Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors

John M. Mellor-Crummey* Michael L. Scott†

January 1991

Abstract

Busy-wait techniques are heavily used for mutual exclusion and barrier synchronization in shared-memory parallel programs. Unfortunately, typical implementations of busy-waiting tend to produce large amounts of memory and interconnect contention, introducing performance bottlenecks that become markedly more pronounced as applications scale. We argue that this problem is not fundamental, and that one can in fact construct busy-wait synchronization algorithms that induce no memory or interconnect contention. The key to these algorithms is for every processor to spin on separate *locally-accessible* flag variables, and for some other processor to terminate the spin with a single remote write operation at an appropriate time. Flag variables may be locally-accessible as a result of coherent caching, or by virtue of allocation in the local portion of physically distributed shared memory.

We present a new scalable algorithm for spin locks that generates $O(1)$ remote references per lock acquisition, independent of the number of processors attempting to acquire the lock. Our algorithm provides reasonable latency in the absence of contention, requires only a constant amount of space per lock, and requires no hardware support other than a swap-with-memory instruction. We also present a new scalable barrier algorithm that generates $O(1)$ remote references per processor reaching the barrier, and observe that two previously-known barriers can likewise be cast in a form that spins only on locally-accessible flag variables. None of these barrier algorithms requires hardware support beyond the usual atomicity of memory reads and writes.

We compare the performance of our scalable algorithms with other software approaches to busy-wait synchronization on both a Sequent Symmetry and a BBN Butterfly. Our principal conclusion is that *contention due to synchronization need not be a problem in large-scale shared-memory multiprocessors.* The existence of scalable algorithms greatly weakens the case for costly special-purpose hardware support for synchronization, and provides a case against so-called "dance hall" architectures, in which shared memory locations are equally far from all processors.

---

*Center for Research on Parallel Computation, Rice University, P.O. Box 1892, Houston, TX 77251-1892. Internet address: `johnmc@rice.edu`. Supported in part by the National Science Foundation under Cooperative Agreement CCR-8809615.

†Computer Science Department, University of Rochester, Rochester, NY 14627. Internet address: `scott@cs.rochester.edu`. Supported in part by the National Science Foundation under Institutional Infrastructure grant CDA-8822724.

1 Introduction

Techniques for efficiently coordinating parallel computation on MIMD, shared-memory multiprocessors are of growing interest and importance as the scale of parallel machines increases. On shared-memory machines, processors communicate by sharing data structures. To ensure the consistency of shared data structures, processors perform simple operations by using hardware-supported atomic primitives and coordinate complex operations by using synchronization constructs and conventions to protect against overlap of conflicting operations.

Synchronization constructs can be divided into two classes: *blocking* constructs that de-schedule waiting processes, and *busy-wait* constructs in which processes repeatedly test shared variables to determine when they may proceed. Busy-wait synchronization is fundamental to parallel programming on shared-memory multiprocessors and is preferred over scheduler-based blocking when scheduling overhead exceeds expected wait time, when processor resources are not needed for other tasks (so that the lower wake-up latency of busy waiting need not be balanced against an opportunity cost), or when scheduler-based blocking is inappropriate or impossible (for example in the kernel of an operating system).

Two of the most widely used busy-wait synchronization constructs are spin locks and barriers. Spin locks provide a means for achieving mutual exclusion (ensuring that only one processor can access a particular shared data structure at a time) and are a basic building block for synchronization constructs with richer semantics, such as semaphores and monitors. Spin locks are ubiquitously used in the implementation of parallel operating systems and application programs. Barriers provide a means of ensuring that no processes advance beyond a particular point in a computation until all have arrived at that point. They are typically used to separate "phases" of an application program. A barrier might guarantee, for example, that all processes have finished updating the values in a shared matrix in step t before any processes use the values as input in step $t + 1$.

The performance of locks and barriers is a topic of great importance. Spin locks are generally employed to protect very small critical sections, and may be executed an enormous number of times in the course of a computation. Barriers, likewise, are frequently used between brief phases of data-parallel algorithms (*e.g.*, successive relaxation), and may be a major contributor to run time. Unfortunately, typical implementations of busy-waiting tend to produce large amounts of memory and interconnection network contention, which causes performance bottlenecks that become markedly more pronounced in larger machines and applications. As a consequence, the overhead of busy-wait synchronization is widely regarded as a serious performance problem [2, 6, 11, 14, 38, 49, 51].

When many processors busy-wait on a single synchronization variable, they create a *hot spot* that is the target of a disproportionate share of the network traffic. Pfister and Norton [38] showed that the presence of hot spots can severely degrade performance for all traffic in multistage interconnection networks, not just traffic due to synchronizing processors. As part of a larger study, Agarwal and Cherian [2] investigated the impact of synchronization on overall program performance. Their simulations of benchmarks on a cache-coherent multiprocessor indicate that memory references due to synchronization cause cache line invalidations much more often than non-synchronization references. In simulations of the benchmarks on a 64-processor "dance hall" machine (in which each access to a shared variable traverses the processor-memory interconnection network), they observed that synchronization accounted for as much as 49% of total network traffic.

In response to performance concerns, the history of synchronization techniques has displayed a trend toward increasing hardware support. Early algorithms assumed only the ability to read and write individual memory locations atomically. They tended to be subtle, and costly in time and space, requiring both a large number of shared variables and a large number of operations to coordinate concurrent invocations of synchronization primitives [12, 25, 26, 36, 40]. Modern multiprocessors generally include more sophisticated atomic operations, permitting simpler and faster coordination strategies. Particularly common are various `fetch_and_`Φ operations [22], which atomically read, modify, and write a memory location. `Fetch_and_`Φ operations include `test_and_set`, `fetch_and_store` (`swap`), `fetch_and_add`, and `compare_and_swap`.[1]

More recently, there have been proposals for multistage interconnection networks that combine concurrent accesses to the same memory location [17, 37, 42], multistage networks that have special synchronization variables embedded in each stage of the network [21], and special-purpose cache hardware to maintain a queue of processors waiting for the same lock [14, 28, 35]. The principal purpose of these hardware primitives is to reduce the impact of busy waiting. Before adopting them, it is worth considering the extent to which software techniques can achieve a similar result.

For a wide range of shared-memory multiprocessor architectures, we contend that appropriate design of spin locks and barriers can eliminate all busy-wait contention. Specifically, by distributing data structures appropriately, we can ensure that each processor spins only on locally-accessible locations, locations that are not the target of spinning references by any other processor. All that is required in the way of hardware support is a simple set of `fetch_and_`Φ operations and a memory hierarchy in which each processor is able to read some portion of shared memory without using the interconnection network. On a machine with coherent caches, processors spin only on locations in their caches. On a machine in which shared memory is distributed (*e.g.*, the BBN Butterfly [8], the IBM RP3 [37], or a shared-memory hypercube [10]), processors spin only on locations in the local portion of shared memory.

The implication of our work is that efficient synchronization algorithms can be constructed *in software* for shared-memory multiprocessors of arbitrary size. Special-purpose synchronization hardware can offer only a small constant factor of additional performance for mutual exclusion, and at best a logarithmic factor for barrier synchronization.[2] In addition, the feasibility and performance of busy-waiting algorithms with local-only spinning provides a case against "dance-hall" architectures, in which shared memory locations are equally far from all processors.

We discuss the implementation of spin locks in section 2, presenting both existing approaches and a new algorithm of our own design. In section 3 we turn to the issue of barrier synchronization, explaining how existing approaches can be adapted to eliminate spinning on remote locations, and introducing a new design that achieves both a short critical path and the theoretical minimum total number of network transactions. We present performance results in section 4 for a variety of spin lock and barrier implementations, and discuss the implications of these results for software and hardware designers. Our conclusions are summarized in section 5.

2 Spin Locks

In this section we describe a series of five implementations for a mutual-exclusion spin lock. The first four appear in the literature in one form or another. The fifth is a novel lock of our own design. Each lock can be seen as an attempt to eliminate some deficiency in the previous design. Each assumes

[1] `Fetch_and_store` exchanges a register with memory. `Compare_and_swap` compares the contents of a memory location against a given value, and sets a condition code to indicate whether they are equal. If so, it replaces the contents of the memory with a second given value.

[2] Hardware combining can reduce the time to achieve a barrier from $O(\log P)$ to $O(1)$ steps if processors happen to arrive at the barrier simultaneously.

a shared-memory environment that includes certain `fetch_and_Φ` operations. As noted above, a substantial body of work has also addressed mutual exclusion using more primitive read and write atomicity; the complexity of the resulting solutions is the principal motivation for the development of `fetch_and_Φ` primitives. Other researchers have considered mutual exclusion in the context of distributed systems [31, 39, 43, 45, 46], but the characteristics of message passing are different enough from shared memory operations that solutions do not transfer from one environment to the other.

Our pseudo-code notation is meant to be more-or-less self explanatory. We have used line breaks to terminate statements, and indentation to indicate nesting in control constructs. The keyword `shared` indicates that a declared variable is to be shared among all processors. The declaration implies no particular physical location for the variable, but we often specify locations in comments and/or accompanying text. The keywords `processor private` indicate that each processor is to have a separate, independent copy of a declared variable. All of our atomic operations are written to take as their first argument the address of the memory location to be modified. All but `compare_and_swap` take the obvious one additional operand and return the old contents of the memory location. `Compare_and_swap (addr, old, new)` is defined as `if addr^ != old return false; addr^ := new; return true`. In one case (algorithm 3) we have used an `atomic_add` operation whose behavior is the same as calling `fetch_and_add` and discarding the result.

2.1 The Simple `Test_and_set` Lock

The simplest mutual exclusion lock, found in all operating system textbooks and widely used in practice, employs a polling loop to access a Boolean flag that indicates whether the lock is held. Each processor repeatedly executes a `test_and_set` instruction in an attempt to change the flag from false to true, thereby acquiring the lock. A processor releases the lock by setting it to false.

The principal shortcoming of the `test_and_set` lock is contention for the flag. Each waiting processor accesses the single shared flag as frequently as possible, using relatively expensive read-modify-write (`fetch_and_Φ`) instructions. The result is degraded performance, not only of the memory bank in which the lock resides, but also of the processor/memory interconnection network and, in a distributed shared-memory machine, the processor that owns the memory bank (as a result of stolen bus cycles).

`Fetch_and_Φ` instructions can be particularly expensive on cache-coherent multiprocessors, since each execution of such an instruction may cause many remote invalidations. To reduce this overhead, the `test_and_set` lock can be modified to use a `test_and_set` instruction only when a previous read indicates that the `test_and_set` might succeed. This so-called test-and–`test_and_set` technique [44] ensures that waiting processors poll with read requests during the time that a lock is held. Once the lock becomes available, some fraction of the waiting processors detect that the lock is free and perform a `test_and_set` operation, exactly one of which succeeds, but each of which causes remote invalidations on a cache-coherent machine.

The total amount of network traffic caused by busy-waiting on a `test_and_set` lock can be reduced further by introducing delay on each processor between consecutive probes of the lock. The simplest approach employs a constant delay; more elaborate schemes use some sort of backoff on unsuccessful probes. Anderson [5] reports the best performance with exponential backoff; our experiments confirm this result. Pseudo-code for a `test_and_set` lock with exponential backoff appears in algorithm 1. `Test_and_set` suffices when using a backoff scheme; test-and–`test_and_set` is not necessary.

```
type lock = (unlocked, locked)

procedure acquire_lock (L : ^lock)
    delay : integer := 1
    while test_and_set (L) = locked     // returns old value
        pause (delay)                   // consume this many units of time
        delay := delay * 2

procedure release_lock (L : ^lock)
    lock^ := unlocked
```

Algorithm 1: Simple `test_and_set` lock with exponential backoff.

2.2 The Ticket Lock

In a test-and–`test_and_set` lock, the number of read-modify-write operations is substantially less than for a simple `test_and_set` lock, but still potentially large. Specifically, it is possible for every waiting processor to perform a `test_and_set` operation every time the lock becomes available, even though only one can actually acquire the lock. We can reduce the number of `fetch_and_`Φ operations to one per lock acquisition with what we call a ticket lock. At the same time, we can ensure FIFO service by granting the lock to processors in the same order in which they first requested it. A ticket lock is fair in a strong sense; it eliminates the possibility of starvation.

A ticket lock consists of two counters, one containing the number of requests to acquire the lock, and the other the number of times the lock has been released. A processor acquires the lock by performing a `fetch_and_increment` operation on the request counter and waiting until the result (its *ticket*) is equal to the value of the release counter. It releases the lock by incrementing the release counter. In the terminology of Reed and Kanodia [41], a ticket lock corresponds to the busy-wait implementation of a semaphore using an eventcount and a sequencer. It can also be thought of as an optimization of Lamport's bakery lock [24], which was designed for fault-tolerance rather than performance. Instead of spinning on the release counter, processors using a bakery lock repeatedly examine the tickets of their peers.

Though it probes with read operations only (and thus avoids the overhead of unnecessary invalidations in coherent cache machines), the ticket lock still causes substantial memory and network contention through polling of a common location. As with the `test_and_set` lock, this contention can be reduced by introducing delay on each processor between consecutive probes of the lock. In this case, however, exponential backoff is clearly a bad idea. Since processors acquire the lock in FIFO order, overshoot in backoff by the first processor in line will delay all others as well, causing them to back off even farther. Our experiments suggest that a reasonable delay can be determined by using information not available with a `test_and_set` lock: namely, the number of processors already waiting for the lock. The delay can be computed as the difference between a newly-obtained ticket and the current value of the release counter.

Delaying for an appropriate amount of time requires an estimate of how long it will take each processor to execute its critical section and pass the lock to its successor. If this time is known exactly, it is in principle possible to acquire the lock with only two probes, one to determine the number of processors already in line (if any), and another (if necessary) to verify that one's predecessor in line has finished with the lock. This sort of accuracy is not likely in practice, however, since critical sections do not in general take identical, constant amounts of time. Moreover, delaying proportional to the expected *average* time to hold the lock is risky: if the processors already in line average less than the expected amount the waiting processor will delay too long and slow the entire system. A more appropriate constant of proportionality for the delay is the *minimum* time that a processor can hold the lock. Pseudo-code for a ticket lock with proportional backoff appears in algorithm 2. It assumes that the `new_ticket` and `now_serving` counters are large enough to accommodate the maximum number of simultaneous requests for the lock.

```
type lock = record
    next_ticket : unsigned integer := 0
    now_serving : unsigned integer := 0

procedure acquire_lock (L : ^lock)
    my_ticket : unsigned integer := fetch_and_increment (&L->next_ticket)
        // returns old value; arithmetic overflow is harmless
    loop
        pause (my_ticket - L->now_serving)
            // consume this many units of time
            // on most machines, subtraction works correctly despite overflow
        if L->now_serving = my_ticket
            return

procedure release_lock (L : ^lock)
    L->now_serving := L->now_serving + 1
```

Algorithm 2: Ticket lock with proportional backoff.

2.3 Array-Based Queuing Locks

Even using a ticket lock with proportional backoff, it is not possible to obtain a lock with an expected constant number of network transactions, due to the unpredictability of the length of critical sections. Anderson [5] and Graunke and Thakkar [18] have proposed locking algorithms that achieve the constant bound on cache-coherent multiprocessors that support atomic `fetch_and_increment` or `fetch_and_store`, respectively. The trick is for each processor to use the atomic operation to obtain the address of a location on which to spin. Each processor spins on a different location, in a different cache line. Anderson's experiments indicate that his queuing lock outperforms a `test_and_set` lock with exponential backoff on the Sequent Symmetry when more than six processors are competing for access. Graunke and Thakkar's experiments indicate that their lock outperforms a `test_and_set` lock on the same machine when more than three processors are competing.

Pseudo-code for Anderson's lock appears in algorithm 3;[3] Graunke and Thakkar's lock appears in algorithm 4. The likely explanation for the better performance of Graunke and Thakkar's lock is that `fetch_and_store` is supported directly in hardware on the Symmetry; `fetch_and_add` is not. To simulate `fetch_and_add`, Anderson protected his queue-based lock with an outer `test_and_set` lock. This outer lock (not shown in algorithm 3) introduces additional overhead and can cause contention when the critical section protected by the queue-based lock is shorter than the time required to acquire and release the outer lock. In this case, competing processors spend their time spinning on the outer `test_and_set` lock rather than the queue-based lock.

[3] Anderson's original pseudo-code did not address the issue of overflow, which causes his algorithm to fail unless `numprocs` $= 2^k$. Our variant of his algorithm addresses this problem.

```
type lock = record
    slots : array [0..numprocs -1] of (has_lock, must_wait)
        := (has_lock, must_wait, must_wait, ..., must_wait)
        // each element of slots should lie in a different memory module
        // or cache line
    next_slot : integer := 0

// parameter my_place, below, points to a private variable
// in an enclosing scope

procedure acquire_lock (L : ^lock, my_place : ^integer)
    my_place^ := fetch_and_increment (&L->next_slot)
        // returns old value
    if my_place^ mod numprocs = 0
        atomic_add (&L->next_slot, -numprocs)
        // avoid problems with overflow; return value ignored
    my_place^ := my_place^ mod numprocs
    repeat while L->slots[my_place^] = must_wait      // spin
    L->slots[my_place^] := must_wait                  // init for next time

procedure release_lock (L : ^lock, my_place : ^integer)
    L->slots[(my_place^ + 1) mod numprocs] := has_lock
```

Algorithm 3: Anderson's array-based queuing lock.

```
type lock = record
    slots : array [0..numprocs -1] of Boolean := true
        // each element of slots should lie in a different memory module
        // or cache line
    tail : record
        who_was_last : ^Boolean := 0
        this_means_locked : Boolean := false
        // this_means_locked is a one-bit quantity.
        // who_was_last points to an element of slots.
        // if all elements lie at even addresses, this tail "record"
        // can be made to fit in one word
processor private vpid : integer // a unique virtual processor index

procedure acquire_lock (L : ^lock)
    (who_is_ahead_of_me : ^Boolean, what_is_locked : Boolean)
        := fetch_and_store (&L->tail, (&slots[vpid], slots[vpid]))
    repeat while who_is_ahead_of_me^ = what_is_locked

procedure release_lock (L : ^lock)
    L->slots[vpid] := not L->slots[vpid]
```

Algorithm 4: Graunke and Thakkar's array-based queuing lock.

Neither Anderson nor Graunke and Thakkar included the ticket lock in their experiments. In qualitative terms, both the ticket lock (with proportional backoff) and the array-based queuing locks guarantee FIFO ordering of requests. Both the ticket lock and Anderson's lock use an atomic `fetch_and_increment` instruction. The ticket lock with proportional backoff is likely to require more network transactions on a cache-coherent multiprocessor, but fewer on a multiprocessor without coherently cached shared variables. The array-based queuing locks require space *per lock* linear in the number of processors, whereas the ticket lock requires only a small constant amount of space.[4] We provide quantitative comparisons of the locks' performance in section 4.3.

2.4 A New List-Based Queuing Lock

We have devised a new mechanism called the MCS lock (after our initials) that

- guarantees FIFO ordering of lock acquisitions;
- spins on locally-accessible flag variables only;
- requires a small constant amount of space per lock; and
- works equally well (requiring only $O(1)$ network transactions per lock acquisition) on machines with and without coherent caches.

The first of these advantages is shared with the ticket lock and the array-based queuing locks, but not with the `test_and_set` lock. The third is shared with the `test_and_set` and ticket locks, but not with the array-based queuing locks. The fourth advantage is in large part a consequence of the second, and is unique to the MCS lock.

Our lock was inspired by the QOLB (Queue On Lock Bit) primitive proposed for the cache controllers of the Wisconsin Multicube [14], but is implemented entirely in software. It requires an atomic `fetch_and_store` (`swap`) instruction, and benefits from the availability of `compare_and_swap`. Without `compare_and_swap` we lose the guarantee of FIFO ordering and introduce the theoretical possibility of starvation, though lock acquisitions are likely to remain very nearly FIFO in practice.

Pseudo-code for our lock appears in algorithm 5. Every processor using the lock allocates a `qnode` record containing a queue link and a Boolean flag. Each processor employs one additional temporary variable during the `acquire_lock` operation. Processors holding or waiting for the lock are chained together by the links. Each processor spins on its own locally-accessible flag. The lock itself contains a pointer to the `qnode` record for the processor at the tail of the queue, or a `nil` if the lock is not held. Each processor in the queue holds the address of the record for the processor behind it—the processor it should resume after acquiring and releasing the lock. `Compare_and_swap` enables a processor to determine whether it is the only processor in the queue, and if so remove itself correctly, as a single atomic action. The spin in `acquire_lock` waits for the lock to become free. The spin in `release_lock` compensates for the timing window between the `fetch_and_store` and the assignment to `predecessor->next` in `acquire_lock`. Both spins are local.

Figure 1, parts (a) through (e), illustrates a series of `acquire_lock` and `release_lock` operations. The lock itself is represented by a box containing an 'L.' The other rectangles are `qnode`

[4] At first glance, one might suspect that the flag bits of the Graunke and Thakkar lock could be allocated on a per-processor basis, rather than a per-lock basis. Once a processor releases a lock, however, it cannot use its bit for anything else until some other processor has acquired the lock it released.

```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode

// parameter I, below, points to a qnode record allocated
// (in an enclosing scope) in shared memory locally-accessible
// to the invoking processor

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor : ^qnode := fetch_and_store (L, I)
    if predecessor != nil      // queue was non-empty
        I->locked := true
        predecessor->next := I
        repeat while I->locked              // spin

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil        // no known successor
        if compare_and_swap (L, I, nil)
            return
            // compare_and_swap returns true iff it swapped
        repeat while I->next = nil          // spin
    I->next->locked := false
```

Algorithm 5: The MCS list-based queuing lock.

records. A box with a slash through it represents a `nil` pointer. Non-`nil` pointers are directed arcs. In (a) the lock is free. In (b), processor 1 has acquired the lock. It is running (indicated by the 'R'), thus its locked flag is irrelevant (indicated by putting the 'R' in parentheses). In (c), two more processors have entered the queue while the lock is still held by processor 1. They are blocked spinning on their locked flags (indicated by the 'B's). In (d), processor 1 has completed, and has changed the locked flag of processor 2 so that it is now running. In (e), processor 2 has completed, and has similarly unblocked processor 3. If no more processors enter the queue in the immediate future, the lock will return to the situation in (a) when processor 3 completes its critical section.

Alternative code for the `release_lock` operation, without `compare_and_swap`, appears in algorithm 6. Like the code in algorithm 5, it spins on processor-specific, locally-accessible memory locations only, requires constant space per lock, and requires only $O(1)$ network transactions regardless of whether the machine provides coherent caches. Its disadvantages are extra complexity and the loss of strict FIFO ordering.

Parts (e) through (h) of figure 1 illustrate the subtleties of the alternative code for `release_lock`. In the original version of the lock, `compare_and_swap` ensures that updates to the tail of the queue happen atomically. There are no processors waiting in line if and only if the tail pointer of the queue points to the processor releasing the lock. Inspecting the processor's next pointer is solely an optimization, to avoid unnecessary use of a comparatively expensive atomic instruction. Without `compare_and_swap`, inspection and update of the tail pointer cannot occur atomically. When processor 3 is ready to release its lock, it assumes that no other processor is in line if its next pointer is `nil`. In other words, it assumes that the queue looks the way it does in (e). (It could

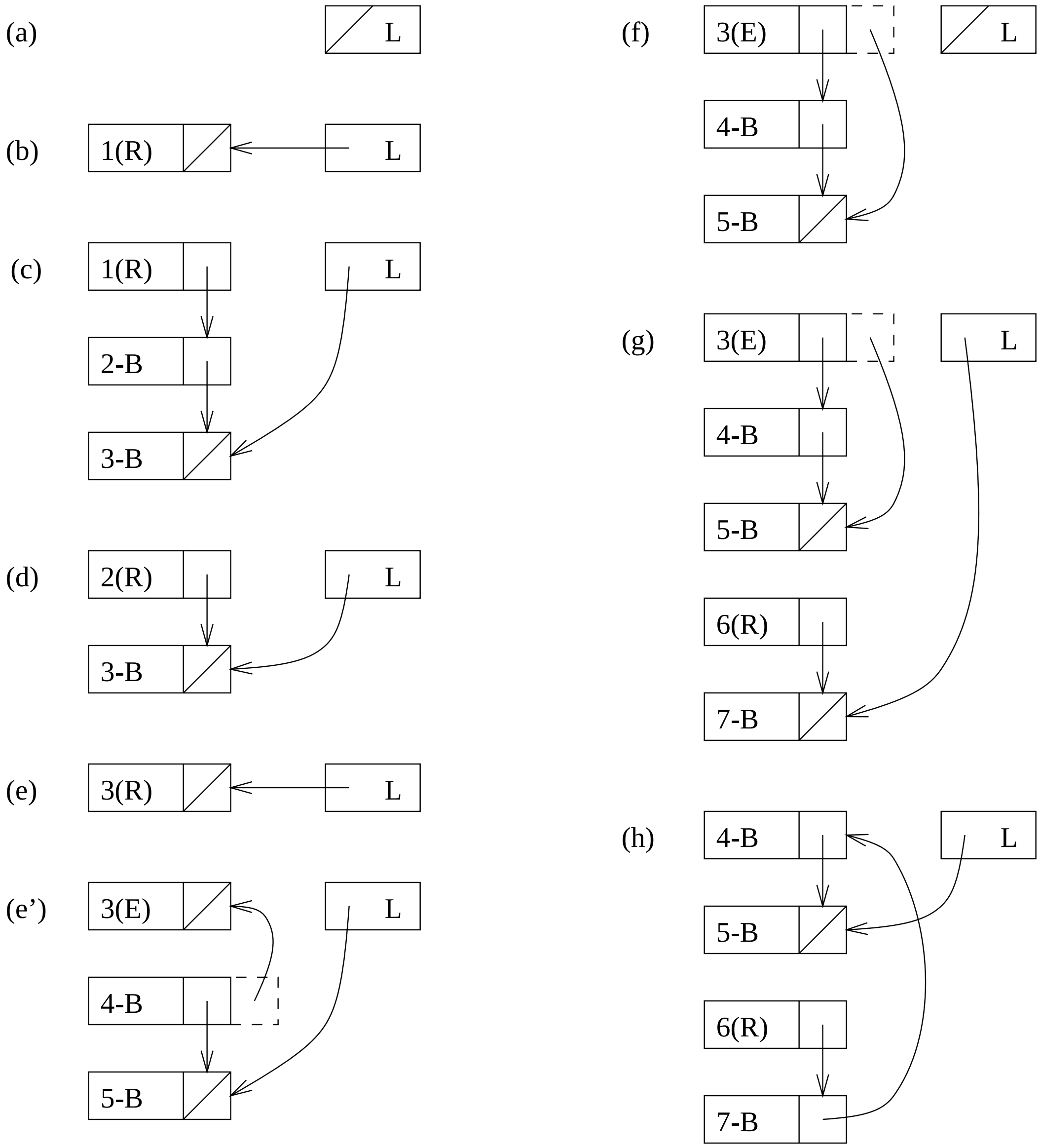

Figure 1: Pictorial example of MCS locking protocol in the presence of competition.

```
procedure release_lock (L : ^lock, I : ^qnode)
    if I->next = nil        // no known successor
        old_tail : ^qnode := fetch_and_store (L, nil)
        if old_tail = I     // I really had no successor
            return
        // we have accidentally removed some processor(s) from the queue;
        // we need to put them back
        usurper := fetch_and_store (L, old_tail)
        repeat while I->next = nil          // wait for pointer to victim list
        if usurper != nil
            // somebody got into the queue ahead of our victims
            usurper->next := I->next        // link victims after the last usurper
        else
            I->next->locked := false
    else
        I->next->locked := false
```

Algorithm 6: Code for `release_lock`, without `compare_and_swap`.

equally well make this assumption after inspecting the tail pointer and finding that it points to itself, but the next pointer is local and the tail pointer is probably not.) This assumption may be incorrect because other processors may have linked themselves into the queue between processor 3's inspection and its subsequent update of the tail. The queue may actually be in the state shown in (e′), with one or more processors in line behind processor 3, the first of which has yet to update 3's next pointer. (The 'E' in parentheses on processor 3 indicates that it is exiting its critical section; the value of its locked flag is irrelevant.)

When new processors enter the queue during this timing window, the data structure temporarily takes on the form shown in (f). The return value of processor 3's first `fetch_and_store` in `release_lock` (shown in the extra dotted box) is the tail pointer for a list of processors that have accidentally been linked out of the queue. By waiting for its next pointer to become non-`nil`, processor 3 obtains a head pointer for this "victim" list. It can patch the victim processors back into the queue, but before it does so additional processors ("usurpers") may enter the queue with the first of them acquiring the lock, as shown in (g). Processor 3 puts the tail of the victim list back into the tail pointer of the queue with a `fetch_and_store`. If the return value of this `fetch_and_store` is `nil`, processor 3 unblocks its successor. Otherwise, as shown in (h), processor 3 inserts the victim list behind the usurpers by writing its next pointer (the head of the victim list) into the next pointer of the tail of the usurper list. In either case, the structure of the queue is restored.

To demonstrate formal correctness of the MCS lock we can prove mutual exclusion, deadlock freedom, and fairness for a version of the algorithm in which the entire `acquire_lock` and `release_lock` procedures comprise atomic actions, broken only when a processor waits (in `acquire_lock`) for the lock to become available. We can then refine this algorithm through a series of correctness-preserving transformations into the code in algorithm 5. The first transformation breaks the atomic action of `acquire_lock` in two, and introduces auxiliary variables that indicate when a processor has modified the tail pointer of the queue to point at its `qnode` record, but has not yet modified its predecessor's next pointer. To preserve the correctness proof in the face of this transformation, we (1) relax the invariants on queue structure to permit a non-tail processor to have a `nil` next pointer, so long as the auxiliary variables indicate that its successor is in the timing window, and (2) introduce a spin in `release_lock` to force the processor to wait for its next pointer to be updated before using it. The second transformation uses proofs of interference freedom to move statements outside of the three atomic actions. The action containing the assignment to a predecessor's next pointer can then be executed without explicit atomicity; it inspects or modifies only one variable that is modified or inspected in any other process. The other two actions are reduced to the functionality of `fetch_and_store` and `compare_and_swap`. The details of this argument are straightforward but lengthy (they appear as an appendix to the technical report version of this paper [33]); we find the informal description and pictures above more intuitively convincing.

3 Barriers

Barriers have received a great deal of attention in the literature, more so even than spin locks, and the many published algorithms differ significantly in notational conventions and architectural assumptions. We present five different barriers in this section, four from the literature and one of our own design. We have modified some of the existing barriers to increase their locality of reference or otherwise improve their performance; we note where we have done so.

3.1 Centralized Barriers

In a centralized implementation of barrier synchronization, each processor updates a small amount of shared state to indicate its arrival, and then polls that state to determine when all of the processors have arrived. Once all of the processors have arrived, each processor is permitted to continue past the barrier. Like the `test_and_set` spin lock, centralized barriers are of uncertain origin. Essentially equivalent algorithms have undoubtedly been invented by numerous individuals.

Most barriers are designed to be used repeatedly (to separate phases of a many-phase algorithm, for example). In the most obvious formulation, each instance of a centralized barrier begins and ends with identical values for the shared state variables. Each processor must spin twice per instance, once to ensure that all processors have left the previous barrier, and again to ensure that all processors have arrived at the current barrier. Without the first spin, it is possible for a processor to mistakenly pass through the current barrier because of state information being used by processors still leaving the previous barrier. Two barrier algorithms proposed by Tang and Yew (the first algorithm appears in [48, Algorithm 3.1] and [49, p. 3]; the second algorithm appears in [49, Algorithm 3.1]) suffer from this type of flaw.

We can reduce the number of references to the shared state variables, and simultaneously eliminate one of the two spinning episodes, by "reversing the sense" of the variables (and leaving them with different values) between consecutive barriers [19].[5] The resulting code is shown in algorithm 7. Arriving processors decrement `count` and then wait until `sense` has a different value than it did in the previous barrier. The last arriving processor resets `count` and reverses `sense`. Consecutive barriers cannot interfere with each other because all operations on `count` occur before `sense` is toggled to release the waiting processors.

Lubachevsky [29] presents a similar barrier algorithm that uses two shared counters and a processor private two-state flag. The private flag selects which counter to use; consecutive barriers use alternate counters. Another similar barrier can be found in library packages distributed by Sequent Corporation for the Symmetry multiprocessor. Arriving processors read the current value

[5] A similar technique appears in [4, p. 445], where it is credited to Isaac Dimitrovsky.

```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true

procedure central_barrier
    local_sense := not local_sense   // each processor toggles its own sense
    if fetch_and_decrement (&count) = 1
        count := P
        sense := local_sense         // last processor toggles global sense
    else
        repeat until sense = local_sense
```

Algorithm 7: A sense-reversing centralized barrier.

of a shared epoch number, update a shared counter, and spin until the epoch number changes. The last arriving processor re-initializes the counter and advances the epoch number.

The potential drawback of centralized barriers is the spinning that occurs on a single, shared location. Because processors do not in practice arrive at a barrier simultaneously, the number of busy-wait accesses will in general be far above the minimum.[6] On broadcast-based cache-coherent multiprocessors, these accesses may not be a problem. The shared flag or sense variable is replicated into the cache of every waiting processor, so subsequent busy-wait accesses can be satisfied without any network traffic. This shared variable is written only when the barrier is achieved, causing a single broadcast invalidation of all cached copies.[7] All busy-waiting processors then acquire the new value of the variable and are able to proceed. On machines without coherent caches, however, or on machines with directory-based caches without broadcast, busy-wait references to a shared location may generate unacceptable levels of memory and interconnect contention.

To reduce the interconnection network traffic caused by busy waiting on a barrier flag, Agarwal and Cherian [2] investigated the utility of adaptive backoff schemes. They arranged for processors to delay between successive polling operations for geometrically-increasing amounts of time. Their results indicate that in many cases such exponential backoff can substantially reduce the amount of network traffic required to achieve a barrier. However, with this reduction in network traffic often comes an increase in latency at the barrier. Processors in the midst of a long delay do not immediately notice when all other processors have arrived. Their departure from the barrier is therefore delayed, which in turn delays their arrival at subsequent barriers.

Agarwal and Cherian also note that for systems with more than 256 processors, for a range of arrival intervals and delay ratios, backoff strategies are of limited utility for barriers that spin on a single flag [2]. In such large-scale systems, the number of network accesses per processor increases sharply as collisions in the interconnection network cause processors to repeat accesses. These observations imply that centralized barrier algorithms will not scale well to large numbers of processors, even using adaptive backoff strategies. Our experiments (see section 4.4) confirm this conclusion.

[6]Commenting on Tang and Yew's barrier algorithm (algorithm 3.1 in [48]), Agarwal and Cherian [2] show that on a machine in which contention causes memory accesses to be aborted and retried, the expected number of memory accesses initiated by each processor to achieve a single barrier is linear in the number of processors participating, even if processors arrive at the barrier at approximately the same time.

[7]This differs from the situation in simple spin locks, where a waiting processor can expect to suffer an invalidation for every contending processor that acquires the lock before it.

```
type node = record
    k : integer             // fan-in of this node
    count : integer         // initialized to k
    locksense : Boolean     // initially false
    parent : ^node          // pointer to parent node; nil if root

shared nodes : array [0..P-1] of node
    // each element of nodes allocated in a different memory module or cache line
processor private sense : Boolean  := true
processor private mynode : ^node    // my group's leaf in the combining tree

procedure combining_barrier
    combining_barrier_aux (mynode)      // join the barrier
    sense := not sense                  // for next barrier

procedure combining_barrier_aux (nodepointer : ^node)
    with nodepointer^ do
        if fetch_and_decrement (&count) = 1     // last one to reach this node
            if parent != nil
                combining_barrier_aux (parent)
            count := k                          // prepare for next barrier
            locksense := not locksense          // release waiting processors
        repeat until locksense = sense
```

Algorithm 8: A software combining tree barrier with optimized wakeup.

3.2 The Software Combining Tree Barrier

To reduce hot-spot contention for synchronization variables, Yew, Tzeng, and Lawrie [51] have devised a data structure known as a software combining tree. Like hardware combining in a multistage interconnection network [17], a software combining tree serves to collect multiple references to the same shared variable into a single reference whose effect is the same as the combined effect of the individual references. A shared variable that is expected to be the target of multiple concurrent accesses is represented as a tree of variables, with each node in the tree assigned to a different memory module. Processors are divided into groups, with one group assigned to each leaf of the tree. Each processor updates the state in its leaf. If it discovers that it is the last processor in its group to do so, it continues up the tree, updating its parent to reflect the collective updates to the child. Proceeding in this fashion, late-coming processors eventually propagate updates to the root of the tree.

Combining trees are presented as a general technique that can be used for several purposes. At every level of the tree, atomic instructions are used to combine the arguments to write operations or to split the results of read operations. In the context of this general framework, Tang and Yew [49] describe how software combining trees can be used to implement a barrier. Writes into one tree are used to determine that all processors have reached the barrier; reads out of a second are used to allow them to continue. Algorithm 8 shows an optimized version of the combining tree barrier. We have used the sense-reversing technique to avoid overlap of successive barriers without requiring two spinning episodes per barrier, and have replaced the atomic instructions of the second combining tree with simple reads, since no real information is returned.

Each processor begins at a leaf of the combining tree, and decrements its leaf's `count` variable. The last processor to reach each node in the tree continues up to the next level. The processor that reaches the root of the tree begins a reverse wave of updates to `locksense` flags. As soon as it awakes, each processor retraces its path through the tree unblocking its siblings at each node along the path. Simulations by Yew, Tzeng, and Lawrie [51] show that a software combining tree can significantly decrease memory contention and prevent tree saturation (a form of network congestion that delays the response of the network to large numbers of references [38]) in multistage interconnection networks by distributing accesses across the memory modules of the machine.

The principal shortcoming of the combining tree barrier, from our point of view, is that is requires processors to spin on memory locations that cannot be statically determined, and on which other processors also spin. On broadcast-based cache-coherent machines, processors may obtain local copies of the tree nodes on which they spin, but on other machines (including the Cedar machine which Yew, Tzeng, and Lawrie simulated), processors will spin on remote locations, leading to unnecessary contention for interconnection network bandwidth. In section 3.5 we present a new tree-based barrier algorithm in which each processor spins on its own unique location, statically determined and thus presumably locally accessible. Our algorithm uses no atomic instructions other than read and write, and performs the minimum possible number of operations across the processor-memory interconnect.

3.3 The Dissemination Barrier

Brooks [9] has proposed a symmetric "butterfly barrier," in which processors participate as equals, performing the same operations at each step. Each processor in a butterfly barrier participates in a sequence of $\sim \log_2 P$ pairwise synchronizations. In round k (counting from zero), processor i synchronizes with processor $i \oplus 2^k$, where $\oplus$ is the exclusive `or` operator. If the number of processors is not a power of 2, then existing processors stand in for the missing ones, thereby participating in as many as $2\lceil \log_2 P \rceil$ pairwise synchronizations.

Hensgen, Finkel, and Manber [19] describe a "dissemination barrier" that improves on Brooks's algorithm by employing a more efficient pattern of synchronizations and by reducing the cost of each synchronization. Their barrier takes its name from an algorithm developed to disseminate information among a set of processes. In round k, processor i signals processor $(i + 2^k) \bmod P$ (Synchronization is no longer pairwise). This pattern does not require existing processes to stand in for missing ones, and therefore requires only $\lceil \log_2 P \rceil$ synchronization operations on its critical path, regardless of P. Reference [19] contains a more detailed description of the synchronization pattern and a proof of its correctness.

For each signalling operation of the dissemination barrier, Hensgen, Finkel, and Manber use alternating sets of variables in consecutive barrier episodes, avoiding interference without requiring two separate spins in each operation. They also use sense reversal to avoid resetting variables after every barrier. The first change also serves to eliminate remote spinning. The authors motivate their algorithmic improvements in terms of reducing the number of instructions executed in the course of a signalling operation, but we consider the elimination of remote spinning to be an even more important benefit. The flags on which each processor spins are statically determined, and no two processors spin on the same flag. Each flag can therefore be located near the processor that reads it, leading to local-only spinning on any machine with local shared memory or coherent caches.

Algorithm 9 presents the dissemination barrier. The `parity` variable controls the use of alternating sets of flags in successive barrier episodes. On a machine with distributed shared memory

```
type flags = record
    myflags : array [0..1] of array [0..LogP-1] of Boolean
    partnerflags : array [0..1] of array [0..LogP-1] of ^Boolean

processor private parity : integer := 0
processor private sense : Boolean := true
processor private localflags : ^flags
shared allnodes : array [0..P-1] of flags
    // allnodes[i] is allocated in shared memory
    // locally accessible to processor i

// on processor i, localflags points to allnodes[i]
// initially allnodes[i].myflags[r][k] is false for all i, r, k
// if j = (i+2^k) mod P, then for r = 0, 1:
//    allnodes[i].partnerflags[r][k] points to allnodes[j].myflags[r][k]

procedure dissemination_barrier
    for instance : integer := 0 to LogP-1
        localflags^.partnerflags[parity][instance]^ := sense
        repeat until localflags^.myflags[parity][instance] = sense
    if parity = 1
        sense := not sense
    parity := 1 - parity
```

Algorithm 9: The scalable, distributed dissemination barrier with only local spinning.

and without coherent caches, the shared `allnodes` array would be scattered statically across the memory banks of the machine, or replaced by a scattered set of variables.

3.4 Tournament Barriers

Hensgen, Finkel, and Manber [19] and Lubachevsky [30] have also devised tree-style "tournament" barriers. The processors involved in a tournament barrier begin at the leaves of a binary tree, much as they would in a combining tree of fan-in two. One processor from each node continues up the tree to the next "round" of the tournament. At each stage, however, the "winning" processor is statically determined, and there is no need for `fetch_and_Φ` instructions.

In round k (counting from zero) of Hensgen, Finkel, and Manber's barrier, processor i sets a flag awaited by processor j, where $i \equiv 2^k \pmod{2^{k+1}}$ and $j = i - 2^k$. Processor i then drops out of the tournament and busy waits on a global flag for notice that the barrier has been achieved. Processor j participates in the next round of the tournament. A complete tournament consists of $\lceil \log_2 P \rceil$ rounds. Processor 0 sets a global flag when the tournament is over.

Lubachevsky [30] presents a CREW (concurrent read, exclusive write) tournament barrier that uses a global flag for wakeup, similar to that of Hensgen, Finkel, and Manber. He also presents an EREW (exclusive read, exclusive write) tournament barrier in which each processor spins on separate flags in a binary wakeup tree, similar to wakeup in a binary combining tree using algorithm 8.

Because all processors busy wait on a single global flag, Hensgen, Finkel, and Manber's tournament barrier and Lubachevsky's CREW barrier are appropriate for multiprocessors that use broadcast to maintain cache consistency. They will cause heavy interconnect traffic, however, on machines that lack coherent caches, or that limit the degree of cache line replication. Lubachevsky's EREW tournament could be used on any multiprocessor with coherent caches, including those that use limited-replication directory-based caching without broadcast. Unfortunately, in Lubachevsky's EREW barrier algorithm, each processor spins on a non-contiguous set of elements in an array, and no simple scattering of these elements will suffice to eliminate spinning-related network traffic on a machine without coherent caches.

By modifying Hensgen, Finkel, and Manber's tournament barrier to use a wakeup tree, we have constructed an algorithm in which each processor spins on its own set of contiguous, statically allocated flags (see algorithm 10). The resulting code is able to avoid spinning across the interconnection network, both on cache-coherent machines and on distributed shared memory multiprocessors. In addition to employing a wakeup tree, we have modified Hensgen, Finkel, and Manber's algorithm to use sense reversal to avoid re-initializing flag variables in each round. These same modifications have been discovered independently by Craig Lee of Aerospace Corporation [27].

Hensgen, Finkel, and Manber provide performance figures for the Sequent Balance (a bus-based, cache-coherent multiprocessor), comparing their tournament algorithm against the dissemination barrier, as well as Brooks's butterfly barrier. They report that the tournament barrier outperforms the dissemination barrier when $P > 16$. The dissemination barrier requires $O(P \log P)$ network transactions, while the tournament barrier requires only $O(P)$. Beyond 16 processors, the additional factor of $\log P$ in bus traffic for the dissemination barrier dominates the higher constant of the tournament barrier. However, on scalable multiprocessors with multi-stage interconnection networks, many of the network transactions required by the dissemination barrier algorithm can proceed in parallel without interference.

3.5 A New Tree-Based Barrier

We have devised a new barrier algorithm that

- spins on locally-accessible flag variables only;
- requires only $O(P)$ space for P processors;
- performs the theoretical minimum number of network transactions $(2P - 2)$ on machines without broadcast; and
- performs $O(\log P)$ network transactions on its critical path.

To synchronize P processors, our barrier employs a pair of P-node trees. Each processor is assigned a unique tree node, which is linked into an arrival tree by a parent link, and into a wakeup tree by a set of child links. It is useful to think of these as separate trees, because the fan-in in the arrival tree differs from the fan-out in the wakeup tree. We use a fan-in of 4 (1) because it produced the best performance in Yew, Tzeng, and Lawrie's experiments with software combining, and (2) because the ability to pack four bytes in a word permits an optimization on many machines in which a parent can inspect status information for all of its children simultaneously at the same cost as inspecting the status of only one. We use a fan-out of 2 because it results in the shortest critical path to resume P spinning processors for a tree of uniform degree. To see this, note that in a p-node tree with fan-out k (and approximately $\log_k p$ levels), the last processor to awaken will be the kth child of the kth child ... of the root. Because $k - 1$ other children are awoken first at each level, the last processor awakes at the end of a serial chain of approximately $k \log_k p$ awakenings,

```
type round_t = record
    role : (winner, loser, bye, champion, dropout)
    opponent : ^Boolean
    flag : Boolean
shared rounds : array [0..P-1][0..LogP] of round_t
    // row vpid of rounds is allocated in shared memory
    // locally accessible to processor vpid
processor private sense : Boolean := true
processor private vpid : integer // a unique virtual processor index

// initially
//    rounds[i][k].flag = false for all i,k
// rounds[i][k].role =
//    winner if k > 0, i mod 2^k = 0, i + 2^(k-1) < P, and 2^k < P
//    bye if k > 0, i mod 2^k = 0, and i + 2^(k-1) >= P
//    loser if k > 0 and i mod 2^k = 2^(k-1)
//    champion if k > 0, i = 0, and 2^k >= P
//    dropout if k = 0
//    unused otherwise; value immaterial
// rounds[i][k].opponent points to
//    rounds[i-2^(k-1)][k].flag if rounds[i][k].role = loser
//    rounds[i+2^(k-1)][k].flag if rounds[i][k].role = winner or champion
//    unused otherwise; value immaterial

procedure tournament_barrier
    round : integer := 1
    loop                // arrival
        case rounds[vpid][round].role of
            loser:
                rounds[vpid][round].opponent^ := sense
                repeat until rounds[vpid][round].flag = sense
                exit loop
            winner:
                repeat until rounds[vpid][round].flag = sense
            bye:        // do nothing
            champion:
                repeat until rounds[vpid][round].flag = sense
                rounds[vpid][round].opponent^ := sense
                exit loop
            dropout:   // impossible
        round := round + 1
    loop                // wakeup
        round := round - 1
        case rounds[vpid][round].role of
            loser:      // impossible
            winner:
                rounds[vpid][round].opponent^ := sense
            bye:        // do nothing
            champion:   // impossible
            dropout:
                exit loop
    sense := not sense
```

Algorithm 10: A scalable, distributed tournament barrier with only local spinning.

and this expression is minimized for $k = 2$.[8] A processor does not examine or modify the state of any other nodes except to signal its arrival at the barrier by setting a flag in its parent's node and, when notified by its parent that the barrier has been achieved, to notify each of its children by setting a flag in each of their nodes. Each processor spins only on state information in its own tree node. To achieve a barrier, each processor executes the code shown in algorithm 11.

Data structures for the tree barrier are initialized so that each node's `parentpointer` variable points to the appropriate `childnotready` flag in the node's parent, and the `childpointers` variables point to the `parentsense` variables in each of the node's children. Child pointers of leaves and the parent pointer of the root are initialized to reference pseudo-data. The `havechild` flags indicate whether a parent has a particular child or not. Initially, and after each barrier episode, each node's `childnotready` flags are set to the value of the node's respective `havechild` flags.

Upon arrival at a barrier, a processor tests to see if the `childnotready` flag is clear for each of its children. For leaf nodes, these flags are always clear, so deadlock cannot result. After a node's associated processor sees that its `childnotready` flags are clear, it immediately re-initializes them for the next barrier. Since a node's children do not modify its `childnotready` flags again until they arrive at the next barrier, there is no potential for conflicting updates. After all of a node's children have arrived, the node's associated processor clears its `childnotready` flag in the node's parent. All processors other than the root then spin on their locally-accessible `parentsense` flag. When the root node's associated processor arrives at the barrier and notices that all of the root node's `childnotready` flags are clear, then all of the processors are waiting at the barrier. The processor at the root node toggles the `parentsense` flag in each of its children to release them from the barrier. At each level in the tree, newly released processors release all of their children before leaving the barrier, thus ensuring that all processors are eventually released. Consecutive barrier episodes do not interfere since, as described earlier, the `childnotready` flags used during arrival are re-initialized before wakeup occurs.

Our tree barrier achieves the theoretical lower bound on the number of network transactions needed to achieve a barrier on machines that lack broadcast and that distinguish between local and remote memory. At least $P - 1$ processors must signal their arrival to some other processor, requiring $P - 1$ network transactions, and must then be informed of wakeup, requiring another $P - 1$ network transactions. The length of the critical path in our algorithm is proportional to $\lceil \log_4 P \rceil + \lceil \log_2 P \rceil$. The first term is the time to propagate arrival up to the root, and the second term is the time to propagate wakeup back down to all of the leaves. On a machine with coherent caches and unlimited replication, we could replace the wakeup phase of our algorithm with a spin on a global flag. We explore this alternative on the Sequent Symmetry in section 4.4.

4 Performance Measurements

We have measured the performance of various spin lock and barrier algorithms on the BBN Butterfly 1, a distributed shared memory multiprocessor, and the Sequent Symmetry Model B, a

[8] Alex Schäffer and Paul Dietz have pointed out that better performance could be obtained in the wakeup tree by assigning more children to processors near the root. For example, if after processor a awakens processor b it takes them the same amount of time to prepare to awaken others, then we can cut the critical path roughly in half by having each processor i awaken processors $i + 2^j$, $i + 2^{j+1}$, $i + 2^{j+2}$, etc., where $j = \lfloor \log_2 i \rfloor$. Then with processor 0 acting as the root, after a serial chain of t awakenings there are 2^t processors active. Unfortunately, the obvious way to have a processor awaken more than a constant number of children involves a loop whose additional overhead partially negates the reduction in tree height. Experiments with this option (not presented in section 4.4), resulted in only about a 3% performance improvement for $p < 80$.

```
type treenode = record
    parentsense : Boolean
    parentpointer : ^Boolean
    childpointers : array [0..1] of ^Boolean
    havechild : array [0..3] of Boolean
    childnotready : array [0..3] of Boolean
    dummy : Boolean     // pseudo-data

shared nodes : array [0..P-1] of treenode
    // nodes[vpid] is allocated in shared memory
    // locally accessible to processor vpid
processor private vpid : integer     // a unique virtual processor index
processor private sense : Boolean

// on processor i, sense is initially true
// in nodes[i]:
//    havechild[j] = true if 4*i+j < P; otherwise false
//    parentpointer = &nodes[floor((i-1)/4)].childnotready[(i-1) mod 4],
//       or &dummy if i = 0
//    childpointers[0] = &nodes[2*i+1].parentsense, or &dummy if 2*i+1 >= P
//    childpointers[1] = &nodes[2*i+2].parentsense, or &dummy if 2*i+2 >= P
//    initially childnotready = havechild and parentsense = false

procedure tree_barrier
    with nodes[vpid] do
        repeat until childnotready = {false, false, false, false}
        childnotready := havechild    // prepare for next barrier
        parentpointer^ := false       // let parent know I'm ready
        // if not root, wait until my parent signals wakeup
        if vpid != 0
            repeat until parentsense = sense
        // signal children in wakeup tree
        childpointers[0]^ := sense
        childpointers[1]^ := sense
        sense := not sense
```

Algorithm 11: A scalable, distributed, tree-based barrier with only local spinning.

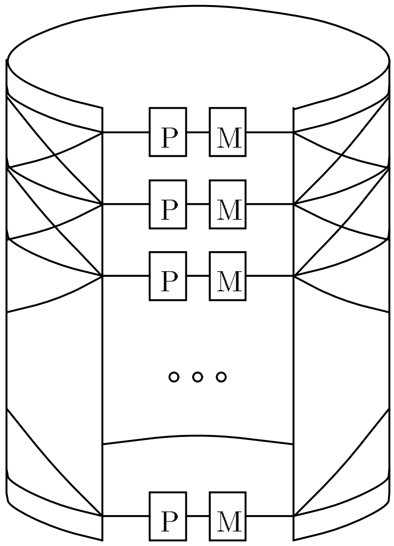

Figure 2: The BBN Butterfly 1.

cache-coherent, shared-bus multiprocessor. Anyone wishing to reproduce our results or extend our work to other machines can obtain copies of our source code (assembler for the spin locks, C for the barriers) via anonymous ftp from titan.rice.edu (directory /public/scalable_synch).

4.1 Hardware Description

BBN Butterfly

The BBN Butterfly 1 is a shared-memory multiprocessor that can support up to 256 processor nodes. Each processor node contains an 8 MHz MC68000 that uses 24-bit virtual addresses, and one to four megabytes of memory (one on our machine). Each processor can access its own memory directly, and can access the memory of any node through a $\log_4$-depth switching network (see figure 2). Transactions on the network are packet-switched and non-blocking. If collisions occur at a switch node, one transaction succeeds and all of the others are aborted, to be retried at a later time (in hardware) by the processors that initiated them. In the absence of contention, a remote memory reference (read) takes about 4 μs, roughly 5 times as long as a local reference.

The Butterfly 1 supports two 16-bit atomic operations: `fetch_and_clear_then_add` and `fetch_and_clear_then_xor`. Each operation takes three arguments: the address of the 16-bit destination operand, a 16-bit mask, and the value of the 16-bit source operand. The value of the destination operand is `and`ed with the one's complement of the mask, and then `add`ed or `xor`ed with the source operand. The resulting value replaces the original value of the destination operand. The previous value of the destination operand is the return value for the atomic operation. Using these two primitives, one can perform a variety of atomic operations, including `fetch_and_add`, `fetch_and_store` (`swap`), and `fetch_and_or` (which, like `swap`, can be used to perform a `test_and_set`).

Sequent Symmetry

The Sequent Symmetry Model B is a shared-bus multiprocessor that supports up to 30 processor nodes. Each processor node consists of a 16 MHz Intel 80386 processor equipped with a 64 KB

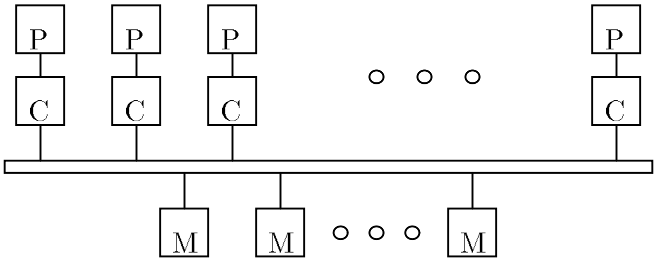

Figure 3: The Sequent Symmetry Model B.

two-way set-associative cache. All caches in the system are kept coherent by snooping on the bus (see figure 3). Each cache line is accompanied by a tag that indicates whether the data is replicated in any other cache. Writes are passed through to the bus only for replicated cache lines, invalidating all other copies. Otherwise the caches are write-back.

The Symmetry provides an atomic `fetch_and_store` operation, and allows various logical and arithmetic operations to be applied atomically as well. Each of these operations can be applied to any 1, 2, or 4 byte quantity. The logical and arithmetic operations do not return the previous value of the modified location; they merely update the value in place and set the processor's condition codes. The condition codes suffice in certain limited circumstances to determine the state of the memory prior to the atomic operation (*e.g.*, to determine when the counter has reached zero in algorithm 7), but the lack of a genuine return value generally makes the Symmetry's atomic instructions significantly less useful than the `fetch_and_`Φ operations of the Butterfly. Neither the Symmetry nor the Butterfly supports `compare_and_swap`.

4.2 Measurement Technique

Our results were obtained by embedding lock acquisitions or barrier episodes inside a loop and averaging over a large number of operations. In the spin lock graphs, each data point (P, T) represents the average time T to acquire and release the lock with P processors competing for access. On the Butterfly, the average is over 10^5 lock acquisitions. On the Symmetry, the average is over 10^6 lock acquisitions. For an individual test of P processors collectively executing K lock acquisitions, we required that each processor acquire and release the lock $\lfloor K/P \rfloor$ times. In the barrier graphs, each data point (P, T) represents the average time T for P processors to achieve a barrier. On both the Symmetry and the Butterfly, the average is over 10^5 barriers.

When $P = 1$ in the spin lock graphs, T represents the latency on one processor of the `acquire_lock` and `release_lock` operations in the absence of competition. When P is moderately large, T represents the time between successive lock acquisitions on competing processors. This *passing time* is smaller than the single-processor latency in almost all cases, because significant amounts of computation in `acquire_lock` prior to actual acquisition, and in `release_lock` after actual release, may be overlapped with work on other processors. When P is 2 or 3, T may represent either latency or passing time, depending on the relative amounts of overlapped and non-overlapped computation. In some cases (Anderson's lock, the MCS lock, and the locks incorporating backoff) the value of T for small values of P may also vary due to differences in the actual series of executed instructions, since code paths may depend on how many other processors are competing for the lock, and on which operations they have so far performed. In all cases, however, the times plotted in the spin lock graphs are the numbers that "matter" for each of the values of P. Passing time is meaningless

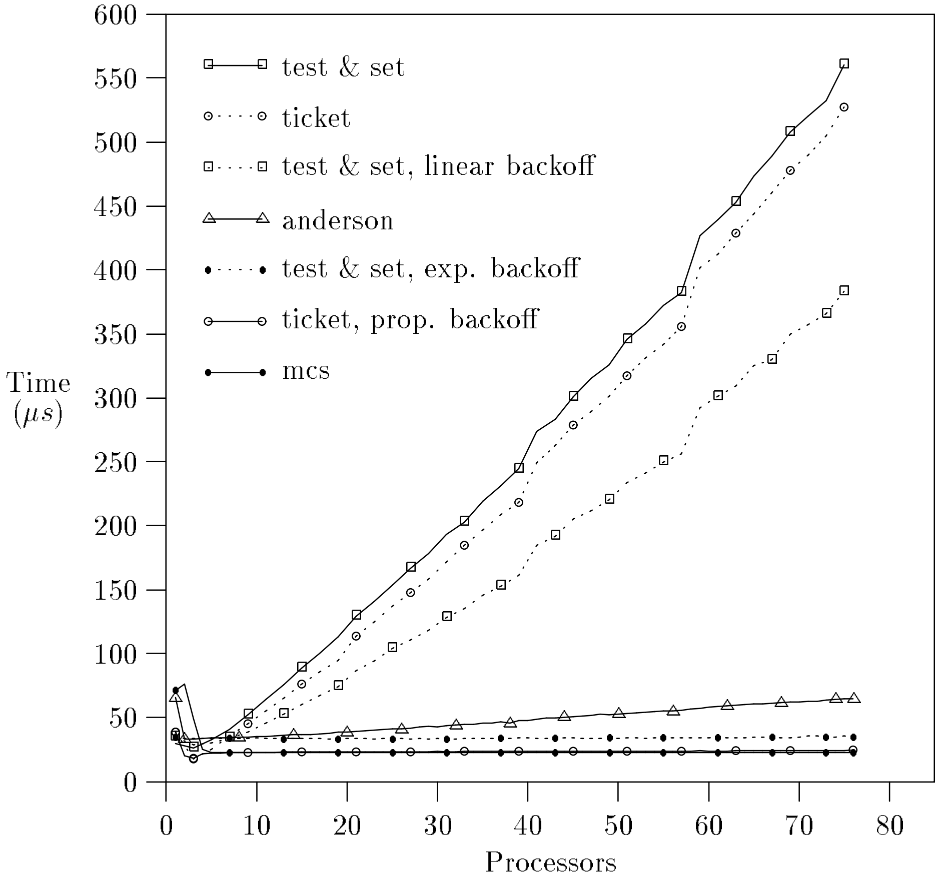

Figure 4: Performance of spin locks on the Butterfly (empty critical section).

on one processor, and latency is swamped by wait time as soon as P is moderately large.[9]

Unless otherwise noted, all measurements on the Butterfly were performed with interrupts disabled. Similarly, on the Symmetry, the `tmp_affinity()` system call was used to bind processes to processors for the duration of our experiments. These measures were taken to provide repeatable timing results.

4.3 Spin Locks

Figure 4 shows the performance on the Butterfly of the spin lock algorithms described in section 2.[10] The top curve is a simple `test_and_set` lock, which displays the poorest scaling behavior.[11] As expected, a ticket lock without backoff is slightly faster, due to polling with a read instead of a `fetch_and_`Φ. We would expect a test-and–`test_and_set` lock to perform similarly. A linear least squares regression on the timings shows that the time to acquire and release the `test_and_set`

[9]Since T may represent either latency or passing time when more than one processor is competing for a lock, it is difficult to factor out overhead due to the timing loop for timing tests with more than one processor. For consistency, we included loop overhead in all of the average times reported (both spin locks and barriers).

[10]Measurements were made for all numbers of processors; the tick marks simply differentiate between line types. Graunke and Thakkar's array-based queuing lock had not yet appeared at the time we conducted our experiments. Performance of their lock should be qualitatively identical to Anderson's lock, but with a smaller constant.

[11]We implement `test_and_set` using the hardware primitive `fetch_and_clear_then_add` with a mask that specifies to clear the lowest bit, and an addend of 1. This operation returns the old value of the lock and leaves the lowest bit in the lock set.

lock increases 7.4 μs per additional processor. The time for a ticket lock without backoff increases 7.0 μs per processor.

By analogy with the exponential backoff scheme described in section 2, we investigated the effect of having each processor delay between polling operations for a period of time directly proportional to the number of unsuccessful `test_and_set` operations. This change reduces the slope of the `test_and_set` lock graph to 5.0 μs per processor, but performance degradation is still linear in the number of competing processors.

The time to acquire and release the `test_and_set` lock, the ticket lock without backoff, and the `test_and_set` lock with linear backoff does not actually increase linearly as more processors compete for the lock, but rather in a piecewise linear fashion. This behavior is a function of the interconnection network topology and the order in which we add processors to the test. For the tests shown in figure 4, our Butterfly was configured as an 80 processor machine with 5 switch cards in the first column of the interconnection network, supporting 16 processors each. Processors were added to the test in a round robin fashion, one from each card. The breaks in the performance graph occur as each group of 20 processors is added to the test. These are the points at which we have included 4 more processors from each switch card, filling the inputs of a 4-input, 4-output switch node on each card. Adding more processors involves a set of previously unused switch nodes. What we see in the performance graphs is that involving new switch nodes causes behavior that is qualitatively different from that obtained by including another processor attached to a switch node already in use. This difference is likely related to the fact that additional processors attached to a switch node already in use add contention in the first level of the interconnection network, while involving new switch nodes adds contention in the second level of the network. In a fully configured machine with 256 processors attached to 16 switch cards in the first column of the interconnection network, we would expect the breaks in spin lock performance to occur every 64 processors.

Figure 5 provides an expanded view of performance results for the more scalable algorithms, whose curves are grouped together near the bottom of figure 4. In this expanded graph, it is apparent that the time to acquire and release the lock in the single processor case is often much larger than the the time required when multiple processors are competing for the lock. As noted above, parts of each acquire/release protocol can execute in parallel when multiple processors compete. What we are measuring in our trials with many processors is not the time to execute an acquire/release pair from start to finish, but rather the length of time between a pair of lock acquisitions. Complicating matters is that the time required to release an MCS lock depends on whether another processor is waiting.

The top curve in figure 5 shows the performance of Anderson's array-based queuing algorithm, modified to scatter the slots of the queue across the available processor nodes. This modification distributes traffic evenly in the interconnection network, by causing each processor to spin on a location in a different memory bank. Because the Butterfly lacks coherent caches, however, and because processors spin on statically unpredictable locations, it is not in general possible with the array-based queuing locks to spin on *local* locations. Linear regression yields a slope for the performance graph of 0.4 μs per processor.

Three algorithms—the `test_and_set` lock with exponential backoff, the ticket lock with proportional backoff, and the MCS lock—all scale extremely well. Ignoring the data points below 10 processors (which helps us separate throughput under heavy competition from latency under light competition), we find slopes for these graphs of 0.025, 0.021, and 0.00025 μs per processor, respectively. Since performance does not degrade appreciably for any of these locks within the range of our tests, we would expect them to perform well even with thousands of processors competing.

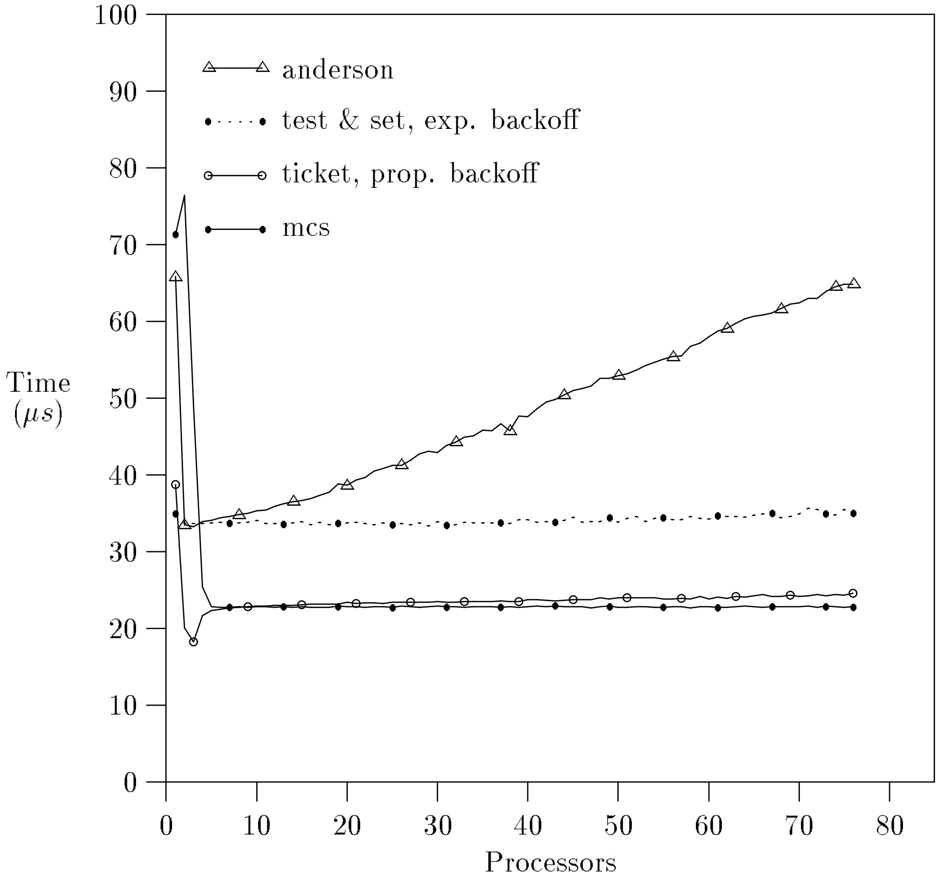

Figure 5: Performance of selected spin locks on the Butterfly (empty critical section).

Figure 6 shows performance results for several spin lock algorithms on the Symmetry. We adjusted data structures in minor ways to avoid the unnecessary invalidations that would result from placing unrelated data items in the same cache line. The test-and–`test_and_set` algorithm showed the poorest scaling behavior, with the time to acquire and release the lock increasing dramatically even over this small range of processors. A simple `test_and_set` would perform even worse.

Because the atomic add instruction on the Symmetry does not return the old value of its target location, implementation of the ticket lock is not possible on the Symmetry, nor is it possible to implement Anderson's lock directly. In his implementation [5], Anderson (who worked on a Symmetry) introduced an outer `test_and_set` lock with randomized backoff to protect the state of his queue.[12] This strategy is reasonable when the critical section protected by the outer lock (namely, acquisition or release of the inner lock) is substantially smaller than the critical section protected by the inner lock. This was not the case in our initial test, so the graph in figure 6 actually results from processors contending for the *outer* lock, instead of the inner, queue-based lock. To eliminate this anomaly, we repeated our tests with a non-empty critical section, as shown in figure 7. With a sufficiently long critical section (6.48 μs in our tests), processors have a chance to queue up on the inner lock, eliminating competition for the outer lock, and allowing the inner lock to eliminate bus transactions due to spinning. The time spent in the critical section has been factored out of the plotted timings.

[12] Given that he required the outer lock in any case, Anderson also replaced the `nextslot` index variable with a pointer, to save time on address arithmetic.

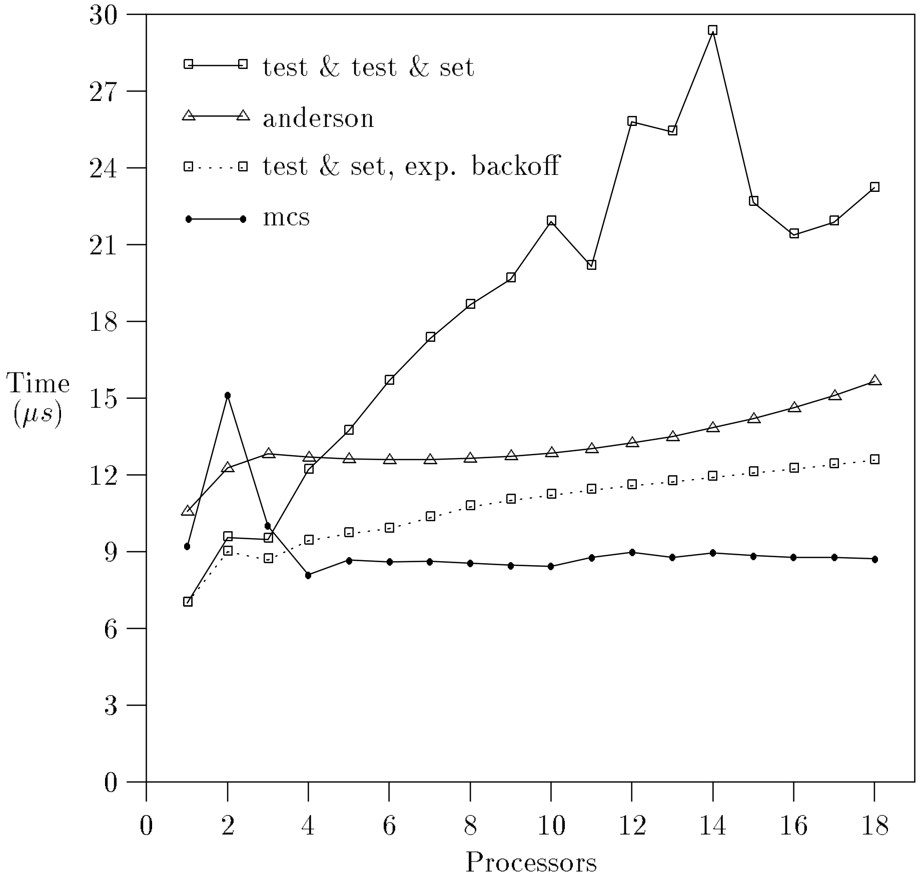

Figure 6: Performance of spin locks on the Symmetry (empty critical section).

In addition to Anderson's lock, our experiments indicate that the MCS lock and the `test_and_set` lock with exponential backoff also scale extremely well. All three of the scalable algorithms have comparable absolute performance. Anderson's lock has a small edge for non-empty critical sections (Graunke and Thakkar's lock would have a larger edge), but requires statically-allocated space per lock linear in the number of processors. The other two algorithms need only constant space, and do not require coherent caches to work well. The `test_and_set` lock with exponential backoff shows a slight increasing trend, and would not be expected to do as well as the others on a very large machine since it causes more network traffic.

The peak in the cost of the MCS lock on two processors reflects the lack of `compare_and_swap`. Some fraction of the time, a processor releasing the lock finds that its `next` variable is `nil` but then discovers that it has a successor after all when it performs its `fetch_and_store` on the lock's tail pointer. Entering this timing window necessitates an additional `fetch_and_store` to restore the state of the queue, with a consequent drop in performance. The non-empty critical sections of figure 7 reduce the likelihood of hitting the window, thereby reducing the size of the two-processor peak. With `compare_and_swap` that peak would disappear altogether.

Latency and Impact on Other Operations

In addition to performance in the presence of many competing processors, an important criterion for any lock is the time it takes to acquire and release it in the absence of competition. Table 1 shows this measure for representative locks on the Butterfly and the Symmetry. The times for the

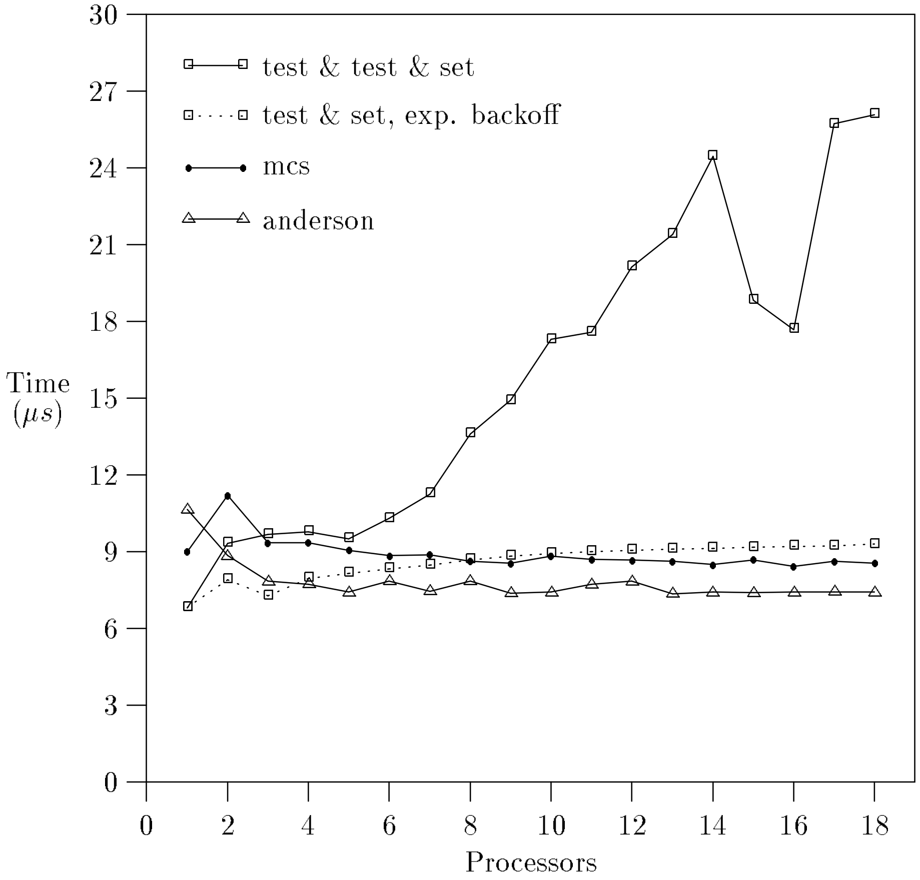

Figure 7: Performance of spin locks on the Symmetry (small critical section).

	`test_and_set`	ticket	Anderson	MCS
Butterfly	34.9 μs	38.7 μs	65.7 μs	71.3 μs
Symmetry	7.0 μs	NA	10.6 μs	9.2 μs

Table 1: Time for an acquire/release pair in the single processor case.

`test_and_set` lock are with code in place for exponential backoff; the time for the ticket lock is with code in place for proportional backoff. The `test_and_set` lock is cheapest on both machines in the single processor case; it has the shortest code path. On the Butterfly, the ticket lock, Anderson's lock, and the MCS lock are 1.11, 1.88, and 2.04 times as costly, respectively. On the Symmetry, Anderson's lock and the MCS lock are 1.51 and 1.31 times as costly as the `test_and_set` lock.

Two factors skew the absolute numbers on the Butterfly, making them somewhat misleading. First, the atomic operations on the Butterfly are inordinately expensive in comparison to their non-atomic counterparts. Most of the latency of each of the locks is due to the cost of setting up a parameter block and executing an atomic operation. (We re-use partially-initialized parameter blocks as much as possible to minimize this cost.) The expense of atomic operations affects the performance of the MCS lock in particular, since it requires at least two `fetch_and_store` operations (one to acquire the lock and another to release it), and possibly a third (if we hit the timing window). Second, the 16-bit atomic primitives on the Butterfly cannot manipulate 24-bit pointers

Busy-wait Lock	Increase in Network Latency Measured From	
	Lock Node	Idle Node
`test_and_set`	1420%	96%
`test_and_set` w/ linear backoff	882%	67%
`test_and_set` w/ exp. backoff	32%	4%
ticket	992%	97%
ticket w/ prop. backoff	53%	8%
Anderson	75%	67%
MCS	4%	2%

Table 2: Increase in network latency (relative to that of an idle machine) on the Butterfly caused by 60 processors competing for a busy-wait lock.

atomically. To implement the MCS algorithm, we were forced to replace the pointers with indices into a replicated, statically-allocated array of pointers to `qnode` records.

Absolute performance for all of the algorithms is much better on the Symmetry than on the Butterfly. The Symmetry's clock runs twice as fast, and its caches make memory in general appear significantly faster. The differences between algorithms are also smaller on the Symmetry, mainly because of the lower difference in cost between atomic and non-atomic instructions, and also because the Symmetry's 32-bit `fetch_and_store` instruction allows the MCS lock to use pointers. We believe the numbers on the Symmetry to be representative of actual lock costs on modern machines. Recent experience with the BBN TC2000 machine confirms this belief: single-processor latencies on this machine (a newer Butterfly architecture based on the Motorola 88000 chipset) vary from 8.1 μs for the simple `test_and_set` lock to 12.2 μs for the MCS lock. The single-processor latency of the MCS lock on the Symmetry is only 31% higher than that of the simple `test_and_set` lock; on the TC2000 it is 50% higher.

A final important measure of spin lock performance is the amount of interconnection network traffic caused by busy-waiting processors, and the impact of this traffic on other activity on the machine. In an attempt to measure these quantities, we obtained an estimate of average network latency on the Butterfly by measuring the total time required to probe the network interface controller on each of the processor nodes during a spin lock test. Table 2 presents our results in the form of percentage increases over the value measured on an idle machine. In the Lock Node column, probes were made from the processor on which the lock itself resided (this processor was otherwise idle in all our tests); in the Idle Node column, probes were made from another processor not participating in the spin lock test. Values in the two columns differ markedly for both the `test_and_set` and ticket locks (particularly without backoff) because competition for access to the lock is focused on a central hot spot, and steals network interface bandwidth from the process attempting to perform the latency measurement on the lock node. Values in the two columns are similar for Anderson's lock because its data structure (and hence its induced contention) is distributed throughout the machine. Values are both similar and low for the MCS lock because its data structure is distributed and because each processor refrains from spinning on the remote portions of that data structure.

Discussion and Recommendations

Spin lock algorithms can be evaluated on the basis of several criteria:

- scalability and induced network load
- one-processor latency
- space requirements
- fairness/sensitivity to preemption
- implementability with given atomic operations

The MCS lock and, on cache-coherent machines, the array-based queuing locks are the most scalable algorithms we studied. The `test_and_set` and ticket locks also scale well with appropriate backoff, but induce more network load. The `test_and_set` and ticket locks have the lowest single-processor latency, but with good implementations of `fetch_and_Φ` instructions the MCS and Anderson locks are reasonable as well. The space needs of Anderson's lock and of the Graunke and Thakkar lock are likely to be prohibitive when a large number of locks is needed—in the internals of an operating system, for example—or when locks are being acquired and released by processes significantly more numerous than the physical processors. If the number of processes can change dynamically, then the data structures of an array-based lock must be made large enough to accommodate the maximum number of processes that may ever compete simultaneously. For the Graunke and Thakkar lock, the set of process ids must also remain fairly dense. If the maximum number of processes is underestimated the system will not work; if it is overestimated then space will be needlessly wasted.

The ticket lock, the array-based queuing locks, and the MCS lock all guarantee that processors attempting to acquire a lock will succeed in FIFO order. This guarantee of fairness is likely to be considered an advantage in many environments, but is likely to waste CPU resources if spinning processes may be preempted. (Busy-wait barriers may also waste cycles in the presence of preemption.) We can avoid this problem by co-scheduling processes that share locks [34]. Alternatively (for mutual exclusion), a `test_and_set` lock with exponential backoff will allow latecomers to acquire the lock when processes that arrived earlier are not running. In this situation the `test_and_set` lock may be preferred to the FIFO alternatives. Additional mechanisms can ensure that a process is not preempted while actually holding a lock [47, 52].

All of the spin lock algorithms we have considered require some sort of `fetch_and_Φ` instructions. The `test_and_set` lock of course requires `test_and_set`. The ticket lock requires `fetch_and_increment`. The MCS lock requires `fetch_and_store`,[13] and benefits from `compare_and_swap`. Anderson's lock benefits from `fetch_and_add`. Graunke and Thakkar's lock requires `fetch_and_store`.

For cases in which competition is expected, the MCS lock is clearly the implementation of choice. It provides a very short passing time, ensures FIFO ordering, scales almost perfectly, and requires only a small, constant amount of space per lock. It also induces the least amount of interconnect contention. On a machine with fast `fetch_and_Φ` operations (particularly if `compare_and_swap` is available), its one-processor latency will be competitive with all the other algorithms.

[13] It could conceivably be used with only `compare_and_swap`, simulating `fetch_and_store` in a loop, but at a significant loss in scalability.

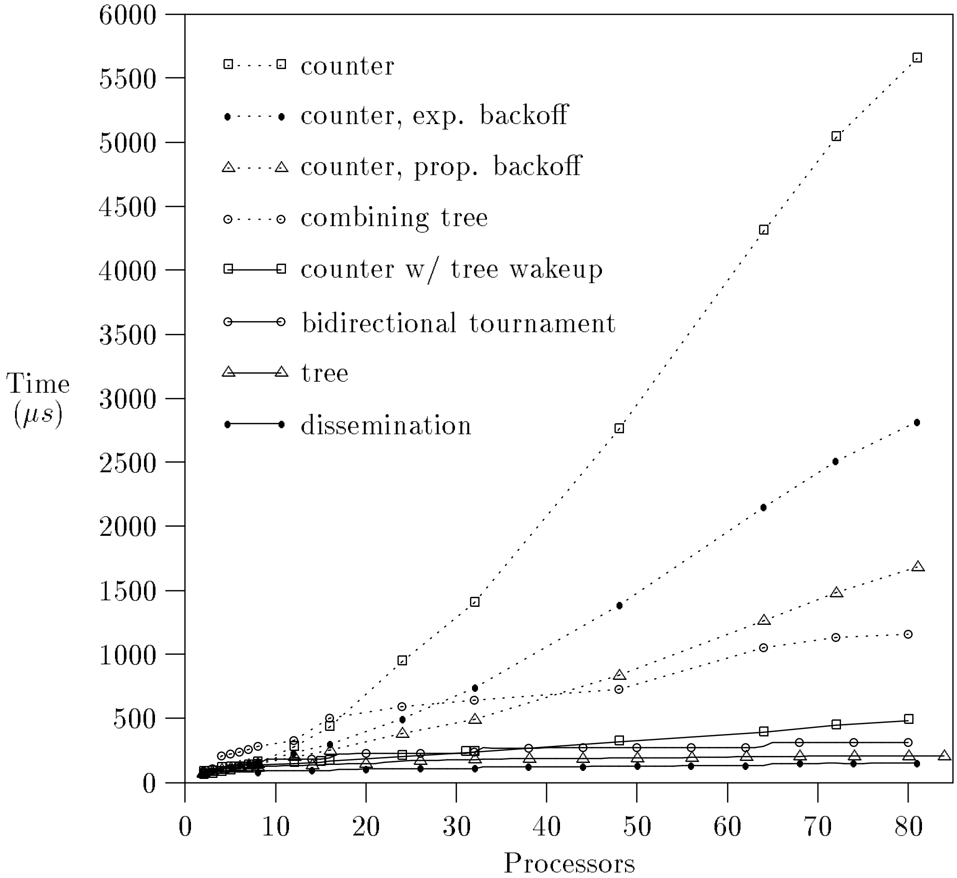

Figure 8: Performance of barriers on the Butterfly.

The ticket lock with proportional backoff is an attractive alternative if one-processor latency is an overriding concern, or if `fetch_and_store` is not available. Although our experience with it is limited to a distributed-memory multiprocessor without cache coherence, our expectation is that the ticket lock with proportional backoff would perform well on cache-coherent multiprocessors as well. The `test_and_set` lock with exponential backoff is an attractive alternative if preemption is possible while spinning, or if neither `fetch_and_store` nor `fetch_and_increment` is available. It is significantly slower than the ticket lock with proportional backoff in the presence of heavy competition, and results in significantly more network load.

4.4 Barriers

Figure 8 shows the performance on the Butterfly of the barrier algorithms described in section 3. The top three curves are all sense-reversing, counter-based barriers as in algorithm 7, with various backoff strategies. The slowest performs no backoff. The next uses exponential backoff. We obtained the best performance with an initial delay of 10 iterations through an empty loop, with a backoff base of 2. Our results suggest that it may be necessary to limit the maximum backoff in order to maintain stability. When many barriers are executed in sequence, the skew of processors arriving at the barriers appears to be magnified by the exponential backoff strategy. As the skew between arriving processors increases, processors back off farther. With a backoff base larger than 2, we had to cap the maximum delay in order for our experiments to finish. Even with a backoff base of 2, a delay cap improved performance. Our best results were obtained with a cap of $8P$

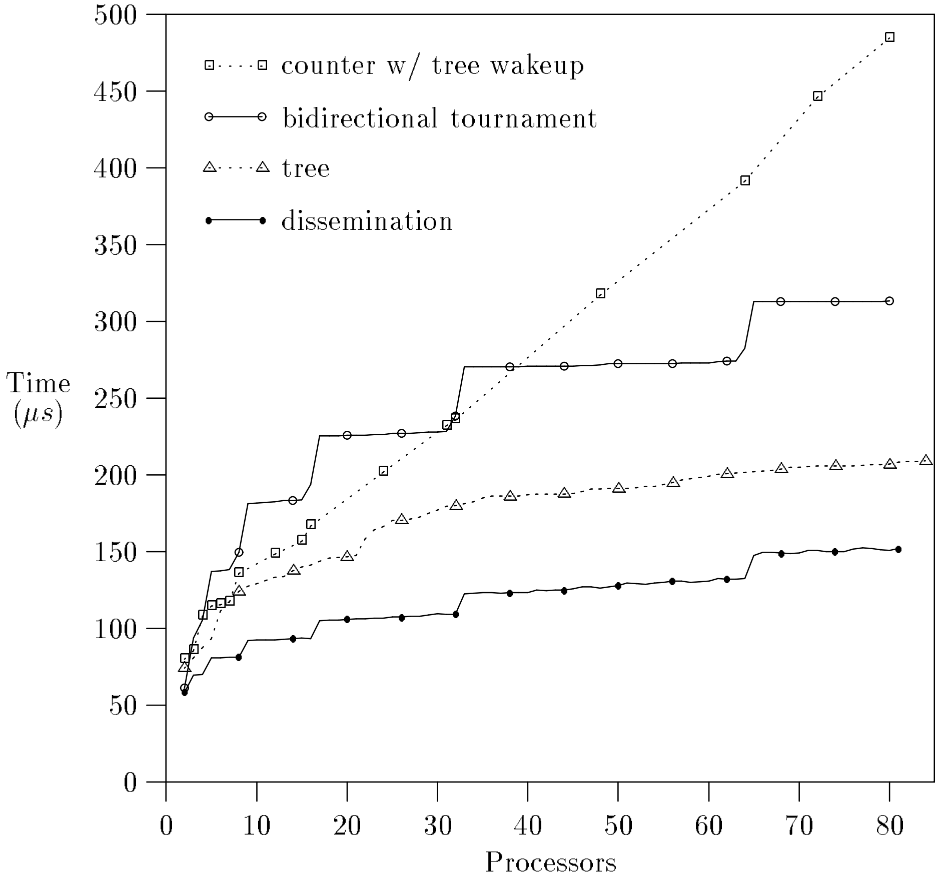

Figure 9: Performance of selected barriers on the Butterfly.

delay loop iterations.

In our final experiment with a centralized, counter-based barrier, we used a variant of the proportional delay idea employed in the ticket lock. After incrementing the barrier count to signal its arrival, each processor participating in the barrier delays a period of time proportional to the total number of participants (*not* the number yet to arrive), prior to testing the sense variable for the first time. The rationale for this strategy is to timeslice available interconnect bandwidth between the barrier participants. Since the Butterfly network does not provide hardware combining, at least $2P-1$ accesses to the barrier state are required (P to signal processor arrivals, and $P-1$ to discover that all have arrived). Each processor delays long enough for later processors to indicate their arrival, and for earlier processors to notice that all have arrived. As shown in figure 8, this strategy outperforms both the naive central barrier and the central barrier with exponential backoff. At the same time, all three counter-based algorithms lead to curves of similar shape. The time to achieve a barrier appears to increase more than linearly in the number of participants. The best of these purely centralized algorithms (the proportional backoff strategy) requires over 1.4 *ms* for an 80 processor barrier.

The fourth curve in figure 8 is the combining tree barrier of algorithm 8. Though this algorithm scales better than the centralized approaches (in fact, it scales roughly logarithmically with P, although the constant is large), it still spins on remote locations, and encounters increasing interconnect contention as the number of processors grows.

Figure 9 provides an expanded view of performance results for the algorithms with the best performance, whose curves are grouped together near the bottom of figure 8. The code for the upper curve uses a central counter to tally arrivals at the barrier, but employs a binary tree for wakeup, as in algorithm 11. The processor at the root of the tree spins on the central counter. Other processors spin on flags in their respective nodes of the tree. All spins are local, but the tallying of arrivals is serialized. We see two distinct sections in the resulting performance curve. With fewer than 10 processors, the time for wakeup dominates, and the time to achieve the barrier is roughly logarithmic in the number of participants. With more than 10 processors, the time to access the central counter dominates, and the time to achieve the barrier is roughly linear in the number of participants.

The three remaining curves in figure 9 are for the bidirectional tournament barrier, our tree barrier, and the dissemination barrier. The time to achieve a barrier with each of these algorithms scales logarithmically with the number of processors participating. The tournament and dissemination barriers proceed through $O(\lceil \log P \rceil)$ rounds of synchronization, leading to a stair-step curve. Since the Butterfly does not provide coherent caches, the tournament barrier employs a binary wakeup tree, as shown in algorithm 10. It requires $2\lceil \log_2 P \rceil$ rounds of synchronization, compared to only $\lceil \log_2 P \rceil$ rounds in the dissemination barrier, resulting in a roughly two-fold difference in performance. Our tree-based barrier lies between the other two; its critical path passes information from one processor to another approximately $\log_4 P + \log_2 P$ times. The lack of clear-cut rounds in our barrier explains its smoother performance curve: each additional processor adds another level to some path through the tree, or becomes the second child of some node in the wakeup tree, delayed slightly longer than its sibling.

Figure 10 shows the performance on the Sequent Symmetry of several different barriers. Results differ sharply from those on the Butterfly for two principal reasons. First, it is acceptable on the Symmetry for more than one processor to spin on the same location; each obtains a copy in its cache. Second, no significant advantage arises from distributing writes across the memory modules of the machine; the shared bus enforces an overall serialization. The dissemination barrier requires $O(P \log P)$ bus transactions to achieve a P-processor barrier. The other four algorithms require $O(P)$ transactions, and all perform better than the dissemination barrier for $P > 8$.

Below the maximum number of processors in our tests, the fastest barrier on the Symmetry used a centralized counter with a sense-reversing wakeup flag (from algorithm 7). P bus transactions are required to tally arrivals, 1 to toggle the sense-reversing flag (invalidating all the cached copies), and $P - 1$ to effect the subsequent re-loads. Our tree barrier generates $2P - 2$ writes to flag variables on which other processors are waiting, necessitating an additional $2P - 2$ re-loads. By using a central sense-reversing flag for wakeup (instead of the wakeup tree), we can eliminate half of this overhead. The resulting algorithm is identified as "arrival tree" in figure 10. Though the arrival tree barrier has a larger startup cost, its $P - 1$ writes are cheaper than the P read-modify-write operations of the centralized barrier, so its slope is lower. For large values of P, the arrival tree with wakeup flag is the best performing barrier, and should become clearly so on larger machines.

The tournament barrier on the Symmetry uses a central wakeup flag. It roughly matches the performance of the arrival tree barrier for $P = 2^i$, but is limited by the length of the execution path of the tournament champion, which grows suddenly by one each time that P exceeds a power of 2.

Discussion and Recommendations

Evaluation criteria for barriers include:

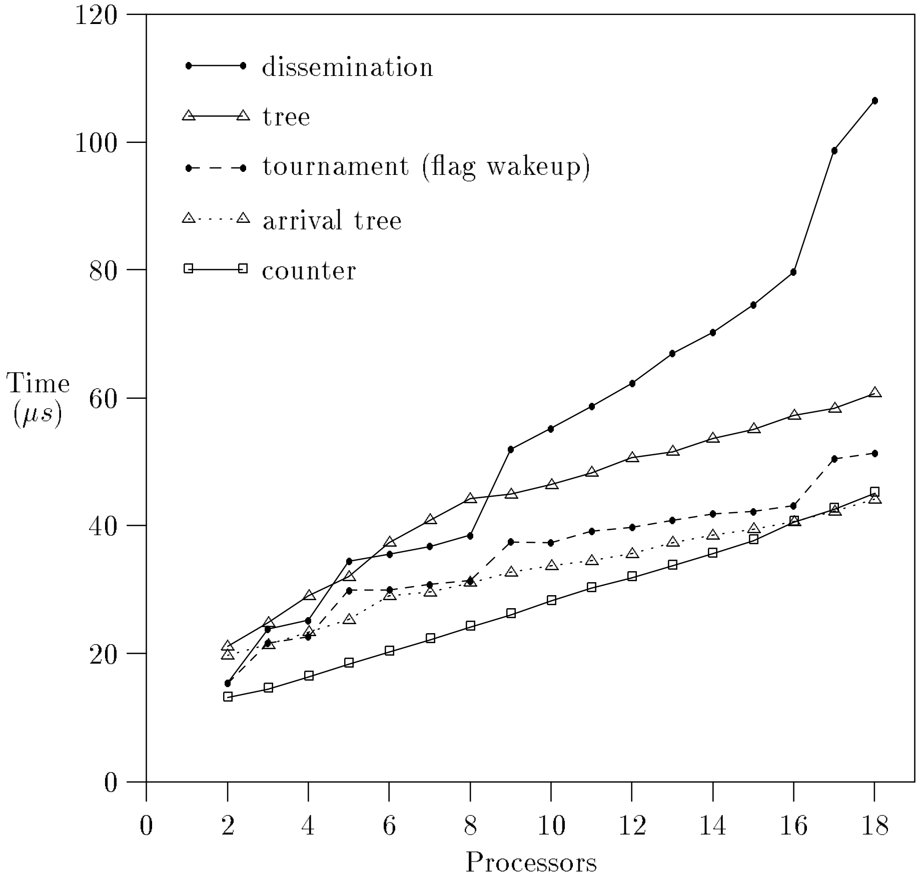

Figure 10: Performance of barriers on the Symmetry.

- length of critical path
- total number of network transactions
- space requirements
- implementability with given atomic operations

Space needs are constant for the centralized barrier, linear for our tree-based barrier and for the combining tree barrier, and $O(P \log P)$ for the dissemination barrier and for all variants of Hensgen, Finkel, and Manber's tournament barrier. (Lubachevsky's EREW version of the tournament barrier is $O(P)$.) With appropriate distribution of data structures, the dissemination barrier requires a total of $O(P \log P)$ network transactions. The tournament barrier and our tree-based barrier require $O(P)$. The centralized and combining tree barriers require $O(P)$ on machines with broadcast-based coherent caches, and a potentially unbounded number on other machines.

On a machine in which independent network transactions can proceed in parallel, the critical path length is $O(\log P)$ for all but the centralized barrier, which is $O(P)$. On a machine that serializes network transactions (*e.g.*, on a shared bus), this logarithmic factor will be dominated asymptotically by the linear (or greater) total number of network transactions. The centralized and combining tree barriers require an atomic increment or decrement instruction with at least a limited mechanism for determining the value of memory prior to modification. The other barriers depend only on the atomicity of ordinary reads and writes.

The dissemination barrier appears to be the most suitable algorithm for distributed memory machines without broadcast. It has a shorter critical path than the tree and tournament barriers (by a constant factor), and is therefore faster. The class of machines for which the dissemination barrier should outperform all other algorithms includes the BBN Butterfly [8], the IBM RP3 [37], Cedar [50], the BBN Monarch [42], the NYU Ultracomputer [17], and proposed large-scale multiprocessors with directory-based cache coherence [3]. Our tree-based barrier will also perform well on these machines. It induces less network load, and requires total space proportional to P, rather than $P \log P$, but its critical path is longer by a factor of about 1.5. It might conceivably be preferred over the dissemination barrier when sharing the processors of the machine among more than one application, if network load proves to be a problem.

Our tree-based barrier with wakeup flag should be the fastest algorithm on large-scale multiprocessors that use broadcast to maintain cache coherence (either in snoopy cache protocols [15] or in directory-based protocols with broadcast [7]). It requires only $O(P)$ updates to shared variables in order to tally arrivals, compared to $O(P \log P)$ for the dissemination barrier. Its updates are simple writes, which are cheaper than the read-modify-write operations of a centralized counter-based barrier. (Note, however, that the centralized barrier outperforms all others for modest numbers of processors). The space needs of the tree-based barrier are lower than those of the tournament barrier ($O(P)$ instead of $O(P \log P)$), its code is simpler, and it performs slightly less local work when P is not a power of 2. Our results are consistent with those of Hensgen, Finkel, and Manber [19], who showed their tournament barrier to be faster than their dissemination barrier on the Sequent Balance multiprocessor. They did not compare their algorithms against a centralized barrier because the lack of an atomic increment instruction on the Balance precludes efficient atomic update of a counter.

The centralized barrier enjoys one additional advantage over all of the other alternatives: it adapts easily to differing numbers of processors. In an application in which the number of processors participating in a barrier changes from one barrier episode to another, the log-depth barriers will all require internal reorganization, possibly swamping any performance advantage obtained in the barrier itself. Changing the number of processors in a centralized barrier entails no more than changing a single constant.

4.5 Architectural Implications

Many different shared memory architectures have been proposed. From the point of view of synchronization, the two relevant issues seem to be (1) whether each processor can access some portion of shared memory locally (instead of through the interconnection network), and (2) whether broadcast is available for cache coherency. The first issue is crucial; it determines whether busy waiting can be eliminated as a cause of memory and interconnect contention. The second issue determines whether barrier algorithms can efficiently employ a centralized flag for wakeup.

The most scalable synchronization algorithms (the MCS spin lock and the tree, bidirectional tournament, and dissemination barriers) are designed in such a way that each processor spins on statically-determined flag variable(s) on which no other processor spins. On a distributed shared memory machine, flag variables can be allocated in the portion of the shared memory co-located with the processor that spins on them. On a cache-coherent machine, they migrate to the spinning processor's cache automatically. Provided that flag variables used for busy-waiting by different processors are in separate cache lines, network transactions are used only to update a location on which some processor is waiting (or for initial cache line loads). For the MCS spin lock the number of network transactions per lock acquisition is constant. For the tree and tournament barriers, the number of network transactions per barrier is linear in the number of processors involved. For the dissemination barrier, the number of network transactions is $O(P \log P)$, but still $O(\log P)$ on the critical path. No network transactions are due to spinning, so interconnect contention is not a problem.

On "dance hall" machines, in which shared memory must always be accessed through a shared processor-memory interconnect, there is no way to eliminate synchronization-related interconnect contention in software. Nevertheless, the algorithms we have described are useful since they minimize memory contention and hot spots caused by synchronization. The structure of these algorithms makes it easy to assign each processor's busy-wait flag variables to a different memory bank so that the load induced by spinning will be distributed evenly throughout memory and the interconnect, rather than being concentrated in a single spot. Unfortunately, on dance hall machines the load will still consume interconnect bandwidth, degrading the performance not only of synchronization operations but also of all other activity on the machine, severely constraining scalability.

Dance hall machines include bus-based multiprocessors without coherent caches, and multistage network architectures such as Cedar [50], the BBN Monarch [42], and the NYU Ultracomputer [17]. Both Cedar and the Ultracomputer include processor-local memory, but only for private code and data. The Monarch provides a small amount of local memory as a "poor man's instruction cache." In none of these machines can local memory be modified remotely. We consider the lack of local shared memory to be a significant architectural shortcoming; the inability to take full advantage of techniques such as those described in this paper is a strong argument against the construction of dance hall machines.

To assess the importance of local shared memory, we used our Butterfly 1 to simulate a machine in which all shared memory is accessed through the interconnection network. By flipping a bit in the segment register for the synchronization variables on which a processor spins, we can cause the processor to go out through the network to reach these variables (even though they are in its own memory), without going through the network to reach code and private data. This trick effectively flattens the two-level shared memory hierarchy of the Butterfly into a single level organization similar to that of Cedar, the Monarch, or the Ultracomputer.

Figure 11 compares the performance of the dissemination and tree barrier algorithms for one and two level memory hierarchies. All timing measurements in the graph were made with interrupts disabled, to eliminate any effects due to timer interrupts or scheduler activity. The bottom two curves are the same as in figures 8 and 9. The top two curves show the corresponding performance of the barrier algorithms when all accesses to shared memory are forced to go through the interconnection network. When busy-waiting accesses traverse the interconnect, the time to achieve a barrier using the tree and dissemination algorithms increases linearly with the number of processors participating. A least squares fit shows the additional cost per processor to be 27.8 μs and 9.4 μs, respectively. For an 84-processor barrier, the lack of local spinning increases the cost of the tree and dissemination barriers by factors of 11.8 and 6.8, respectively.

In a related experiment, we measured the impact on network latency of executing the dissemination or tree barriers with and without local access to shared memory. The results appear in table 3. As in table 2, we probed the network interface controller on each processor to compare network latency of an idle machine with the latency observed during a 60 processor barrier. Table 2 shows that when processors are able to spin on shared locations locally, average network latency increases only slightly. With only network access to shared memory, latency more than doubles.

Studies by Pfister and Norton [38] show that hot-spot contention can lead to tree saturation in multistage interconnection networks with blocking switch nodes and distributed routing control,

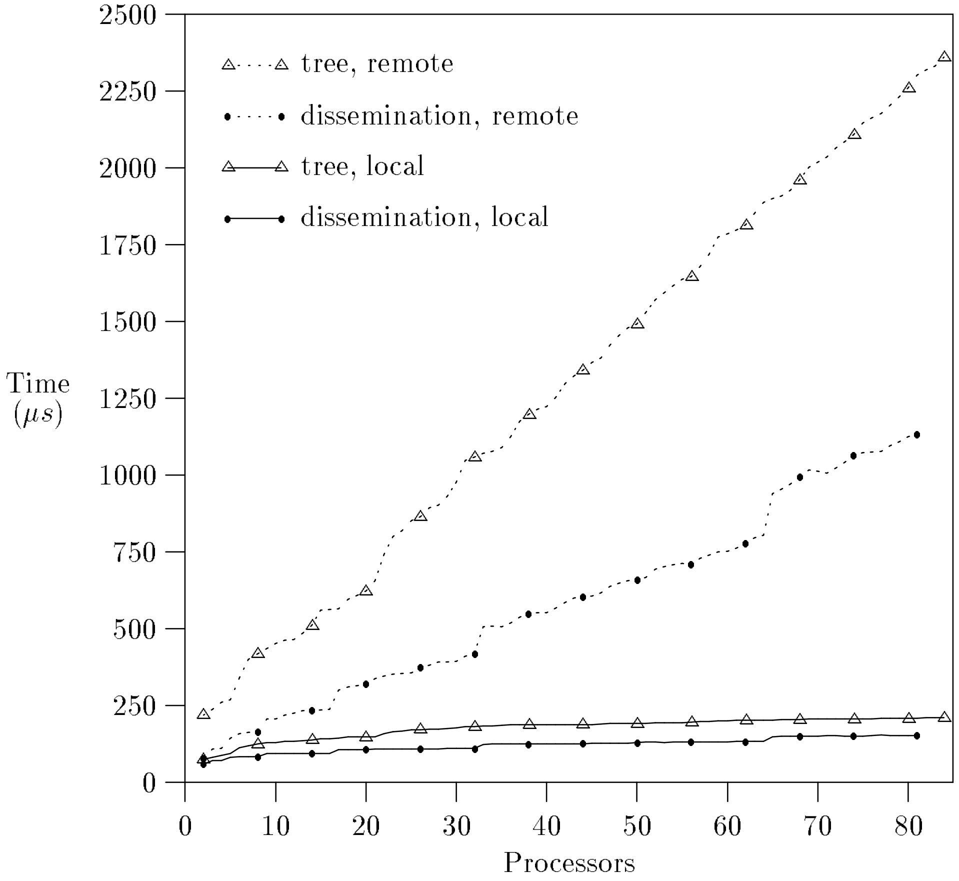

Figure 11: Performance of tree and dissemination barriers with and without local access to shared memory.

barrier	local polling	network polling
tree	10%	124%
dissemination	18%	117%

Table 3: Increase in network latency (relative to that of an idle machine) on the Butterfly caused by 60 processor barriers using local and network polling strategies.

independent of the network topology. A study by Kumar and Pfister [23] shows the onset of hot-spot contention to be rapid. Pfister and Norton argue for hardware message combining in interconnection networks to reduce the impact of hot spots. They base their argument primarily on anticipated contention for locks, noting that they know of no quantitative evidence to support or deny the value of combining for general memory traffic. Our results indicate that the cost of synchronization in a system without combining, and the impact that synchronization activity will have on overall system performance, is much less than previously thought (provided that the architecture incorporates a shared memory hierarchy of two or more levels). Although the scalable algorithms presented in this paper are unlikely to match the performance of hardware combining, they will come close enough to provide an extremely attractive alternative to complex, expensive hardware.[14]

Other researchers have suggested building special-purpose hardware mechanisms solely for synchronization, including synchronization variables in the switching nodes of multistage interconnection networks [21] and lock queuing mechanisms in the cache controllers of cache-coherent multiprocessors [14, 28, 35]. Our results suggest that simple exploitation of a multi-level memory hierarchy, *in software*, may provide a more cost-effective means of avoiding lock-based contention.

The algorithms we present in this paper require no hardware support for synchronization other than commonly-available atomic instructions. The scalable barrier algorithms rely only on atomic read and write. The MCS spin lock algorithm uses `fetch_and_store` and maybe `compare_and_swap`. Graunke and Thakkar's lock requires `fetch_and_store`. Anderson's lock benefits from `fetch_and_increment`, and the ticket lock requires it. All of these instructions have uses other than the construction of busy-wait locks. `Fetch_and_store` and `compare_and_swap`, for example, are essential for manipulating pointers to build concurrent data structures [20, 32]. Because of their general utility, `fetch_and_`Φ instructions are substantially more attractive than special-purpose synchronization primitives. Future designs for shared memory machines should include a full set of `fetch_and_`Φ operations, including `compare_and_swap`.

Our measurements on the Sequent Symmetry indicate that special-purpose synchronization mechanisms such as the QOLB instruction [14] are unlikely to outperform our MCS lock by more than 30%. A QOLB lock will have higher single-processor latency than a `test_and_set` lock [14, p.68], and its performance should be essentially the same as the MCS lock when competition for a lock is high. Goodman, Vernon, and Woest suggest that a QOLB-like mechanism can be implemented at very little incremental cost (given that they are already constructing large coherent caches with multi-dimensional snooping). We believe that this cost must be extremely low to make it worth the effort.

Of course, increasing the performance of busy-wait locks and barriers is not the only possible rationale for implementing synchronization mechanisms in hardware. Recent work on weakly-consistent shared memory [1, 13, 28] has suggested the need for synchronization "fences" that provide clean points for memory semantics. Combining networks, likewise, may improve the performance of memory with bursty access patterns (caused, for example, by sharing after a barrier). We do not claim that hardware support for synchronization is unnecessary, merely that the most commonly-cited rationale for it—that it is essential to reduce contention due to synchronization—is invalid.

[14]Pfister and Norton estimate that message combining will increase the size and possibly the cost of an interconnection network 6- to 32-fold. Gottlieb [16] indicates that combining networks are difficult to bit-slice.

5 Summary of Recommendations

We have presented a detailed comparison of new and existing algorithms for busy-wait synchronization on shared-memory multiprocessors, with a particular eye toward minimizing the network transactions that lead to contention. We introduced the MCS lock, the new tree-based barrier, and the notion of proportional backoff for the ticket lock and the centralized barrier. We demonstrated how to eliminate `fetch_and_Φ` operations from the wakeup phase of the combining tree barrier, presented a wakeup mechanism for the tournament barrier that uses contiguous, statically allocated flags to ensure local-only spinning, and observed that the data structures of the dissemination barrier can be distributed for local-only spinning.

The principal conclusion of our work is that memory and interconnect contention due to busy-wait synchronization in shared-memory multiprocessors need not be a problem. This conclusion runs counter to widely-held beliefs. We have presented empirical performance results for a wide variety of busy-wait algorithms on both a cache-coherent multiprocessor and a multiprocessor with distributed shared memory. These results demonstrate that appropriate algorithms using simple and widely-available atomic instructions can reduce synchronization contention effectively to zero.

For spin locks on a shared-memory multiprocessor, regardless of architectural details, we suggest:

1. If the hardware provides an efficient `fetch_and_store` instruction (and maybe `compare_and_swap`), then use the MCS lock. One-processor latency will be reasonable, and scalability will be excellent.

2. If `fetch_and_store` is not available, or if atomic operations are very expensive relative to non-atomic instructions and one-processor latency is an overwhelming concern, then use the ticket lock with proportional backoff (assuming the hardware supports `fetch_and_increment`). The code for such a lock is typically more complicated than code for the MCS lock, and the load on the processor-memory interconnect will be higher in the presence of competition for the lock, but speed on a single processor will be slightly better and scalability will still be reasonable.

3. Use the simple lock with exponential backoff (with a cap on the maximum delay) if processes might be preempted while spinning, or if one-processor latency is an overwhelming concern and the hardware does not support `fetch_and_increment` (assuming of course that it *does* support `test_and_set`).

For barrier synchronization we suggest:

1. On a broadcast-based cache-coherent multiprocessor (with unlimited replication), use either a centralized counter-based barrier (for modest numbers of processors), or a barrier based on our 4-ary arrival tree and a central sense-reversing wakeup flag.

2. On a multiprocessor without coherent caches, or with directory-based coherency without broadcast, use either the dissemination barrier (with data structures distributed to respect locality) or our tree-based barrier with tree wakeup. The critical path through the dissemination barrier algorithm is about a third shorter than that of the tree barrier, but the total amount of interconnect traffic is $O(P \log P)$ instead of $O(P)$. The dissemination barrier will outperform the tree barrier on machines such as the Butterfly, which allow non-interfering network transactions from many different processors to proceed in parallel.

For the designers of large-scale shared-memory multiprocessors, our results argue in favor of providing distributed memory or coherent caches, rather than dance-hall memory without coherent caches (as in Cedar, the Monarch, or the Ultracomputer). Our results also indicate that combining networks for such machines must be justified on grounds other than the reduction of synchronization overhead. We strongly suggest that future multiprocessors include a full set of `fetch_and_Φ` operations (especially `fetch_and_store` and `compare_and_swap`).

Acknowledgments

Tom LeBlanc and Evangelos Markatos provided helpful comments on an early version of this paper. Comments of the referees were also very helpful.

Some of the experiments described in this paper were performed on a BBN TC2000 that is part of the Advanced Computing Research Facility, Mathematics and Computer Science Division, Argonne National Laboratory.

References

[1] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *Proc. of the International Symposium on Computer Architecture*, pages 2–14, Seattle, WA, May 1990.

[2] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proc. of the International Symposium on Computer Architecture*, pages 396–406, Jerusalem, Israel, May 1989.

[3] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. of the International Symposium on Computer Architecture*, pages 280–289, New York, NY, June 1988.

[4] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, CA, 1989.

[5] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[6] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *Performance Evaluation Review*, 17(1):49–60, May 1989. SIGMETRICS '89 Conference Paper.

[7] J. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In *Proc. of the International Symposium on Computer Architecture*, pages 355–362, 1984.

[8] BBN Laboratories. Butterfly parallel processor overview. Technical Report 6148, Version 1, BBN Laboratories, Cambridge, MA, Mar. 1986.

[9] E. D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[10] E. D. Brooks III. The shared memory hypercube. *Parallel Computing*, 6:235–245, 1988.

[11] H. Davis and J. Hennessy. Characterizing the synchronization behavior of parallel programs. In *Proc. of the ACM/SIGPLAN PPEALS 1988 Conference on Parallel Programming: Experience with Applications, Languages and Systems*, pages 198–211, New Haven, CT, July 1988.

[12] E. Dijkstra. Solution of a problem in concurrent programming and control. *Communications of the ACM*, 8(9):569, Sept. 1965.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the International Symposium on Computer Architecture*, pages 15–26, Seattle, WA, May 1990.

[14] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Boston, MA, Apr. 1989.

[15] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A new large-scale cache coherent multiprocessor. In *Proc. of the International Symposium on Computer Architecture*, pages 422–431, Honolulu, HI, May 1988.

[16] A. Gottlieb. Scalability, Combining and the NYU Ultracomputer. Ohio State University Parallel Computing Workshop, Mar. 1990. Invited Lecture.

[17] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer—Designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, Feb. 1983.

[18] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, June 1990.

[19] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[20] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 197–206, Seattle, WA, Mar. 1990.

[21] D. N. Jayasimha. Distributed synchronizers. In *Proc. of the 1988 International Conference on Parallel Processing*, pages 23–27, St. Charles, IL, Aug. 1988.

[22] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Proc. of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 218–228, 1986.

[23] M. Kumar and G. F. Pfister. The onset of hot spot contention. In *Proc. of the 1986 International Conference on Parallel Processing*, pages 28–34, 1986.

[24] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, Aug. 1974.

[25] L. Lamport. The mutual exclusion problem: Part I—A theory of interprocess communication; Part II—Statement and solutions. *Journal of the ACM*, 33(2):313–348, Apr. 1986.

[26] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, Feb. 1987.

[27] C. A. Lee. Barrier synchronization over multistage interconnection networks. In *Proc. of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 130–133, Dallas, TX, Dec. 1990.

[28] J. Lee and U. Ramachandran. Synchronization with multiprocessor caches. In *Proc. of the International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[29] B. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. I. *Acta Informatica*, pages 125–169, 1984.

[30] B. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. In *Proc. of the 1989 International Conference on Parallel Processing*, pages II–175–II–179, Aug. 1989.

[31] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[32] J. M. Mellor-Crummey. Concurrent queues: Practical fetch-and-Φ algorithms. Technical Report 229, Computer Science Department, University of Rochester, Nov. 1987.

[33] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. Technical Report 342, Computer Science Department, University of Rochester, Apr. 1990. Also COMP TR90-114, Department of Computer Science, Rice University, May 1990.

[34] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2), Feb. 1980.

[35] P1596 Working Group of the IEEE Computer Society Microprocessor Standards Committee. SCI (scalable coherent interface): An overview of extended cache-coherence protocols, Feb. 5, 1990. Draft 0.59 P1596/Part III-D.

[36] G. L. Peterson. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Transactions on Programming Languages and Systems*, 5(1):56–65, Jan. 1983.

[37] G. Pfister et al. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proc. of the 1985 International Conference on Parallel Processing*, pages 764–771, St. Charles, Illinois, Aug. 1985.

[38] G. F. Pfister and V. A. Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, Oct. 1985.

[39] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, Feb. 1989.

[40] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press Series in Scientific Computation. MIT Press, Cambridge, MA, 1986. Translated from the French by D. Beeson.

[41] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, Feb. 1979.

[42] R. D. Rettberg, W. R. Crowther, P. P. Carvey, and R. S. Tomlinson. The Monarch parallel processor hardware design. *Computer*, 23(4):18–30, Apr. 1990.

[43] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, Jan. 1981.

[44] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proc. of the International Symposium on Computer Architecture*, pages 340–347, 1984.

[45] B. A. Sanders. The information structure of distributed mutual exclusion algorithms. *ACM Transactions on Computer Systems*, 5(3):284–299, Aug. 1987.

[46] F. B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):179–195, Apr. 1982.

[47] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Multi-model parallel programming in Psyche. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 70–78, Seattle, WA, Mar. 1990.

[48] P. Tang and P.-C. Yew. Processor self-scheduling for multiple-nested parallel loops. In *Proc. of the 1986 International Conference on Parallel Processing*, pages 528–535, St. Charles, IL, Aug. 1986.

[49] P. Tang and P.-C. Yew. Algorithms for distributing hot-spot addressing. CSRD report 617, Center for Supercomputing Research and Development, University of Illinois Urbana-Champaign, Jan. 1987.

[50] P.-C. Yew. Architecture of the Cedar parallel supercomputer. CSRD report 609, Center for Supercomputing Research and Development, University of Illinois Urbana-Champaign, Aug. 1986.

[51] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, Apr. 1987.

[52] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel processors. Technical Report TR-89-07-03, Computer Science Department, University of Washington, July 1989.