistical standard deviation of 110μs. Figure 13 presents the achieved performance improvements for various slacks and for a tree degree of 2, 4, and 16. We also present a combining tree of degree of two to test our dynamic placement barriers with as deep a combining tree as possible. Each set of measurements presents the combining tree depth seen by the last processor releasing the barrier[5] and the synchronization speedup achieved by using dynamic placement instead of static placement.

[3] The denominator, 16, corresponds to the cache sub-line size on the KSR1.

[4] We used 56 processors out of 64 to avoid using dedicated nodes (IOs) that would perturb the precision of the measurements.

[5] On the KSR1, the processors are organized in rings of 32 processors. To preserve the ring locality, our dynamic placement scheme does not cross ring boundaries. As a result, the number of tree levels corresponds to two subtrees of 32 processors merged by an additional level. This explains why a tree degree of 16 results in an initial tree depth of three.

	Slack				
	0ms	1ms	5ms	10ms	15ms
Degree: 2					
Last Proc Depth	4.38	3.93	2.23	1.755	1.67
Sync. Speedup	0.89	0.98	1.11	1.44	1.73
Degree: 4					
Last Proc Depth	3.24	2.88	1.63	1.75	1.44
Sync. Speedup	0.84	0.96	1.15	1.23	1.25
Degree: 16					
Last Proc Depth	2.88	2.7	1.72	1.57	1.24
Speedup	0.82	1.09	1.53	1.34	1.32

Figure 13: Performance of dynamic placement barriers on the KSR1.

First, we see that the combining tree depth for the last processor is effectively decreased from 4.38 to 1.67 and from 2.88 to 1.24 for trees of degree 2 and 16 respectively. Second, we see that dynamic placements result in a slower performance up to approximately a slack of 1 ms, and a performance improvement up to 1.73 and 1.32 for trees of degree 2 and 16 respectively.

8 Conclusion

Prior to our work, software synchronization barriers have been developed, evaluated and optimized under the assumption that all processors arrive simultaneously at the barrier. But, in most parallel systems there is some load imbalance. Therefore, we evaluate and reconfigure software combining trees for varying amounts of load imbalance.

Through analytic models and simulation results we show that the synchronization delay is minimized by choosing an optimum combining tree degree that is a function of the load imbalance. Thus, software barriers configured under the common assumption of simultaneous arrival are not merely over-designed for the probably more common case of distributed arrivals but have higher synchronization delays. Our analytic model can be used by a compiler to estimate the optimum degree and this estimate yields a performance that is within 7% of the actual optimum obtained by exhaustive simulation. This finding also indicates the feasibility of barriers that would adapt their degree at run time to minimize their synchronization delay.

Fuzzy barriers are effective in reducing the impact of nondeterministic load imbalance on overall performance. These barrier constructs also tend to distribute the arrival times of processors at a barrier over the slack interval. As a result, higher degree combining trees perform better when fuzzy barriers are used. Furthermore, processor arrival times are asymmetrically distributed with a few processors being much slower than average.

We show that dynamic placement schemes are very effective when fuzzy barriers are used. The average depth of the last processor to arrive at the barrier is indicative of the update delay component of the overall synchronization delay. The average depth reduces from close to L to close to 1.2 as the slack increases.

Since the contention delay component is small once the slack is sufficiently large, a speedup of close to L/1.2 is obtained over the static placement scheme for large slack.

The two techniques are combined and evaluated: a suitable tree degree is chosen based on load imbalance considerations and a dynamic placement scheme is used to exploit the last processor predictability under fuzzy barriers. The resulting synchronization delay is relatively insensitive to the number of processors when sufficient slack is present. These experiments demonstrate that software barriers implemented using simple hardware locks are scalable to large numbers of processors provided slack is available.

Acknowledgements

This work was supported in part by the Office of Naval Research under grant number N00014-93-1-0163 and by Hewlett-Packard. The University of Michigan's Center for Parallel Processing, site of the KSR1, is partially funded by NSF Grant CDA-92-14296.

References

[1] A. Gottlieb et al., "The NYU ultracomputer-designing an MIMD shared memory parallel computer," *IEEE Transactions on Computers*, vol. 32, no. 2, pp. 175–189, February 1983.

[2] G. Pfister et al., "The IBM research parallel processor prototype (RP3): Introduction and architecture," *Proceedings of the International Conference on Parallel Processing*, pp. 764–771, August 1985.

[3] *Cray T3D System Architecture Overview*, Cray Research, Inc, revision 1.c edition, September 1993.

[4] G. Pfister and V. A. Norton, "Hot spot contention and combining in multistage interconnection networks," *IEEE Transactions on Computers*, vol. C-34, no. 4, pp. 943–948, October 1985.

[5] M. M. Michael and M. L. Scott, "Fast mutual exclusion, even with contention," Technical Report TR-460, University of Rochester, 1993.

[6] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, "Distributing hot-spots addressing in large-scale multiprocessors," *IEEE Transactions on Computers*, vol. 36, no. 4, pp. 388–395, April 1987.

[7] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, February 1991.

[8] A. Agrawal and M. Cherian, "Adaptive backoff synchronization techniques," *Proceedings of the Sixteenth Annual International Symposium on Computer Architecture*, pp. 396–406, May 1989.

[9] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficient synchronization primitives for large-scale cache-coherent multiprocessors," *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 64–75, apr 1989.

[10] J. Lee and U. Ramachandran, "Synchronization with multiprocessor cache," *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, pp. 27–37, May 1990.

[11] R. Gupta, "The fuzzy barrier: A mechanism for high speed synchronization of processors," *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 54–63, 1989.

[12] R. Gupta, "Loop displacement: An approach for transforming and scheduling loops for parallel execution," *Proceedings of Supercomputing '90*, pp. 388–397, 1990.

[13] A. E. Eichenberger and S. G. Abraham, "Modeling load imbalance and fuzzy barriers for scalable shared-memory multiprocessors," *Proceeding of the 28th Hawaii International Conference on System Sciences*, vol. I, pp. 262–271, January 1995.

[14] J. Nguyen, *Compiler Analysis to Implement Point-to-Point Synchronization in Parallel Programs*, PhD thesis, MIT, August 1993.

[15] V. S. Adve and M. K. Vernon, "The influence of random delays on parallel execution times," *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 61–73, 1993.

[16] M. Dubois and F. A. Briggs, "Performance of synchronized iterative processes in multiprocessor systems," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 419–431, July 1982.

[17] V. Sarkar, "Determining average program execution times and their variance," *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, vol. 24, no. 7, pp. 298–312, 1989.

[18] C. P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1001–1016, October 1985.

[19] S. Madala and J. B. Sinclair, "Performance of synchronous parallel algorithms with regular structure," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 1, pp. 105–116, January 1991.

[20] M. D. Durand, T. Montaut, L. Kervella, and W. Jalby, "Impact of memory contention on dynamic scheduling on numa multiprocessors," *Proceedings of the International Conference on Parallel Processing*, vol. 1, pp. 258–267, 1993.

[21] T. S. Axelrod, "Effects of synchronization barriers on multiprocessor performance," *Parallel Computing*, vol. 3, pp. 129–140, 1986.

[22] C. J. Beckmann and C. D. Polychronopoulos, "The effect of barrier synchronization and scheduling overhead on parallel loops," *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. 200–204, August 1989.

[23] A. H.-S. Ang and W. H. Tang, *Probability Concepts In Engineering Planning And Design*, volume 2, New York : Wiley, 1984.

[24] *KSR1 Principles of Operation*, Kendall Square Research Corporation, 1991.