Synchronization Without Contention

John M. Mellor-Crummey*
(johnmc@rice.edu)
Center for Research on Parallel Computation
Rice University, P.O. Box 1892
Houston, TX 77251-1892

Michael L. Scott†
(scott@cs.rochester.edu)
Computer Science Department
University of Rochester
Rochester, NY 14627

Abstract

Conventional wisdom holds that contention due to busy-wait synchronization is a major obstacle to scalability and acceptable performance in large shared-memory multiprocessors. We argue the contrary, and present fast, simple algorithms for contention-free mutual exclusion, reader-writer control, and barrier synchronization. These algorithms, based on widely available `fetch_and_`Φ instructions, exploit local access to shared memory to avoid contention. We compare our algorithms to previous approaches in both qualitative and quantitative terms, presenting their performance on the Sequent Symmetry and BBN Butterfly multiprocessors. Our results highlight the importance of local access to shared memory, provide a case against the construction of so-called "dance hall" machines, and suggest that special-purpose hardware support for synchronization is unlikely to be cost effective on machines with sequentially consistent memory.

1 Introduction

Busy-waiting synchronization, including locks and barriers, is fundamental to parallel programming on shared-memory multiprocessors. Busy waiting 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 in the implementation of an operating system kernel where scheduler-based blocking is inappropriate or impossible.

Because busy-wait mechanisms are often used to protect or separate very small critical sections, their performance is a matter of paramount importance. Unfortunately, typical implementations of busy waiting tend to produce large amounts of memory and interconnection network contention, introducing performance bottlenecks that become markedly more pronounced in larger machines and applications. When many processors busy-wait on a single synchronization variable, they create a *hot spot* that gets a disproportionate share of the network traffic. Pfister and Norton [22] 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. Agarwal and Cherian [2] found that references to synchronization variables on cache-coherent multiprocessors cause cache line invalidations much more often than non-synchronization references. They also observed that synchronization accounted for as much as 49% of network traffic in simulations of a 64-processor "dance hall" machine, in which each access to a shared variable traverses the processor-memory interconnection network.

Pfister and Norton [22] argue for message combining in multistage interconnection networks. They base their argument primarily on anticipated hot-spot contention for locks, noting that they know of no quantitative evidence to support or deny the value of combining for general memory traffic. (Hardware combining appears in the original designs for the NYU UltraComputer [10], the IBM RP3 [21], and the BBN Monarch [24].) Other researchers have suggested adding special-purpose hardware solely for synchronization, including synchronization variables in the switching nodes of multistage interconnection networks [14] and lock queuing mechanisms in the cache controllers of cache-coherent multiprocessors [9, 20]. The principal purpose of these mechanisms is to reduce contention caused by busy waiting. Before adopting them, it is worth considering the extent to which software techniques can achieve a similar effect.

Our results indicate that the cost of synchronization in systems without combining, and the impact that synchronization activity will have on overall system performance, is much less than previously thought. This paper describes novel algorithms that use distributed data structures to implement protocols for busy-wait synchronization. All that our algorithms require in the way of hardware support is a

*Supported in part by the National Science Foundation under Cooperative Agreement number CCR-9045252.

†Supported in part by the National Science Foundation under Institutional Infrastructure grant CDA-8822724.

simple set of fetch_and_Φ operations[1] and a memory hierarchy in which each processor is able to read some portion of shared memory without using the interconnection network. By allocating data structures appropriately, we ensure that each processor busy waits only on *locally-accessible* variables on which no other processor spins, thus eliminating hot-spot contention caused by busy waiting. On a machine in which shared memory is distributed (*e.g.*, the BBN Butterfly [4], the IBM RP3 [21], or a shared-memory hypercube [6]), a processor busy waits only on flag variables located in its local portion of the shared memory. On a machine with coherent caches, a processor busy waits only on flag variables resident in its cache; flag variables for different processors are placed in different cache lines, to eliminate false sharing.

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 present scalable algorithms for mutual exclusion spin locks, reader-writer spin locks, and multi-processor barriers in section 2. We relate these algorithms to previous work in section 3, and present performance results in section 4 for both a distributed shared-memory multiprocessor (the BBN Butterfly 1), and a machine with coherent caches (the Sequent Symmetry Model B). Our architectural conclusions are summarized in section 5.

2 Scalable Synchronization

In this section we present novel busy-wait algorithms for mutual exclusion, reader-writer control, and barrier synchronization. All are designed to use local-only spinning. The reader-writer lock requires atomic fetch_and_store (xmem) [3] and compare_and_swap[4] instructions. a specified 'new' value is written into the memory location.) The mutual-exclusion lock requires fetch_and_store, and benefits from the availability of compare_and_swap. The barrier requires nothing more than the usual atomicity of memory reads and writes.

Our pseudo-code notation is meant to be more-or-less self explanatory. Line breaks terminate statements (except in obvious cases of run-on), and indentation indicates nesting in control constructs. The keyword shared indicates that a declared variable is to be shared by all processors. This 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.

[1] A fetch_and_Φ operation [15] reads, modifies, and writes a memory location atomically. Common fetch_and_Φ operations include test_and_set, fetch_and_store (swap), fetch_and_add, and compare_and_swap.

[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.

[3] fetch_and_store(x,new)≡ old := ^x; ^x := new; return old

[4] compare_and_swap(x,old,new) ≡
cc := (^x = old); if cc then ^x := new; fi; return cc

2.1 Mutual Exclusion

Our spin lock algorithm (called the MCS lock, after our initials):

- 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 MCS lock (algorithm 1) was inspired by the QOSB (Queue On Synch Bit) hardware primitive proposed for the cache controllers of the Wisconsin Multicube [9], but is implemented entirely in software. Like QOSB, our lock algorithm maintains a queue of processors requesting a lock; this organization enables each processor to busy wait on a unique, locally-accessible flag variable.

```
type qnode = record
    next : ^qnode     // ptr to successor in queue
    locked : Boolean  // busy-waiting necessary
type lock = ^qnode    // ptr to tail of queue

// I points to a queue link record allocated
// (in an enclosing scope) in shared memory
// locally-accessible to the invoking processor

procedure acquire_lock(L : ^lock; I : ^qnode)
    var pred : ^qnode
    I->next := nil        // initially, no successor
    pred := fetch_and_store(L, I) // queue for lock
    if pred != nil        // lock was not free
        I->locked := true // prepare to spin
        pred->next := I   // link behind predecessor
        repeat while I->locked // busy-wait for lock

procedure release_lock(L : ^lock; I : ^qnode)
    if I->next = nil      // no known successor
        if compare_and_swap(L, I, nil)
            return        // no successor, lock free
        repeat while I->next = nil // wait for succ.
    I->next->locked := false       // pass lock
```

Algorithm 1: A queue-based spin lock with local-only spinning.

To use our locking algorithm, a process allocates a locally-accessible record containing a link pointer and a Boolean flag in a scope enclosing the calls to the locking primitives (each processor can use a single statically-allocated record if lock acquisitions are never nested). One additional private temporary variable is employed during the acquire_lock operation. Processors holding or waiting for the same lock are chained together by the links in the local records. Each processor

spins on its own local flag. The lock itself contains a pointer to the record for the processor at the tail of the queue, or `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 `pred->next` in `acquire_lock`. Both spins are local.

Alternative code for the `release_lock` operation, without `compare_and_swap`, appears in algorithm 2. Like the code in

```
procedure release_lock(L : ^lock; I : ^qnode)
    var old_tail : ^qnode
    if I->next = nil      // no known successor
        old_tail := fetch_and_store(L, nil)
        if old_tail = I   // really no successor
            return
        // We accidentally removed some processors
        // from the queue and have to put them back.
        usurper := fetch_and_store(L, old_tail)
        // wait for pointer to victim list
        repeat while I->next = nil
        if usurper != nil
            // usurper in queue ahead of our victims
            // link victims after the last usurper
            usurper->next := I->next
        else // pass lock to first victim
            I->next->locked := false
    else // pass lock to successor
        I->next->locked := false
```

Algorithm 2: Code for `release_lock`, without `compare_and_swap`.

algorithm 1, it spins on locally-accessible, processor-specific memory locations only, requires constant space per lock, and requires only $O(1)$ network transactions regardless of whether the machine provides coherent caches. It does not guarantee FIFO ordering, however, and admits the theoretical possibility of starvation, though lock acquisitions are likely to remain very nearly FIFO in practice. A correctness proof for the MCS lock appears in a technical report [18].

2.2 Reader-Writer Control

The MCS spin lock algorithm can be modified to grant concurrent access to readers without affecting any of the lock's fundamental properties. As with traditional semaphore-based approaches to reader-writer synchronization [7], we can design versions that provide fair access to both readers and writers, or that grant preferential access to one or the other. We present a fair version here. A read request is granted when all previous write requests have completed. A write request is granted when all previous read and write requests have completed.

Code for the reader-writer lock appears in algorithm 3. As in the MCS spin lock, we maintain a linked list of requesting processors. In this case, however, we allow a requester to read and write fields in the link record of its predecessor (if any). To ensure that the record is still valid (and has not been deallocated or reused), we require that a processor access its predecessor's record *before* initializing the record's `next` pointer. At the same time, we force every processor that has a successor to wait for its `next` pointer to become non-`nil`, even if the pointer will never be used. As in the MCS lock, the existence of a successor is determined by examining `L->tail`.

A reader can begin reading if its predecessor is a reader that is already active, but it must first unblock its successor (if any) if that successor is a waiting reader. To ensure that a reader is never left blocked while its predecessor is reading, each reader uses `compare_and_swap` to *atomically* test if its predecessor is an active reader, and if not, notify its predecessor that it has a waiting reader as a successor.

Similarly, a writer can proceed if its predecessor is done and there are no active readers. A writer whose precedessor is a writer can proceed as soon as its predecessor is done, as in the MCS lock. A writer whose predecessor is a reader must go through an additional protocol using a count of active readers, since some readers that started earlier may still be active. When the last reader of a group finishes (`reader_count = 0`), it must resume the writer (if any) next in line for access. This may require a reader to resume a writer that is not its direct successor. When a writer is next in line for access, we write its name in a global location. We use `fetch_and_store` to read and erase this location atomically, ensuring that a writer proceeds on its own if and only if no reader is going to try to resume it. To make sure that `reader_count` never reaches zero prematurely, we increment it *before* resuming a blocked reader, and before updating the `next` pointer of a reader whose reading successor proceeds on its own.

```
type qnode = record
    class : (reading, writing)
    next : ^qnode
    state : record
        blocked : Boolean    // need to spin
        successor_class : (none, reader, writer)
type lock = record
    tail : ^qnode := nil
    reader_count : integer := 0
    next_writer : ^qnode := nil

// I points to a queue link record allocated
// (in an enclosing scope) in shared memory
// locally-accessible to the invoking processor

procedure start_write(L : ^lock; I : ^qnode)
    with I^, L^
        class := writing; next := nil;
        state := [true, none]
        pred : ^qnode := fetch_and_store(&tail, I)
        if pred = nil
            next_writer := I
            if reader_count = 0 and
                  fetch_and_store(&next_writer,nil)=I
                // no reader who will resume me
                blocked := false
        else
            // must update successor_class before
            // updating next
            pred->successor_class := writer
            pred->next := I
        repeat while blocked
```

```
procedure start_read(L : ^lock; I : ^qnode)
    with I^, L^
        class := reading; next := nil
        state := [true, none]
        pred : ^qnode := fetch_and_store(&tail, I)
        if pred = nil
            atomic_increment(reader_count)
            blocked := false  // for successor
        else
            if pred->class = writing or
                    compare_and_swap(&pred->state,
                        [true, none], [true, reader])
                // pred is a writer, or a waiting
                // reader. pred will increment
                // reader_count and release me
                pred->next := I
                repeat while blocked
            else
                // increment reader_count and go
                atomic_increment(reader_count)
                pred->next := I
                blocked := false
        if successor_class = reader
            repeat while next = nil
            atomic_increment(reader_count)
            next->blocked := false

procedure end_write(L: ^lock; I : ^qnode)
    with I^, L^
        if next != nil or not
                compare_and_swap(&tail, I, nil)
            // wait until succ inspects my state
            repeat while next = nil
            if next->class = reading
                atomic_increment(reader_count)
            next->blocked := false

procedure end_read(L : ^lock; I : ^qnode)
    with I^, L^
        if next != nil or not
                compare_and_swap(&tail, I, nil)
            // wait until succ inspects my state
            repeat while next = nil
            if successor_class = writer
                next_writer := next
        if fetch_and_decrement(reader_count) = 1
            // I'm last reader, wake writer if any
            w : ^qnode :=
                fetch_and_store(&next_writer, nil)
            if w != nil
                w->blocked := false
```

Algorithm 3: A fair reader-writer lock with local-only spinning.

2.3 Barrier Synchronization

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.[5] 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 flag variables in its own tree node. To achieve a barrier, each processor executes the code shown in algorithm 4.

Our tree barrier achieves the theoretical lower bound on the number of network transactions needed to achieve a barrier on machines that lack broadcast. 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 of cache lines, we could replace the wakeup phase of our algorithm with a busy wait on a global flag. We explore this alternative on the Sequent Symmetry in section 4.2.

3 Related Work

In [18] we survey existing spin lock and barrier algorithms, examining their differences in detail and in many cases presenting improvements over previous-published versions. Many of these algorithms appear in the performance results of section 4; we describe them briefly here.

3.1 Spin locks

The simplest algorithm for mutual exclusion repeatedly polls a Boolean flag with a `test_and_set` instruction, attempting to acquire the lock by changing the flag from false to true. A processor releases the lock by setting it to false. Protocols based on the so-called test-and-`test_and_set` [25] reduce memory and interconnection network contention by polling with read requests, issuing a `test_and_set` only when the lock appears to be free. Such protocols eliminate contention on cache-coherent machines *while the lock is held* but with

[5] We use a fan-out of 2 because it results in the the shortest critical path required to resume P spinning processors for a tree of uniform degree. We use a fan-in of 4 (1) because it produced the best performance in Yew, Tzeng, and Lawrie's experiments [27] with a related technique they call 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. Alex Schäffer and Paul Dietz have pointed out that slightly better performance might be obtained in the wakeup tree by assigning more children to processors near the root.

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

processor private vpid : integer
    // a unique ''virtual processor'' index
processor private sense : Boolean

shared nodes : array [0..P-1] of treenode
    // nodes[vpid] is allocated in shared memory
    // locally-accessible to processor vpid

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

procedure tree_barrier
    with nodes[vpid] do
        repeat until cnotready =
            [false, false, false, false]
        cnotready := havechild  // init for next time
        parentpointer^ := false // signal parent
        if vpid != 0
            // not root, wait until parent wakes me
            repeat until wsense = sense
        // signal children in wakeup tree
        childpointers[0]^ := sense
        childpointers[1]^ := sense
        sense := not sense
```

Algorithm 4: A scalable, distributed, tree-based barrier with only local spinning.

P competing processors can still induce $O(P)$ network traffic (as opposed to $O(1)$ for the MCS lock) each time the lock is freed. Alternatively, a `test_and_set` lock can be designed to pause between its polling operations. Anderson [3] found exponential backoff to be the most effective form of delay; our experiments confirm this result.

A "ticket lock" [18] (analogous to a busy-wait implementation of a semaphore constructed using an eventcount and a sequencer [23]) consists of a pair of counters, one containing the number of requests to acquire the lock, and the other the number of times the lock has been released. A process acquires the lock by performing a `fetch_and_increment` 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. The ticket lock ensures that processors acquire the lock in FIFO order, and reduces the number of `fetch_and_`Φ operations per lock acquisition. Contention is still a problem, but the counting inherent in the lock allows us to introduce a very effective form of backoff, in which each processor waits for a period of time proportional to the difference between the values of its ticket and the release counter.

In an attempt to eliminate contention entirely, Anderson [3] has proposed a locking algorithm that requires only a constant number of memory references through the interconnection network per lock acquisition/release on cache-coherent multiprocessors. The trick is for each processor to use `fetch_and_increment` 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 algorithm outperforms a `test_and_set` lock with exponential backoff when contention for the lock is high [3]. A similar lock has been devised by Graunke and Thakkar [11]. Unfortunately, neither lock generalizes to multiprocessors without coherent caches, and both require statically-allocated space *per lock* linear in the number of processors.

3.2 Barriers

In the simplest barrier algorithms (still widely used in practice), each processor increments a centralized shared count as it reaches the barrier, and spins until that count (or a flag set by the last arriving processor) indicates that all processors are present. Like the simple `test_and_set` spin lock, such centralized barriers induce enormous amounts of contention. Moreover, Agarwal and Cherian [2] found backoff schemes to be of limited use for large numbers of processors. Our results (see section 4) confirm this conclusion.

In a study of what they call "software combining," Yew, Tzeng, and Lawrie [27] note that their technique could be used to implement a barrier. They organize processors into groups of k, for some k, and each group is assigned to a unique leaf of a k-ary tree. Each processor performs a `fetch_and_increment` on the count in its group's leaf node; the last processor to arrive at a node continues upwards, incrementing a count in the node's parent. Continuing in this fashion, a single processor reaches the root of the tree and initiates a reverse wave of wakeup operations. Straightforward application of software combining suggests the use of `fetch_and_decrement` for wakeup, but we observe in [18] that simple reads and writes suffice. A combining tree barrier still spins on non-local locations, but causes less contention than a centralized counter barrier, since at most $k-1$ processors share any individual variable.

Hensgen, Finkel, and Manber [12] and Lubachevsky [17] have devised tree-style "tournament" barriers. In their algorithms, processors start at the leaves of a binary tree. One processor from each node continues up the tree to the next "round" of the tournament. The "winning" processor is statically determined; there is no need for `fetch_and_`Φ. In Hensgen, Finkel, and Manber's tournament barrier, and in one of two tournament barriers proposed by Lubachevsky, processors spin for wakeup on a single global flag, set by the tournament champion. This technique works well on cache-coherent machines with broadcast and unlimited cache line replication, but does not work well on distributed-memory machines. Lubachevsky presents a second tournament barrier that employs a tree for wakeup as well as arrival; this

barrier will work well on cache-coherent machines that limit cache line replication, but does not permit local-only spinning on distributed-memory machines. In [18] we present a bidirectional tournament barrier (based on Hensgen, Finkel, and Manber's algorithm) that spins only on locally-accessible variables, even on distributed-memory machines. As shown in section 4, however, the resulting barrier is slightly slower than algorithm 4. The latter employs a squatter tree, with processors assigned to both internal and external nodes.

To the best of our knowledge, only one barrier other than our tree or modified tournament achieves the goal of local-only spinning on machines without coherent caches (though its authors did not address this issue). Hensgen, Finkel, and Manber [12] describe a "dissemination barrier" based on their own algorithm for disseminating information in a distributed system, and on an earlier "butterfly barrier" of Brooks [5]. Processors in a butterfly or dissemination barrier engage in a series of symmetric, pairwise synchronizations, with no distinguished latecomer or champion. The dissemination barrier performs $O(P \log P)$ accesses across the interconnection network, but only $O(\log P)$ on its critical path. If these accesses contend for a serialized resource such as a central processor-memory bus, then a tree or tournament barrier, with only $O(P)$ network accesses, would be expected to out-perform it. However, on machines in which multiple independent network accesses can proceed in parallel, our experiments indicate that the dissemination barrier outperforms its competitors by a small constant factor.

4 Empirical Performance Results

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 cache-coherent, shared-bus multiprocessor. We were unable to test our reader-writer lock on either of these machines because of the lack of a `compare_and_swap` instruction, though recent experiments on a BBN TC2000 [19] confirm that it scales extremely well. The MCS lock was implemented using the alternative version of `release_lock` from algorithm 2. Anyone wishing to reproduce our results or extend our work to other machines can obtain copies of our C and assembler source code via anonymous ftp from titan.rice.edu (/public/scalable_synch).

Our results were obtained by embedding a lock acquisition/release pair or a barrier episode 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 for an individual processor to acquire and release the lock once, with P processors competing for access. In the barrier graphs, points represent the average time required for P processors to achieve a single barrier. Reported values include the overhead of an iteration of the test loop. To eliminate any effects due to timer interrupts or scheduler activity, on the Butterfly all timing measurements were made with interrupts disabled; on the Symmetry, the `tmp_affinity` system call was used to bind processes to processors.

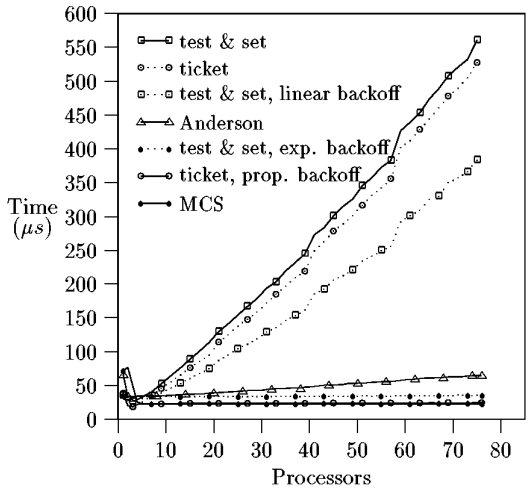

Figure 1: Performance of spin locks on the Butterfly.

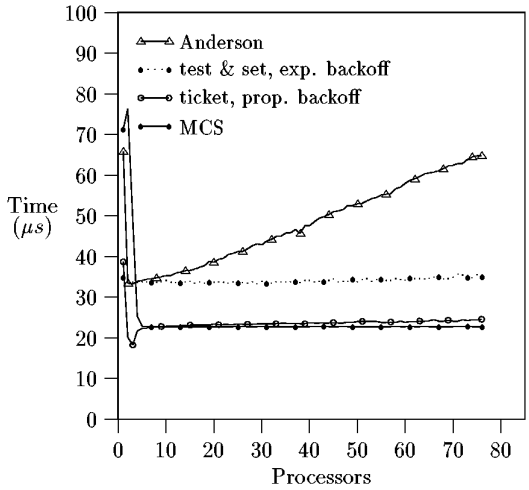

Figure 2: Performance of selected spin locks on the Butterfly.

4.1 Spin locks

Figure 1 shows the performance on the Butterfly of several spin lock algorithms. (Measurements were made for all numbers of processors; the tick marks simply differentiate between line types.) Figure 2 provides an expanded view of the four best performers, and figure 3 shows analogous results on the Symmetry. Since the Symmetry lacks an atomic `fetch_and_increment` instruction, a ticket lock cannot be readily implemented and Anderson's lock is implemented as he suggests [3], by protecting its data structures with an outer `test_and_set` lock. So long as Anderson's lock is used for critical sections whose length exceeds that of the operations on its own data structures, processors do not seriously compete for the outer lock. In the absence of coherent caches, we modified Anderson's algorithm on the Butterfly to scatter the slots of its array across the available processor nodes. This modification reduces the impact of contention by spreading

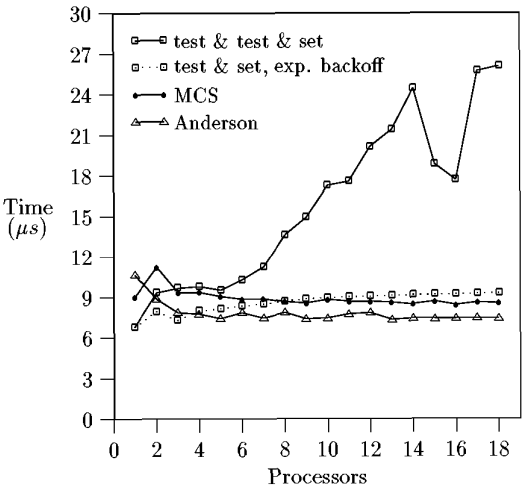

Figure 3: Performance of spin locks on the Symmetry.

it throughout the interconnection network and memory.

The top two curves in figure 1 are for simple `test_and_set` and ticket locks without backoff. They scale extremely badly. Most of the concern in the literature over synchronization-induced contention appears to have been based on the behavior represented by the top curve in this graph. Linear backoff in a `test_and_set` lock doesn't help much, but exponential backoff for the `test_and_set` lock and proportional backoff for the ticket lock lead to much better scaling behavior. The MCS lock scales best of all. Figure 2 suggests that these three algorithms would continue to perform well even with thousands of processors competing. The slope of the curve for the MCS lock is 0.00025 μs per processor.

In many of the curves, the time to acquire and release a lock in the single processor case is significantly larger than the time required when multiple processors are competing for the lock. The principal reason for this apparent anomaly is that 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 consecutive lock acquisitions on separate processors. Complicating matters is that the time required to release an MCS lock depends on whether another processor is waiting.

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 longer critical sections on the Symmetry (introduced to be fair to Anderson's lock) 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.

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

Several factors skew the absolute numbers on the Butterfly, making them somewhat misleading. First, the `test_and_set` lock is implemented completely in line, while the others use subroutine calls. Second, the atomic operations on the Butterfly are inordinately expensive in comparison to their non-atomic counterparts. Third, those operations take 16-bit operands, and cannot be used to manipulate pointers directly. We believe the numbers on the Symmetry to be more representative of actual lock costs on modern machines; our recent experience on the TC2000 confirms this belief. We note that the single-processor latency of the MCS lock on the Symmetry is only 31% higher than that of the simple `test_and_set` lock.

In an attempt to uncover contention not evident in the spin lock timing measurements, 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 1 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 are very different for 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.

4.2 Barriers

Figure 4 shows the performance on the Butterfly of several

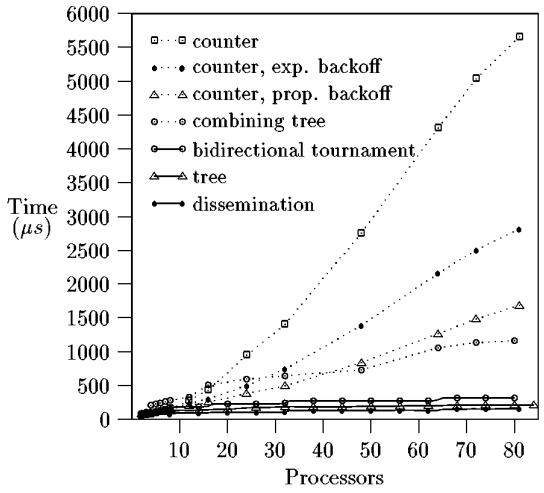

Figure 4: Performance of barriers on the Butterfly.

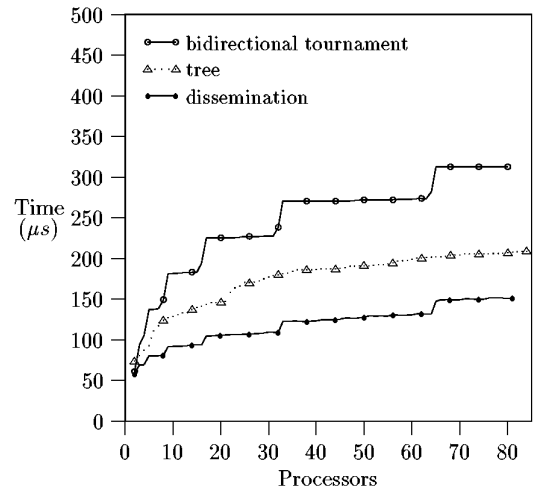

Figure 5: Performance of selected barriers on the Butterfly.

barrier algorithms. The upper three curves represent centralized, counter-based barriers. The top one uses no backoff. The next uses exponential backoff. The third backs off in proportion to the total number of processors in the barrier, with the hope that doing so will not only allow the remaining processors to reach the barrier, but also allow processors that have already arrived to notice that the barrier has been achieved. At a minimum, these three barriers require time linear in the number of participating processors. The combining tree barrier [27] (modified to avoid `fetch_and_`Φ operations during wakeup [18]) appears to perform logarithmically, but it still spins on remote locations, and has a large enough constant to take it out of the running.

Timings for the three most scalable algorithms on the Butterfly appear in expanded form in figure 5. The dissemination barrier [12] performs best, followed by our tree barrier (algorithm 4), and by Hensgen, Finkel, and Manber's tournament barrier [12] (with our code [18] for wakeup on machines

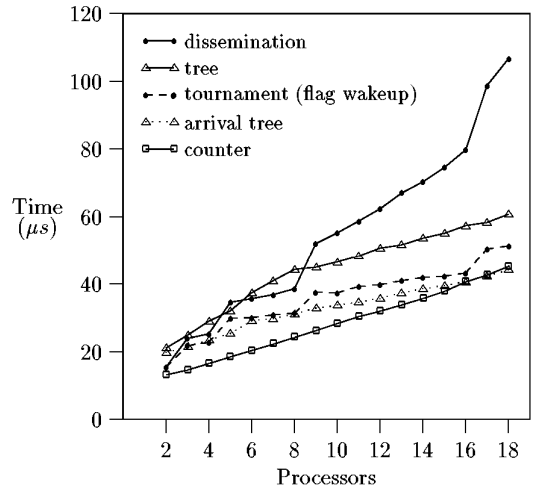

Figure 6: Performance of barriers on the Symmetry.

lacking broadcast—see section 3). All three of these barriers scale logarithmically in the number of participating processors, with reasonable constants.

The comparative performance of barriers on the Symmetry is heavily influenced by the presence of coherent caches. Figure 6 presents results for the dissemination, tree, and tournament barriers (this time with the original wakeup flag for the tournament), together with two other algorithms that capitalize on the caches. One is a simple centralized barrier; the other combines the arrival phase of our tree barrier with a central wakeup flag. Since the bus-based architecture of the Symmetry serializes all network accesses, each algorithm's scaling behavior is ultimately determined by the number of network accesses it requires. The dissemination barrier requires $O(P \log P)$; each of the other algorithms requires only $O(P)$. Our experiments show that the arrival tree outperforms even the counter-based barrier for more than 16 processors; its $O(P)$ writes are cheaper than the latter's $O(P)$ `fetch_and_`Φ operations.

4.3 The Importance of Local Shared Memory

Because our software techniques can eliminate contention by spinning on locally-accessible locations, they argue in favor of architectures in which shared memory is physically distributed or coherently cached. On "dance hall" machines, in which shared memory must always be accessed through the processor-memory interconnect, we see no way to eliminate synchronization-related contention in software (except perhaps with a very high latency mechanism based on interprocessor messages or interrupts). All that any algorithm can do on such a machine is reduce the impact of hot spots, by distributing load throughout the memory and interconnection network.

Dance hall machines include bus-based multiprocessors without coherent caches, and multistage interconnection net-

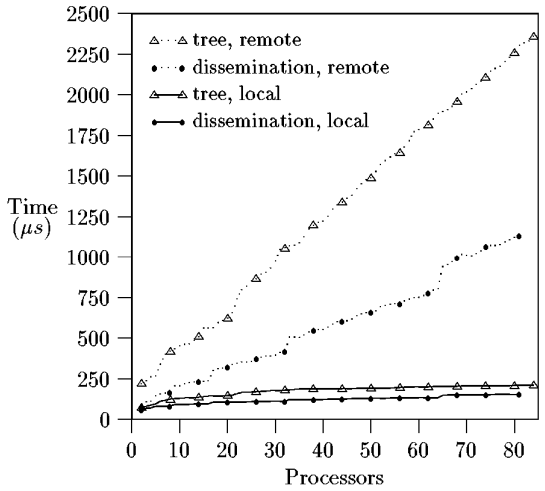

Figure 7: Performance of tree and dissemination barriers on the Butterfly with and without local access to shared memory.

work architectures such as Cedar [26], the BBN Monarch [24], and the NYU Ultracomputer [10]. 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 quantify 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 7 compares the performance of the dissemination and tree barrier algorithms for one and two level memory hierarchies. The bottom two curves are the same as in figures 4 and 5. 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: the time to achieve a barrier increases linearly with the number of processors participating.

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 4.3. As in table 1, 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 1 shows that when processors

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

Table 2: 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.

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.

5 Discussion and Conclusions

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 that exploit locality in a machine's memory hierarchy can virtually eliminate synchronization-related contention.

Although the scalable algorithms presented in this paper are unlikely to match the synchronization performance of a combining network, they will come close enough to provide an extremely attractive alternative to complex, expensive hardware.[6] All that our algorithms require is locally-accessible shared memory and common atomic instructions. The scalable barrier algorithms rely only on atomic read and write. The MCS lock uses `fetch_and_store` and (if available) `compare_and_swap`. Each of these instructions is useful for more than busy-wait locks. Herlihy has shown, for example [13], that `compare_and_swap` is a *universal* primitive for building non-blocking concurrent data structures. Because of their general utility, `fetch_and_`Φ instructions are substantially more attractive to the programmer than special-purpose synchronization primitives.

Our measurements on the Sequent Symmetry indicate that special-purpose synchronization mechanisms such as the QOSB instruction [9] are unlikely to outperform our MCS lock by more than 30%. A QOSB lock will have higher single-processor latency than a `test_and_set` lock [9, 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 QOSB-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 implement-

[6] Pfister and Norton [22] estimate that message combining will increase the size and/or cost of an interconnection network 6- to 32-fold.

ing synchronization mechanisms in hardware. Recent work on weakly-consistent shared memory [1, 8, 16] 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.

For future 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`).

References

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

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

[3] 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.

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

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

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

[7] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with 'readers' and 'writers'. *Communications of the ACM*, 14(10):667–668, Oct. 1971.

[8] 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 *Proceedings of the International Symposium on Computer Architecture*, pages 15–26, May 1990.

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

[10] 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.

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

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

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

[14] D. N. Jayasimha. Distributed synchronizers. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 23 27, Aug. 1988.

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

[16] J. Lee and U. Ramachandran. Synchronization with multiprocessor caches. In *Proceedings of the International Symposium on Computer Architecture*, pages 27–37, May 1990.

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

[18] 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.

[19] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, Apr. 1991.

[20] 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.

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

[22] 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.

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

[24] 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.

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

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

[27] 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.