Fast, Contention-Free Combining Tree Barriers for Shared-Memory Multiprocessors

Michael L. Scott
Computer Science Department
University of Rochester
Rochester, NY 14627–0226
scott@cs.rochester.edu

John M. Mellor-Crummey
Computer Science Department
Rice University, P.O. Box 1892
Houston, TX 77251
johnmc@cs.rice.edu

Abstract

In a previous article [5], Gupta and Hill introduced an *adaptive combining tree* algorithm for busy-wait barrier synchronization on shared-memory multiprocessors. The intent of the algorithm was to achieve a barrier in logarithmic time when processes arrive simultaneously, and in constant time after the last arrival when arrival times are skewed. A *fuzzy* [6] version of the algorithm allows a process to perform useful work between the point at which it notifies other processes of its arrival and the point at which it waits for all other processes to arrive.

Unfortunately, adaptive combining tree barriers as originally devised perform a large amount of work at each node of the tree, including the acquisition and release of locks. They also perform an unbounded number of accesses to non-local locations, inducing large amounts of memory and interconnect contention. We present new adaptive combining tree barriers that eliminate these problems. We compare the performance of the new algorithms to that of other fast barriers on a 64-node BBN Butterfly 1 multiprocessor, a 35-node BBN TC2000, and a 126-node KSR 1. The results reveal scenarios in which our algorithms outperform all known alternatives, and suggest that both adaptation and the combination of fuzziness with tree-style synchronization will be of increasing importance on future generations of shared-memory multiprocessors.

Key words: synchronization, scalability, fuzzy barriers, adaptive combining trees.

At the University of Rochester, this work was supported in part by NSF Institutional Infrastructure grant number CDA-8822724 and ONR research contract number N00014-92-J-1801 (in conjunction with the ARPA Research in Information Science and Technology---High Performance Computing, Software Science and Technology program, ARPA Order No. 8930). At Rice University, this work was supported in part by NSF Cooperative Agreements CCR-8809615 and CCR-9120008.

International Journal of Parallel Programming, Vol. 22, No. 4, 1994.

1. Introduction

A *barrier* is a synchronization mechanism that ensures that no process advances beyond a particular point in a computation until all processes have arrived at that point. Barriers are widely used to delimit algorithmic phases; they 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 $t+1$. If phases are brief (as they are in many applications), barrier overhead may be a major contributor to execution time; fast barrier implementations are thus of great importance. This paper focuses on busy-wait (spinning) barrier implementations for shared-memory multiprocessors.

In the simplest barrier algorithms, each process increments a shared, centralized counter as it reaches a barrier, and spins until that counter (or a flag set by the last arriving process) indicates that all processes are present. Such centralized algorithms suffer from several limitations:

linear asymptotic latency

On a machine without hardware combining of atomic instructions, achieving a barrier requires time linear in the number of processes, P. Specifically, it requires a sequence of $O(P)$ updates to the central counter followed (in the absence of broadcast or fast multicast) by $O(P)$ reads.

contention

Because processes access a central location, traditional centralized barriers can generate contention for memory and for the processor-memory interconnection network. Such contention degrades the performance of any process that initiates references involving the network or a saturated memory bank.

unnecessary waiting

Processes that arrive at a barrier early (to announce to their peers that they have completed some critical computation) must wait for their peers to arrive as well, even if they have other work they could be doing that does not depend on the arrival of those peers.

To improve asymptotic latency, several barriers have been developed that run in time $O(\log P)$. Most use some form of tree to gather and scatter information [7, 13, 15, 19]; the butterfly and dissemination barriers of Brooks [2] and of Hensgen, Finkel, and Manber [7] use a symmetric pattern of synchronization operations that resembles an FFT or parallel prefix computation. The butterfly and dissemination barriers perform a total of $O(P \log P)$ writes to shared locations, but only $O(\log P)$ on their critical paths. The various tree-based barriers perform a total of $O(P)$ writes to shared locations, with $O(\log P)$ on their critical paths. On most machines, logarithmic barriers can be designed to eliminate contention by having processes spin only on locally-accessible locations (either in a local coherent cache, or in a local portion of shared memory) [15].

To reduce unnecessary waiting at barriers, Gupta introduced the notion of a *fuzzy barrier* [6]. Such a barrier consists of two distinct phases. In the first phase, processes announce that they have completed all the work on which their peers depend. In the second phase they wait until all their peers have made similar announcements. A traditional centralized barrier can be modified trivially to implement these two phases as separate `enter_barrier` and `exit_barrier` routines. Unfortunately, none of the logarithmic barriers mentioned above has such an obvious fuzzy version. In the butterfly and dissemination barriers, no process knows that all other processes have arrived until the very end of the algorithm. In a static tree barrier [15], and in the tournament barriers of Hensgen, Finkel, and Manber [7] and Lubachevsky [13], static synchronization orderings force some processes to wait for their peers before announcing that they have reached the barrier. In all of the tree-based barriers, processes waiting near the leaves cannot discover that the barrier has been achieved until processes higher in the tree have already noticed this fact.

Logarithmic barriers also introduce an additional problem:

lack of amortization

The critical path requires $O(\log P)$ writes to shared locations *after the arrival of the last process* before any process can continue. In a traditional centralized barrier, the last arriving process discovers that the barrier has been achieved in constant time (ignoring possible delay due to contention).

To address this problem, Gupta and Hill [5] introduced the concept of an *adaptive combining tree* barrier. Each process arriving at an adaptive combining tree barrier performs a local modification to the tree that allows later arrivals to start their work closer to the root. Given sufficient skew in the arrival times of processes, the last arriving process performs only a constant amount of work before discovering that the barrier has been achieved. To address unnecessary waiting, Gupta and Hill also devised a fuzzy version of their algorithm, with a separate tree traversal for the wakeup phase of the barrier. Both the regular and fuzzy versions use per-tree-node locks to ensure that updates to the structure of the tree are viewed consistently by all processes. Unfortunately, the scheme for maintaining consistency requires that processes spin on non-local locations, and can be expected to lead to contention.

Our contribution is to demonstrate that adaptivity and fuzziness can be achieved without non-local spinning, and that the resulting algorithms are of practical utility.

We review Gupta and Hill's adaptive combining tree barrier in section 2, providing fixes for several bugs in the fuzzy version of their algorithm. We present new algorithms in section 3, and examine their performance in section 4. Our algorithms avoid locking tree nodes by performing their updates in an asynchronous, wait-free fashion. They also spin only on locally-accessible locations. In our performance experiments, we consider the new algorithms, Gupta and Hill's algorithms, and the fastest previously-known centralized and logarithmic barriers, using two of BBN's NUMA[1] machines (the Butterfly 1 and the TC2000), and Kendall Square's cache-coherent KSR 1. Our results indicate that fuzziness is valuable on all three machines, and that adaptation pays off on the KSR 1, where the cost of a remote operation is comparatively high. There are scenarios in which the new algorithms outperform all known alternatives on the TC2000 and the KSR 1, and architectural trends suggest that their relative performance will increase on future machines. Section 5 reviews these conclusions, and provides recommendations for practitioners and architects.

2. Previous Algorithms

Gupta and Hill's adaptive combining tree barrier appears in figure 2. The algorithm employs two instances of the barrier data structure for use in alternating barrier episodes. An initialization routine (not shown) establishes each data structure as a binary tree of nodes, with one leaf for every process. The `reinitialize` routine (called but not shown) restores the `left`, `right`, `parent`, `visited`, and `notify` fields of a node to their original values.

To take part in a barrier episode, a process starts at its leaf and proceeds upward, stopping at the first node (w) that has not been visited by any other process.[2] It then modifies the tree (see figure 1) so that w's other child (o, the child through which the process did not climb) is one level closer to the root. Specifically, the process changes o's parent to be p (the parent of w), and

---

[1] Non-Uniform Memory Access. NUMA machines have shared memory — a single physical address space — but the memory is distributed among the nodes of the machine, and is not coherently cached.

[2] A program that wishes to change the set of processes that are to take part in a given barrier episode must modify the barrier's data structures accordingly. The complexity of these modifications is a weakness shared by all of the logarithmic time barriers.

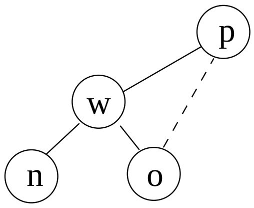

Figure 1: Naming of nodes in the adaptive combining tree barrier.

makes *o* a child of *p*. A process that reaches *p* through *w*'s sibling will promote *o* another level, and a later-arriving process, climbing through *o*, will traverse fewer levels of the tree than it would have otherwise.

A process that finds that its leaf has a nil parent knows that it is the last arrival, and can commence a wave of wakeups. It sets the `notify` flag in the root of the tree. The process waiting at the root then sets the `notify` flags in the root's children, and so on. Each process on its way out of the tree reinitializes its leaf and the node at which it waited. Two instances of the barrier data structure are required to ensure that no process can get to the next barrier episode and see nodes that have not yet been reinitialized in the wake of the previous episode.

The key to the correctness of Gupta and Hill's algorithm is its synchrony: no two processes ever see changes to the tree in an inconsistent order. In the initial loop, for example, one might think that a process that finds that *w* has already been visited could simply proceed to *w*'s parent. Allowing it to do so, however, would mean that a process might discover that the barrier has been achieved while some of its peers are still adapting nodes farther down in the tree. These adaptations could then interfere with node reinitializations during wakeup. In a similar vein, the lock on *o* in the second (''`adapt tree`'') loop ensures that *o*'s pointer to *p* and *p*'s pointer to *o* are changed mutually and atomically. Both loops release any node (*w* or *o*) that is found to have been visited already, in the knowledge that some other process will replace the pointer to it with a pointer to some unvisited node. There is no guarantee how quickly the replacement will occur, however, so there is no bound on the number of times that either loop may execute.

```
type node = record
    lock : volatile syncvar := free
        // Volatile fields are changed only by a process that holds the lock,
        // and are always changed consistently.
    visited : volatile (no, left, right) := no
        // Has this node been visited, and if so, from which child?
    root, bin_left, bin_right, bin_parent : ^node := // tree
    left, right, parent : volatile ^node := bin_left, bin_right, bin_parent
    notify : volatile Boolean := false
        // Notify will eventually become true once the barrier is achieved.

type instance = record
    my_leaf : ^node := // as appropriate, in tree

private instances : array [0..1] of instance
private current_instance : ^instance := &instances[0]
```

```
procedure barrier ()
    n : ˆnode := current_instanceˆ.my_leaf
    loop
        w : ˆnode := nˆ.parent
        if w = nil                      // We are the last arrival.
            nˆ.rootˆ.notify := true
            reinitialize (n)
            goto rtn;
        acquire (wˆ.lock)
        if wˆ.visited = no
            exit loop
        release (wˆ.lock)
        // Continue; n’s parent pointer will eventually change,
        // and we’ll get a new w.
    reinitialize (n)
    if wˆ.left = n
        wˆ.visited := left
    else
        wˆ.visited := right
    release (wˆ.lock)

    // adapt tree:
    loop
        if wˆ.visited = left
            o : ˆnode := wˆ.right
        else
            o : ˆnode := wˆ.left
        acquire (oˆ.lock)
        if oˆ.visited = no
            exit loop
        release (oˆ.lock)
        // Continue; w’s right or left pointer will eventually change,
        // and we’ll get a new o.
    p : ˆnode := wˆ.parent
    if p <> nil
        if pˆ.left = w
            pˆ.left := o
        else
            pˆ.right := o
        // This change to pˆ.left or pˆ.right may allow another process
        // to break out of the second loop above.
    oˆ.parent := p
        // This may allow another process to break out of the first loop above.
    release (oˆ.lock)

    // wait for barrier to be achieved:
    repeat until wˆ.notify              // spin
        // Rootˆ.notify is set at line 5 of this procedure;
        // other nodesˆ.notify are set by code below.

    // notify descendants:
    wˆ.bin_leftˆ.notify := true         // unnecessary but harmless if
    wˆ.bin_rightˆ.notify := true        // children are leaves
    reinitialize (w)
rtn:
    if current_instance = &instances[1]
        current_instance := &instances[0]
    else
        current_instance := &instances[1]
```

Figure 2: Gupta and Hill’s adaptive combining tree barrier.

To construct a fuzzy version of their adaptive combining tree barrier, Gupta and Hill observed that a simple separation of the arrival and wakeup phases does not suffice to minimize unnecessary waiting: processes may not call `exit_barrier` in the same order they called `enter_barrier`. Processes that acquired nodes near the fringe of the tree in `enter_barrier`, but which call `exit_barrier` early might have to wait (needlessly) for processes that acquired nodes near the root of the tree in `enter_barrier`, but which call `exit_barrier` relatively late. The solution to this problem is to employ a separate tree traversal in the wakeup phase of the algorithm, so that processes that call `exit_barrier` early busy-wait on nodes that are close to the root.

Code for a modified version of Gupta and Hill's fuzzy adaptive combining tree barrier appears in figure 3. We have broken out the recursive part of the wakeup phase as a separate routine to make the use of alternating trees explicit. We have also introduced changes to address some subtle bugs in the original version that are not obvious on paper, but which emerged in the course of experimentation:

(1) set *n*'s `occupied` flag in the first `if` statement of `rec_exit`, even if *n*'s `notify` flag is already set. This prevents a process from entering a node and setting the `notify` flags of children when some other process has already returned from that node and reinitialized one of the children.

(2) set *p*'s `last_visitor` field in the third `if` statement of `rec_exit`, prior to releasing the lock on *n* and moving (recursively) up the tree. This ensures that when a process makes a recursive call up into a parent node, no other process will reinitialize that node out from under it (the other process must first acquire the locks on the children).

(3) introduce `left_notify` and `right_notify` flags to ensure that no node is notified more than once. Without these flags it is possible for a process to climb into a node, notice that it has been notified, reinitialize it, and return, while another process higher in the tree is about to notify it again. If the second process then returns into the re-notified node, it will enter an infinite wait, expecting some other process to clear the `notify` flag.

(4) wait for *n* to be reinitialized by a more recently-arriving process only after `last_visitor` has stabilized. After the series of nested `else`s in `rec_exit` we know that the barrier has been achieved. After the `if not leaf` clause near the end of `rec_exit` we know that no other process will climb through node *n* in the future. If no other process has climbed through after us, then we can safely reinitialize *n*. Otherwise, we wait for the last process that got through to reinitialize it. It is safe to execute the last `if` statement after *n* has been reinitialized: `n^.last_visitor` will be unequal to anybody's pid, and `n^.notify` will be false.

    In the original version of the algorithm, the wait for reinitialization appeared in a separate `if n^.last_visitor <> pid` clause immediately after the nested `else`s — before the setting of child `notify` flags. The wait could inadvertently be skipped if a later-arriving process had not yet set `n^.last_visitor`; we could therefore return into a child and reinitialize it before the late-arriving process set the child's `notify` flag.

```
type node = record
    lock : volatile syncvar := free
        // Volatile fields are changed only by a process that holds the lock,
        // and are always changed consistently.
    visited : volatile (no, left, right) := no
        // Has this node been visited in enter_barrier,
        // and if so, from which child?
```

```
    last_visitor : volatile pid := none
        // Last process to visit this node in rec_exit.
        // Modified only by a process that holds locks both on this node
        // and on the child through which it reached this node.
    root, bin_left, bin_right, bin_parent : ^node := // tree
    left, right, parent : volatile ^node := bin_left, bin_right, bin_parent
    left_notify, right_notify : volatile Boolean := false, false
        // added to prevent multiple notifies
    notify, occupied : volatile Boolean := false, false
        // Notify will eventually become true once the barrier is achieved.
        // Occupied is true iff some process has ended its search at this node
        // in rec_exit, and will not continue until the notified bit is set.
type instance = record
    my_leaf : ^node := // as appropriate, in tree
private instances : array [0..1] of instance
private current_instance : ^instance := &instances[0]

procedure enter_barrier ()
    n : ^node := current_instance^.my_leaf
    loop
        w : ^node := n^.parent
        if w = nil                      // We are the last arrival.
            n^.root^.notify := true
            return
        acquire (w^.lock)
        if w^.visited = no
            exit loop
        release (w^.lock)
        // Continue; n's parent pointer will eventually change,
        // and we'll get a new w.
    if w^.left = n
        w^.visited := left
    else
        w^.visited := right
    release (w^.lock)
    // adapt tree:
    loop
        if w^.visited = left
            o : ^node := w^.right
        else
            o : ^node := w^.left
        acquire (o^.lock)
        if o^.visited = no
            exit loop
        release (o^.lock)
        // Continue; w's right or left pointer will eventually change,
        // and we'll get a new o.
    p : ^node := w^.parent
    if p <> nil
        if p^.left = w
            p^.left := o
        else
            p^.right := o
        // This change to p^.left or p^.right may allow another process
        // to break out of the second loop above.
    o^.parent := p
        // This may allow another process to break out of the first loop above.
    release (o^.lock)
```

```
procedure rec_exit (n : ^node)
// n^.lock is held, and n^.last_visitor = my_pid
    if n^.notify
        n^.occupied := true             // missing in original
        release (n^.lock)
    else
        p : ^node := n^.bin_parent
        if p = nil                      // n is the root
            n^.occupied := true
            release (n^.lock)
            repeat until n^.notify      // spin
                // root^.notify is set at line 5 of enter_barrier;
                // other nodes^.notify are set by code below.
        else
            acquire (p^.lock)
            if not p^.occupied
                p^.last_visitor := my_pid
                release (n^.lock)       // before previous line in original
                rec_exit (p)            // recursive call
            else
                release (p^.lock)
                n^.occupied := true
                release (n^.lock)
                repeat until n^.notify  // spin
                    // root^.notify is set at line 5 of enter_barrier;
                    // other nodes^.notify are set by code below.
    // At this point we know the barrier has been achieved.
    // Each process is responsible for notifying any not-yet-notified children
    // of nodes along the path between its leaf and the node it occupied.
    // Each process also is responsible for re-initializing each node that it
    // was the last to visit on that path.
    if not leaf (n)
        // signal children ONCE AND ONLY ONCE:
        acquire (n^.bin_left^.lock)
        if n^.last_visitor = my_pid and n^.left_notify = false
            n^.left_notify := true
            n^.bin_left^.notify := true
        release (n^.bin_left^.lock)
        acquire (n^.bin_right^.lock)
        if n^.last_visitor = my_pid and n^.right_notify = false
            n^.right_notify := true
            n^.bin_right^.notify := true
        release (n^.bin_right^.lock)
        // The left_notify and right_notify flags prevent multiple notifies
        // of the same child, which could otherwise lead to an infinite wait.
    // At this point if n^.last_visitor = my_pid it will stay so,
    // because the children have been notified while their locks were held
    // by the most recent process to climb through n.
    if n^.last_visitor = my_pid
        reinitialize (n)
    else
        repeat while n^.notify          // spin
            // Wait until reinitialized; that way we don’t return into a child
            // and reinitialize it before its notify flag gets set.
            // This line was originally before the sets of child notify flags.
```

```
procedure exit_barrier ()
    n : ^node := current_instance^.my_leaf
    acquire (n^.lock)
    n^.last_visitor := my_pid
    rec_exit (n)
    if current_instance = &instances[1]
        current_instance := &instances[0]
    else
        current_instance := &instances[1]
```

Figure 3: Gupta and Hill's fuzzy adaptive combining tree barrier (modified).

3. New Algorithms

It is well known that contention for memory locations and processor-memory interconnect bandwidth seriously degrades the performance of traditional busy-wait synchronization algorithms. Previous work has shown how to eliminate this contention for mutual exclusion locks, reader-writer locks, and barriers [1, 4, 9, 11, 15-17]. The key is for every process to spin on separate *locally-accessible* flag variables, and for some other process 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. Our experience indicates that the elimination of remote spinning can yield dramatic performance improvements.

We present our barriers in pseudo-code below; complete C versions for the Butterfly 1, TC2000, and KSR 1 can be obtained via anonymous ftp from cs.rochester.edu (directory pub/scalable_synch/adaptive).

3.1. A Local-Spinning Adaptive Combining Tree Barrier

To eliminate remote spinning in the (non-fuzzy) adaptive combining tree barrier of figure 2, we must address three sources of spinning on remote locations:

(1) While waiting for the barrier to be achieved, processes spin on a flag in a dynamically-chosen tree node. (This is a problem on NUMA machines, though not on machines with coherent caches.)

(2) In order to ensure consistent modifications to the tree, processes acquire and release `test_and_set` locks in every node they visit.

(3) In both the original search for a parent node at which to wait, and in the subsequent search for a sibling node whose parent should be changed, processes spin until they succeed in locking the node they are looking for *and* find it to be unvisited.

For NUMA machines, we eliminate the first type of remote spinning by using a statically-allocated per-process flag, and storing a pointer to this flag in the dynamically-chosen tree node.[3] We eliminate the second and third types of remote spinning by using `fetch_and_store` instructions[4] to modify the tree in an asynchronous, wait-free fashion [8]. Contention-free spin locks [15] would eliminate the second kind of remote spin, but not the third. With the third kind of spin, the number of remote references per processor in a barrier episode has no fixed bound.

[3] On the KSR 1, we spin on flags in the tree nodes themselves, and rely on coherent caching to bring the flag into local memory.

[4] `Fetch_and_store (L, V)` returns the value in location `L` and replaces it with `V`, as a single atomic operation.

The wait-free solution leads to a fixed upper bound on the number of remote references as well as achieving higher concurrency and lower per-node overhead than the locking alternative.

Code for a (non-fuzzy) adaptive combining tree barrier with local-only spinning appears in figure 4. In general form, it mirrors figure 2. The code to eliminate remote spinning while waiting for notification is more or less straightforward. Rather than set a `visited` flag, a process uses `fetch_and_store` to set a `visitor` pointer. The atomicity of the operation enables it to determine if another process has already acquired the node, in which case it puts that process's pointer back.

By using a more powerful `compare_and_swap` instruction [8] we could eliminate the need to re-write pointers that are erroneously over-written.[5] A similar optimization is also possible on the KSR 1; see section 4. We have presented the algorithms with `fetch_and_store` because it is available on a wide variety of machines, including the Butterfly 1 and the TC2000. There is no correctness problem induced by re-writing pointers, since the values of `visitor` fields are not used (except to compare them to `&pseudodata`) until after the barrier is achieved, and all mistakenly overwritten values are restored before that time.

The code required to eliminate per-node locks and to avoid the spins while looking for unvisited parent and sibling nodes is more subtle. With simple `fetch_and_Φ` instructions we cannot change child and parent pointers in a consistent fashion in one atomic step. We have therefore resorted to an asynchronous approach in which processes may see changes to the tree in different orders. In particular, if a process finds that the parent *p* of *w* has already been visited, we allow it to proceed immediately to *w*'s grandparent, even though some other process must of necessity be about to change the pointer from *w* to *p*. With sufficient skew in the arrival times of processes, changes to the tree occur in the same order as they do in figure 2. When processes arrive at about the same time, however, the ''winner'' may follow more than one parent pointer to reach, and visit, the root.

When splicing a sibling *o* into its grandparent *p* (see figure 1), we change *p*'s child field first, before changing *o*'s parent field. In between, there is a timing window when a process climbing up through *w*'s sibling may find *o* and attempt to splice it into its *great*-grandparent. Because the updates to *o*'s parent field are unsynchronized, we must take care to recover in the event that they occur in the incorrect order.[6] `Depth` fields in each node enable us to discover whether the new value of a parent field is an improvement on the old, and to restore the old value if necessary. It is possible for a process to climb up through a node when its parent pointer has just been overwritten with an out-of-date value, and before the better value is restored, but no correctness problems result: the process simply follows more pointers than it would have if it had missed the timing window. (The situation is analogous to what occurs in a concurrent B^{link} tree, when a newly-inserted node becomes available via a pointer from its left sibling, but not yet from its parent [10, 12, 18].)

---

[5] `Compare_and_swap (L, O, N)` compares the value in location `L` to `O` and, if they are equal, replaces `O` with `N`, as a single atomic operation. It returns `true` if it performed the swap, and `false` otherwise.

[6] `Compare_and_swap` does not help in this case; one can read the pointer to determine whether it is desirable to overwrite it, but even an immediately subsequent `compare_and_swap` may fail because some other process has overwritten the pointer in the interim.

```
shared pseudodata : Boolean

type node = record
    // Volatile fields may change spontaneously; there are no locks.
    visitor : volatile ˆBoolean := &pseudodata
        // first process to visit this node; may be temporarily inconsistent,
        // but will stabilize before being dereferenced
    bin_left, bin_right, bin_parent : ˆnode := // tree
    left, right, parent : volatile ˆnode := bin_left, bin_right, bin_parent
    depth, inorder : integer := // as appropriate, in unmodified tree;
        // inorder allows us to determine left and right descendancy.

type instance = record
    f : Boolean := false
    root, my_leaf, my_internal_node : ˆnode := // as appropriate, in tree
        // my_internal_node is used only for reinitialization.

private instances : array [0..2] of instance
    // separate copy for each process, but allocated
    // in memory accessible to other processes
private current_instance : ˆinstance := &instances[0]
private previous_instance : ˆinstance := &instances[2]

procedure barrier ()
    // find place to wait:
    n : ˆnode := current_instanceˆ.my_leaf
    loop
        w : ˆnode := nˆ.parent
        if w = nil
            // signal achievement of barrier:
            current_instanceˆ.rootˆ.visitorˆ := true
                // may unblock a process at the spin below
            goto rtn
        x : ˆnode := fetch_and_store (&wˆ.visitor, &current_instanceˆ.f)
        if x = &pseudodata
            exit loop
        wˆ.visitor := x                  // already visited; put it back
        n := w                           // continue up the tree

    // adapt tree:
    if nˆ.inorder < wˆ.inorder
        o : ˆnode := wˆ.right
    else
        o : ˆnode := wˆ.left
    p : ˆnode := wˆ.parent
    if p = nil
        oˆ.parent := nil
    else
        // update down pointer:
        if wˆ.inorder < pˆ.inorder
            pˆ.left := o
        else
            pˆ.right := o
        // update up pointer:
        loop
            t : ˆnode := fetch_and_store (&oˆ.parent, p)
            if t <> nil and then tˆ.depth > pˆ.depth
                exit loop                // swap was a good thing
            // else some other process linked o even higher in the tree;
            // continue loop to undo our poorer update
            p := t

    // await notification and pass on the news:
```

```
    repeat until current_instance^.f    // spin
        // set at the root by line 7 of this procedure;
        // set at other nodes by the code below
    w^.bin_left^.visitor^ := true
    w^.bin_right^.visitor^ := true
rtn:
    reinitialize (previous_instance)
    previous_instance := current_instance
    if current_instance = &instances[2]
        current_instance := &instances[0]
    else
        current_instance := current_instance + 1
```

Figure 4: An adaptive combining tree barrier with local-only spinning.

At first glance, it would appear that a potentially unbounded number of remote references might be performed while executing the loop to update *o*'s parent field. This would violate our claim of performing only a bounded number of remote references per barrier episode. Fortunately, the number of loop iterations is bounded by $depth(o)-1$, since each iteration sees *o*'s parent link move at least one step closer to the root. Moreover, the worst case is very unlikely; in practice one would expect to see a single iteration.

One might suspect that recovery might also be required when we update to children, but in fact these updates are serialized. If *w* is initially the left child of *p*, then initially only the process that visits *w* (call this process *X*) can change *p*'s left child field. Moreover only process *X* can cause any node to the left of *p* (other than *w*) to point to *p* as parent, so no other process will acquire the ability to modify *p*'s left child field until after *X* has first made *p* point to *o*, and then made *o* point to *p*. Inorder traversal numbers allow us to determine whether a given node is to the left or the right of its parent without inspecting the parent's (possibly inconsistent) child pointers.

Because of the asynchrony with which processes climb the tree, a slow process can still be modifying pointers when all of its peers have left the barrier and continued other work. We are therefore unable to reinitialize nodes on the way out of the barrier, as did Gupta and Hill in figure 2. Instead, we employ *three* sets of data structures. We reinitialize the one that was used before the current barrier episode, and that will not be used again until after the next episode. Each process takes responsibility for reinitializing its own leaf and one (statically determined) internal node.

3.2. The Fuzzy Variant

To eliminate remote spinning from the fuzzy barrier code in figure 3, we replace `notify` flags with pointers to local flags, eliminate `test_and_set` locks, and adapt the tree asynchronously, just as we did in the non-fuzzy version. Code to enter the barrier and adapt the tree can be taken almost verbatim from figure 4. In `exit_barrier`, however, we must find a way for processes to climb to the highest unoccupied node without the double-locking of the original fuzzy algorithm. In figure 3, a process retains the lock on a child node while locking and inspecting its parent. If the parent is unoccupied, the process releases the child. If the parent is already occupied, the process occupies the child. Our solution is again to adopt an asynchronous approach, in which each process writes a pointer to its wakeup flag into *every* node that appears to be unoccupied on the path from its leaf to the root.[7]

---

[7] Because it may end up occupying more than one node, a process must store pointers to a local flag, even on the KSR 1; there is no one tree node in which it could spin on a flag.

Code for a fuzzy adaptive combining tree barrier with local-only spinning appears in figure 5. With sufficient skew in arrival times, processes will write pointers to their wakeup flags into distinct nodes of the tree, ending at the same nodes at which they would have ended in figure 3. If processes arrive at about the same time, however, more than one of them may write a pointer to its wakeup flag into the same node. Since every process begins by writing its pointer into a leaf, at least one of the pointers will never be overwritten. As an optimization, each process is informed at wakeup time of the node at which its pointer was found. It must perform wakeup operations along the path back down from this node, but can skip any higher-level nodes at which its pointer was overwritten.

For the sake of space and time efficiency, we have written `exit_barrier` as a non-recursive routine. As in our non-fuzzy barrier, we employ three sets of data structures to cope with the asynchrony of tree adaptations and reinitialization. By reinitializing the data structures of the previous barrier instance, we eliminate the need for the `last_visitor` fields, which account for much of the complexity of figure 3.

```
shared pseudodata : ^node

type node = record
    // Volatile fields may change spontaneously; there are no locks.
    visited, notified : volatile Boolean := false, false
        // visited = true iff some process has visited this node in
        // enter_barrier.
        // notified will eventually become true once the barrier is achieved.
    owner : volatile ^node := // address of appropriate f field for leaves,
                              // &pseudodata for internal nodes
        // address of f field of a recent visitor in exit_barrier
        // (not necessarily the *most* recent)
    bin_left, bin_right, bin_parent : ^node := // tree
    depth, inorder : integer := // as appropriate, in unmodified tree;
        // inorder allows us to determine left and right descendancy.
    left, right, parent : volatile ^node := bin_left, bin_right, bin_parent

type instance = record
    f : ^node := nil                    // node at which we were awakened
    root, my_leaf, my_internal_node : ^node := // as appropriate, in tree
        // my_internal_node is used only for reinitialization.

private instances : array [0..2] of instance
    // separate copy for each process, but allocated
    // in memory accessible to other processes
private current_instance : ^instance := &instances[0]
private previous_instance : ^instance := &instances[2]

procedure enter_barrier ()
    n : ^node := current_instance^.my_leaf
    loop
        w : ^node := n^.parent
        if w = nil
            // signal achievement of barrier:
            current_instance^.root^.notified := true
            current_instance^.root^.owner^ := current_instance^.root
                // may unblock a process at the spin in exit_barrier,
                // informing it that we woke it up at the root
            return
        if fetch_and_store (&n^.visited, true) = false
            exit loop
        n := w                          // continue up the tree

    // adapt tree:
```

```
    if n^.inorder < w^.inorder
        o : ^node := w^.right
    else
        o : ^node := w^.left
    p : ^node := w^.parent
    if p = nil
        o^.parent := nil
    else
        // update down pointer:
        if w^.inorder < p^inorder
            p^.left := o
        else
            p^.right := o
        // update up pointer:
        loop
            t : ^node := fetch_and_store (&o^.parent, p)
            if t <> nil and then t^.depth > p^.depth
                exit loop               // swap was a good thing
            // else some other process linked o even higher in the tree;
            // continue loop to undo our poorer update
            p := t

procedure exit_barrier ()
    n : ^node := current_instance^.my_leaf
    if n^.notified
        goto rtn
    p : ^node := n^.bin_parent
    loop
        if p^.owner = &pseudodata
            p^.owner := &current_instance^.f
            if p^.notified
                exit loop
            else if p^.bin_parent = nil
                repeat
                    p := current_instance^.f
                until p <> nil          // spin
                    // set at root by line 7 of enter_barrier;
                    // set at other nodes by the code below
                exit loop
            else
                p := p^.bin_parent
        else if p^.notified
            exit loop
        else
            repeat
                p := current_instance^.f
            until p <> nil              // spin
                // set at root by line 7 of enter_barrier;
                // set at other nodes by the code below
            exit loop
    // work way back down to leaf, giving notifications:
    while p <> current_instance^.my_leaf
        if n^.inorder < p^.inorder
            o := p^.bin_right
            p := p^.bin_left
        else
            o := p^.bin_left
            p := p^.bin_right
```

```
          o^.notified := true
          o^.owner^ := o
              // may unblock a process in the spin above,
              // informing it that we woke it up at o
rtn:
    reinitialize (previous_instance)
    previous_instance := current_instance
    if current_instance = &instances[2]
        current_instance := &instances[0]
    else
        current_instance := current_instance + 1
```

Figure 5: A fuzzy adaptive combining tree barrier with local-only spinning.

4. Performance Results

We have compared the performance of the various forms of combining tree barrier with that of the centralized and logarithmic barriers found to perform best in previous experiments [15]. After describing our experimental environment in section 4.1, we consider latency for non-fuzzy barriers in section 4.2, focusing in particular on the impact of skew in arrival times. We then consider the utility of fuzzy algorithms in section 4.3, with an emphasis on the cross-over points at which the savings in needless spinning makes up for additional overhead.

4.1. Experimental Environment

Our timing tests employed three different machines: Rochester's BBN Butterfly 1 multiprocessor, a more modern BBN TC2000 machine at Argonne National Laboratory, and the Kendall Square KSR 1 at the Cornell Theory Center.

The Butterfly 1 employs MC68000 processors clocked at 8 MHz, with up to four megabytes of memory (one on our machine) located at each processor. There are no caches, coherent or otherwise. Each processor can access its own memory directly, and can access the memory of any node through a $\log_4$-depth switching network. 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 firmware) 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 TC2000 is architecturally similar to the Butterfly 1, but employs 20 MHz MC88100 processors with (non-coherent) caches and a faster $\log_8$-depth switching network based on virtual circuit connections rather than packet switching. With caching disabled, a remote memory reference takes about 1.9 μs, slightly over 3 times as long as a local reference, and about 13 times as long as a cache hit. Experiments by Markatos and LeBlanc [14] indicate that while the TC2000 has relatively good switch bandwidth and latency, it is starved for shared memory bandwidth, and hence vulnerable to contention. One would expect the centralized barriers to perform comparatively badly on the TC2000; our results confirm this expectation.

The KSR 1 is a *cache-only* machine constructed of custom 64-bit two-way superscalar processors clocked at 20 MHz, and connected by a two-level hierarchy of rings. Each lower-level ring houses 32 processing nodes and an interface to the (single) upper-level ring. The memory at each processing node is organized as a 32 MB secondary cache, with a 512 KB ''subcache.'' Access time ratios for the subcache, the local (secondary) cache, a remote cache in the same ring, and a remote cache in a different ring are approximately 1:9:88:300, making remote operations substantially more expensive than on the Butterfly 1 or the TC2000. A hardware coherence protocol maintains sequential consistency across the caches of all processors.

The Butterfly 1 supports a 16-bit atomic `fetch_and_clear_then_add` operation in firmware. This operation takes three arguments: the address of the destination operand, a mask, and a source operand. For the locks in figures 2 and 3 we perform a `test_and_set` by specifying a mask of `0xFFFF` and an addend of `1`. For the central barriers we perform a `fetch_and_increment` by specifying a mask of `0` and an addend of `1`; for the other barriers we perform a `fetch_and_store` by specifying a mask of `0xFFFF` and an addend of the value to be stored. In comparison to ordinary loads and stores, atomic operations are relatively expensive on the Butterfly 1; `fetch_and_clear_then_add` takes slightly longer than a call to a null procedure.

The TC2000 supports the MC88100 `XMEM` (`fetch_and_store`) instruction in hardware, at a cost comparable to that of an ordinary memory reference. It provides additional atomic operations in software, but these must be triggered in kernel mode: they make use of a special hardware mechanism that locks down a path through the switch to memory. They are available to user programs only via kernel calls, and as on the Butterfly 1 are relatively expensive. For Gupta and Hill's algorithms (figures 2 and 3), `XMEM` can be used directly to implement `test_and_set`. The expense of the kernel-mediated `fetch_and_increment` contributes to the poor performance of the centralized barriers.

The KSR 1 supports atomic operations only via the ''locking'' of lines (''subpages'') in a subcache. The `acquire_subpage` operation brings a copy of a specified line into the local subcache and sets a special state bit. It fails (setting a condition code) if any other processor currently has the state bit set. The `release_subpage` operation clears the bit and permits it to be set in the next requesting processor in ring-traversal order. (There is an alternative form of `acquire_subpage` that stalls until the bit can be set, but Dunigan reports [3], and our experience confirms, that this stalling version performs worse than polling in a loop for all but very small numbers of processors.) By following a programming discipline in which modifications to a given word are made only when its line has been acquired, one can implement the equivalent of arbitrary `fetch_and_`Φ operations.

For Gupta and Hill's algorithms, `acquire_subpage` and `release_subpage` provide the equivalent of `test_and_set` locks. For the new algorithms (figures 4 and 5), we use these operations to bracket code sequences that read a pointer, decide if it should be changed, and if so change it, atomically, without the need to re-write mistakenly-swapped locations. Similar code could be written (albeit with the possibility of some remote spinning) for machines based on the MIPS R4000 or DEC Alpha architectures, which provide `load_linked` and `store_conditional` instructions.[8]

The barriers included in our timing tests are listed in figure 6, together with an indication of their line types for subsequent graphs. We have used solid lines for non-fuzzy algorithms, and dotted lines for fuzzy algorithms. When one algorithm has fuzzy and non-fuzzy variants, they share the same tick marks.

The *dissemination* barrier is due to Hensgen, Finkel, and Manber [7]. As mentioned in section 1, it employs $\lceil \log_2 P \rceil$ rounds of synchronization operations in a pattern that resembles a parallel prefix computation: in round k, process i signals process $(i+2^k) \bmod P$. The total number of synchronization operations (remote writes) is $O(P \log P)$ (rather than $O(P)$ as in other logarithmic time barriers) but as many as $\log P$ of these operations can proceed in parallel, when using non-overlapping portions of the interconnection network.

---

[8] `Load_linked` reads a memory location and saves some status information in the local cache controller. `Store_conditional` writes the location read by a previous `load_linked`, provided that no other processor has performed an intervening write to the same location, and that various possible interfering operations have not occurred on the local processor.

central flag
fuzzy central flag
dissemination
static tree
original adaptive combining tree
fuzzy original adaptive combining tree
local-spinning non-adaptive combining tree
fuzzy local-spinning non-adaptive combining tree
local-spinning adaptive combining tree
fuzzy local-spinning adaptive combining tree
KSR pthread barrier

Figure 6: Barrier algorithms tested.

Our previous experiments [15] found the dissemination barrier to be the fastest alternative on the Butterfly 1. The *static tree* barrier was a close runner-up. It has a slightly longer critical path, but less overall communication, and might be preferred when the impact of interconnect contention on other applications is a serious concern. Each process in the static tree barrier is assigned a unique tree node, which is linked into a 4-ary arrival tree by a parent link, and into a binary wakeup tree by a set of child links. Upon arriving at the barrier, each process spins on a local word whose four bytes are set, upon arrival, by the process's children. It then sets a byte in its parent and spins on another local word awaiting notification from its parent. The root process starts a downward wave of notifications when it discovers that all of its children have arrived.

The *central flag* barrier and its fuzzy variant employ a central counter and wakeup flag. On the Butterfly 1 and the TC2000, they pause after an unsuccessful poll of the flag for a period of time proportional to the number of processes participating in the barrier. Our previous experiments found this technique to be more effective at reducing contention (and increasing performance) than either a constant pause or a linear or exponential backoff strategy. On the KSR 1, processors can spin on copies of the flag in their local cache, and backoff is not required.

All of the other barriers were introduced in sections 2 and 3. The *original adaptive combining tree* and *fuzzy original adaptive combining tree* are from figures 2 and 3. The *local-spinning adaptive combining tree* and *fuzzy local-spinning adaptive combining tree* are from figures 4 and 5. The *local-spinning non-adaptive combining tree* and *fuzzy local-spinning non-adaptive combining tree* are from figures 4 and 5, but without the block of code that begins with ‘‘`adapt tree`’’. As noted in footnote 3, our code for the (non-fuzzy) adaptive and non-adaptive local-spinning combining tree barriers on the KSR 1 spins on flags in the tree nodes themselves, rather than using pointers to statically-allocated local flags.

For comparison purposes, we have included the barrier synchronization algorithm provided with KSR's `pthreads` library. In addition, all of the tree-based barriers on the KSR 1 were modified to use a central flag for the wakeup phase of the algorithm. Previous experiments [15] on the cache-coherent Sequent Symmetry found the combination of the arrival phase of the static tree barrier with a central wakeup flag to be the best-performing barrier on more than 16 processors. Experiments on the KSR 1 confirm that all of the tree-based barriers run faster with a central wakeup flag (see figure 8). The KSR 1 does not have broadcast in the same sense as bus-based machines, but it can perform a global invalidation with a single transit of each ring, and many of the subsequent reloads can occur in parallel.

For each of our timing tests we ran 1000 barrier episodes and calculated the average time per episode. On the Butterfly 1 and the TC2000, we placed each process on a different processing node, and ran with timeslicing disabled, to avoid interference from the scheduler. We observed that timings were repeatable to three significant digits. On the KSR 1, we were unable to disable the scheduler, but it usually arranged for each process to run without interference on a separate processor. To accommodate the occasional interruption, we averaged each experiment over at least three separate program runs, throwing out any data points that seemed unusually high with respect to other runs.

In many of the tests we introduced delays between barrier episodes, or between `enter_barrier` and `exit_barrier` calls in fuzzy tests. The delays were implemented by iterating an appropriate number of times around a loop whose execution time was calibrated at 10 μs. In some cases the number of iterations was the same in every process. In other cases we introduced random fluctuations. A figure caption indicating a delay of, say, 1 ms ± 500 μs indicates that the number of iterations of the 10 μs delay loop was chosen uniformly in the closed interval 50 . . 150. Random numbers were calculated off-line prior to the tests. In order to obtain a measure of synchronization cost alone, we subtracted delays and loop overhead from the total measured time in each test. On the KSR 1, each tree node was split among two cache lines, with all read-only fields in one line, and all mutable fields in the other.

4.2. Basic Barrier Latency

Figure 7 plots the time required to achieve a barrier against the number of processes (and hence processors) participating in the barrier, with no inter-episode or fuzzy delays. We can observe that the explicit locking and non-local spinning of the original adaptive combining tree barriers (□) impose a large amount of overhead. We can also see the impact of contention: the performance of the centralized barriers (○) degrades markedly on all three machines as the number of processes increases. The curves for the original adaptive combining tree barriers also lose their logarithmic shape and assume a roughly linear upward trajectory around 20 processes on the TC2000 and (less dramatically) around 30 processes on the Butterfly 1. Contention in the centralized barriers on the TC2000 is severe enough, even with exponential backoff, to make the curves appear highly erratic. Timings are repeatable, however; the number of processes is simply not the dominant factor in performance. More important is the interconnection network topology and the assignment of processes and variables to particular processing nodes.

The dotted (fuzzy) and solid (regular) curves for the local-spinning combining tree barriers (× and △) show that fuzziness is a small net loss; the extra overhead is pointless in the absence of fuzzy computation, particularly on the Butterfly 1, with its expensive atomic operations.[9]

The fastest algorithms on the NUMA machines are the dissemination barrier (•) and static tree barrier (▪). None of the combining tree barriers come close in this ‘‘tight loop’’ case. The fastest algorithm on the KSR 1 is the static tree with flag wakeup (▪), followed by the barrier from KSR’s `pthreads` library (∗) and the local-spinning adaptive (×) and non-adaptive (△) combining tree barriers. The dissemination barrier does not perform as well on the KSR 1 as it does on the BBN machines. It requires $O(n \log n)$ remote operations (as opposed to $O(n)$ for the tree-based barriers), and fewer of these operations can take place in parallel with the KSR topology.

Comparing the bottom of figure 7 to the graph in figure 8 reveals that the pseudo-broadcast capability of the KSR 1 makes flag wakeup a clear performance win. At the same time, comparing the three graphs in figure 7 reveals that the NUMA machines are able to synchronize much faster than the KSR 1. The fastest 64-node barrier on the Butterfly 1 takes 151 μs; the fastest 64-node barrier on the KSR 1 takes 587 μs. These numbers are in rough proportion to the maximum

---

[9] In figure 8, fuzziness is a net loss on the KSR 1 for the local-spinning adaptive combining tree barrier without flag wakeup, but a small net win for its non-adaptive cousin. The difference is clearly related to the number of invalidations of tree nodes by `fetch_and_store` operations on `visitor` fields, but we have been unable to track it down precisely.

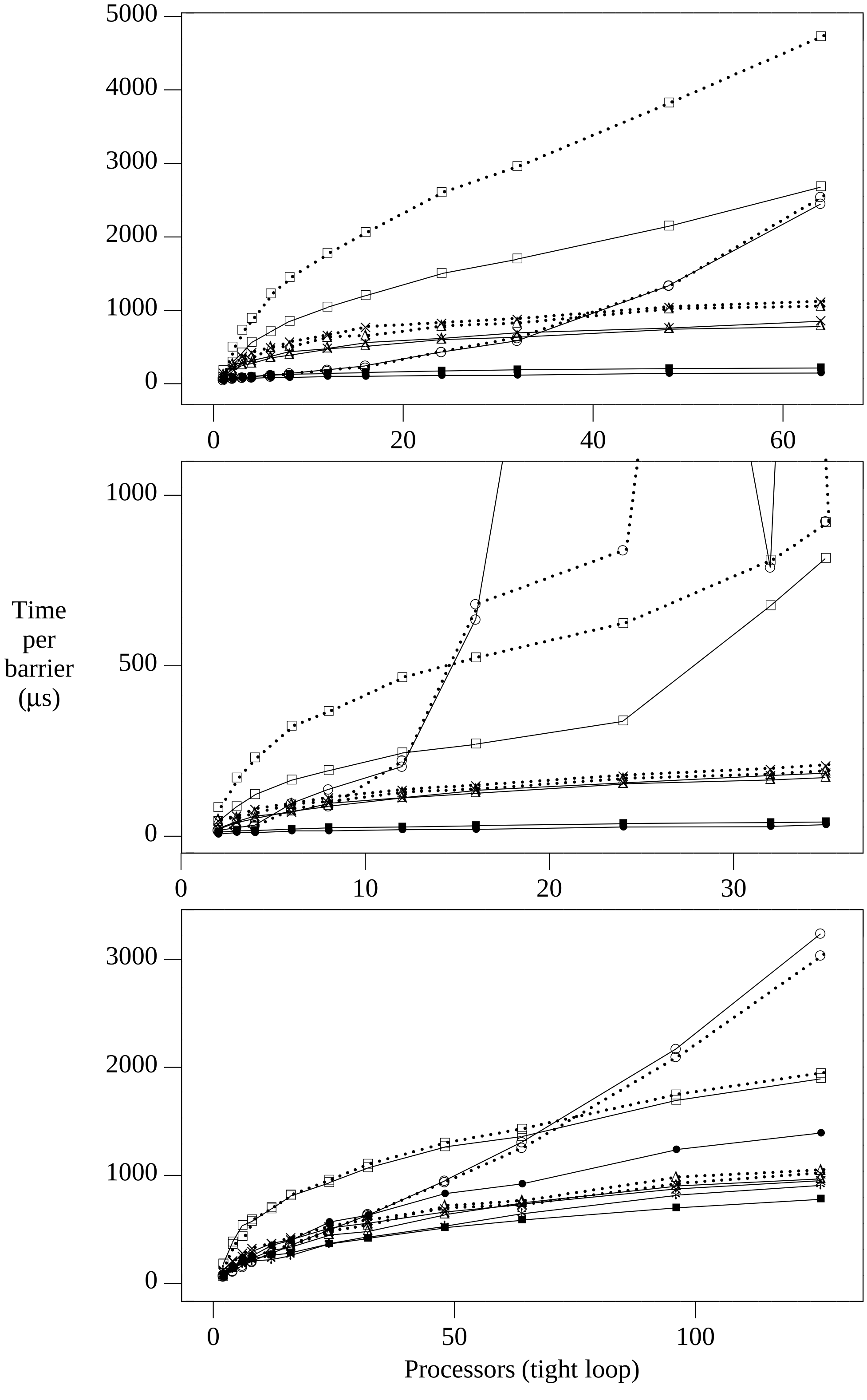

Figure 7: Basic barrier performance on the Butterfly 1 (top), TC2000 (middle), and KSR 1 (bottom).

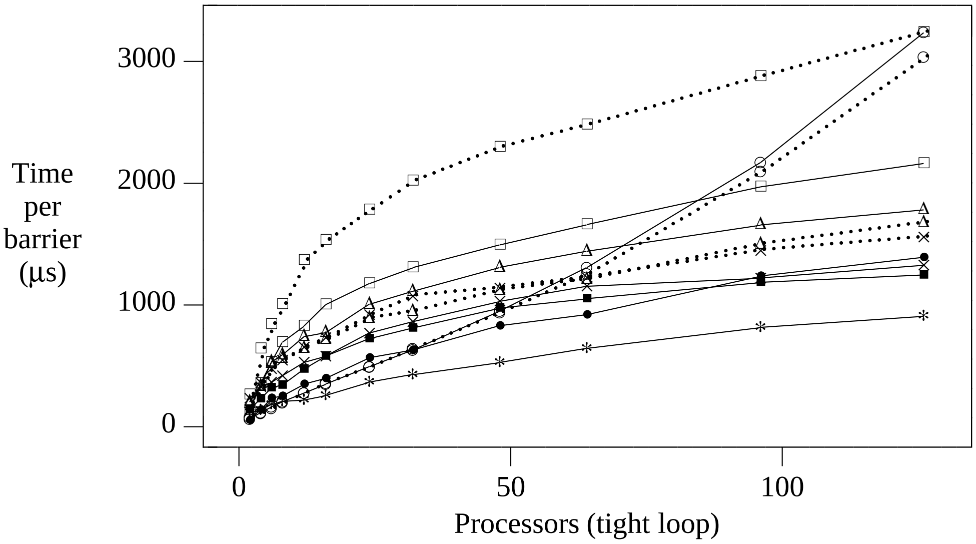

Figure 8: Basic barrier performance on the KSR 1, without a central flag for wakeup in the tree-based algorithms.

no-contention remote reference times on the two machines (4 and 15 μs, respectively). The KSR 1 is a decade newer, but pays dearly for cache coherence.

Any real program, of course, will have some inter-episode delay; figure 7 represents the (unrealistic) limiting case. Graphs similar to figure 7, but with significant inter-episode delays and skew, display much less contention in the centralized and original adaptive combining tree barriers. They also show the local-spinning adaptive combining tree barriers gaining a small advantage over their non-adaptive counterparts on the NUMA machines. (The adaptive versions are always better on the KSR 1.) On the KSR 1, the local-spinning adaptive combining tree barrier (with flag wakeup) becomes the best-performing algorithm, though again several others are close.

Figure 9 displays these effects by plotting time per barrier against the maximum random fluctuation in inter-episode delay, with 64 processes on the Butterfly 1, 35 processes on the TC2000, and 126 processes on the KSR 1. Point (x,y) represents the time y per barrier episode, with an inter-episode delay of 1 ms $\pm\, x$ μs. (Delays may differ by at most $2x$.) In the dissemination barrier (•), the static tree barrier (▪), and the non-adaptive combining tree barrier (▵), the time to achieve the barrier rises roughly linearly with the skew in arrival time. The slope is gentler for machines with a longer remote memory access delay, since this delay serves to hide a portion of the skew in arrival times: the reader of a value can be as much as one remote delay behind the writer before it will notice the skew.

Line-crossings between the local-spinning adaptive (×) and non-adaptive (▵) combining tree barriers suggest that adaptation serves to mitigate the increase in synchronization time due to increased arrival skew, but only to a very small extent.[10] Much more pronounced is the improvement in performance of the centralized (○) barriers on the KSR 1 and the fuzzy original adaptive combining tree (□) barrier on the TC2000, all of which were seen in figure 7 to suffer badly from contention. As the skew in inter-barrier times increases, delays in some processes allow other processes to finish their work and get out of the way.

---

[10] The dissemination barrier also appears to be able to cope with modest amounts of skew on the KSR 1; we're not sure why.

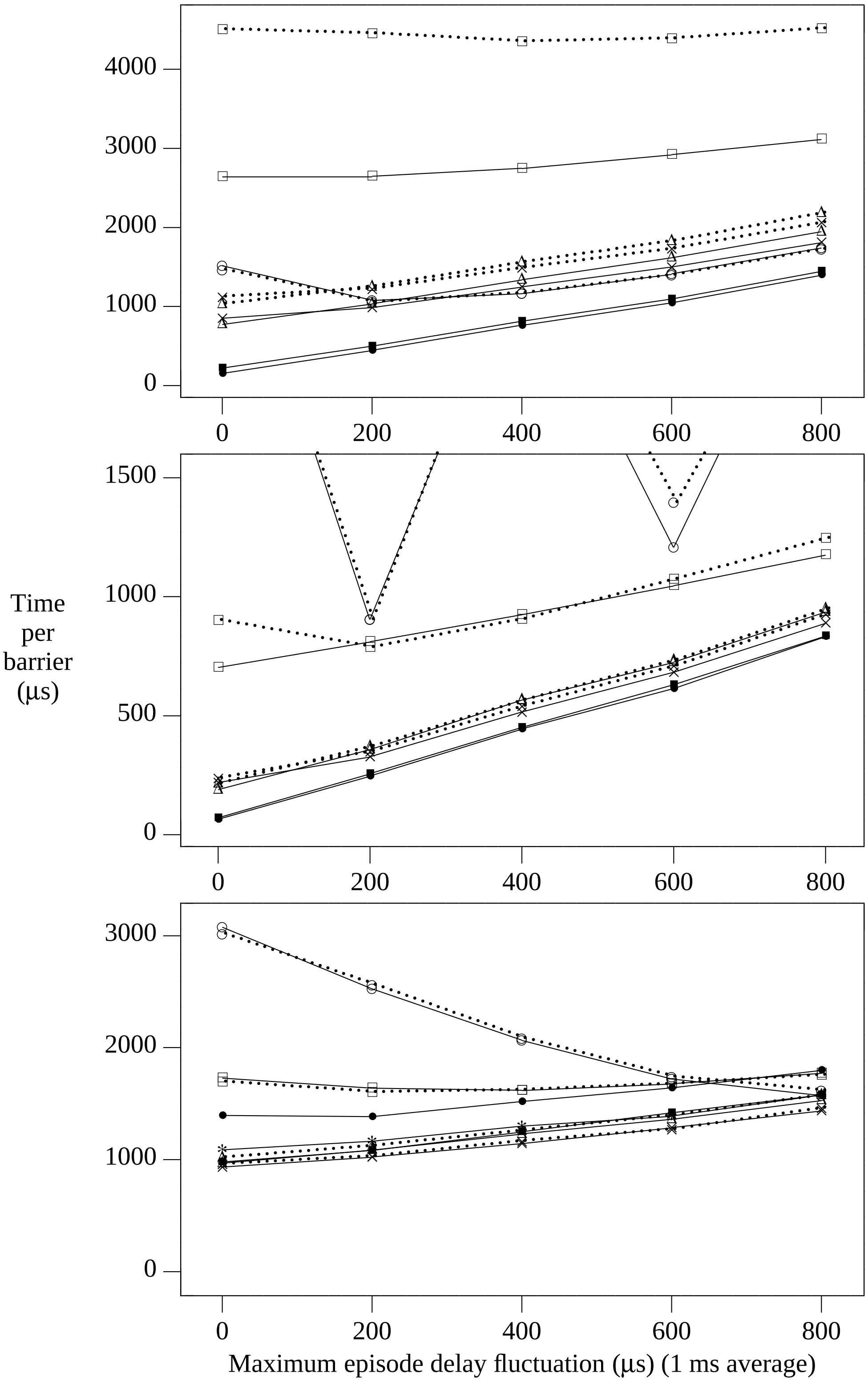

Figure 9: Dependence of performance on skew in process arrival times on the Butterfly 1 (top), TC2000 (middle), and KSR 1 (bottom).

4.3. Barrier Episodes with Fuzzy Delay

In figure 10, we have added a fuzzy delay of 500 μs to each iteration of the timing loop. We again plot time per barrier against the number of participating processes (processors). The inter-episode delay remains at 1 ms ± 200 μs. Introducing a reasonable amount of randomness into the fuzzy delays (up to 50%) had no noticeable effect on the timings.

In all cases the fuzzy versions of the centralized barrier (○) and the local-spinning combining tree barriers (× and △) outperform the non-fuzzy versions by significant amounts. On the Butterfly 1 the margin is large enough to enable the fuzzy centralized barrier to outperform the dissemination (•) and static tree (▪) barriers all the way out to 64 processes (though it appears that the curves would cross again on a bigger machine). In the original adaptive combining tree barriers (□), the benefits of fuzziness can outweigh the overhead of additional locking only on the newer two machines, where the fuzzy delay is comparatively long when measured in processor cycles, and atomic operations are not significantly more expensive than ordinary loads and stores.

The overhead of the fuzzy algorithms can be seen directly in figure 11 by comparing the performance of each fuzzy algorithm to that of its non-fuzzy counterpart. The graph plots the time required for processes to achieve a barrier against the length of the fuzzy delay (in addition to a 1 ± 0.5 ms inter-episode delay), with 64 processes on the Butterfly 1, 35 processes on the TC2000, and 126 processes on the KSR 1. For the centralized barrier, separating `enter_barrier` from `exit_barrier` introduces no overhead beyond the additional subroutine call. The separation therefore pays off with even very small fuzzy delays. For the adaptive (×) and non-adaptive (△) versions of the local-spinning combining tree barriers, the extra walk up the tree incurs overhead that is recovered almost immediately on the TC2000 and the KSR 1, and for fuzzy delays starting around 150 μs (15% of the inter-episode delay) on the Butterfly 1. For the original adaptive combining tree barrier (□), we need almost 700 μs before fuzziness pays off on the Butterfly 1, less than 20 μs on the TC2000, and almost none on the KSR 1. (Without flag wakeup, the break-even point is almost 600 μs on the KSR 1; these curves are not shown in the graph.) Both atomic operations and remote operations in general are relatively cheap on the TC2000. Atomic operations are expensive on the Butterfly 1; remote operations are expensive on the KSR 1.

On all three machines, the differences in performance between the adaptive (×) and non-adaptive (△) versions of the local-spinning combining tree barriers are relatively small. The difference is most noticeable on the KSR 1, where remote operations are much more expensive. Performing extra work to reduce the number of remote operations on the barrier's critical path is a good idea when those operations are slow. Since processor technology is improving more rapidly than memory or interconnect technology, it seems likely that adaptation will become increasingly important for future generations of machines.

On the Butterfly 1, slow atomic operations and slow processors make all of the combining tree algorithms uncompetitive: the static tree barrier, the dissemination barrier, and the fuzzy centralized barrier with proportional backoff work much better. On more modern machines, however, fuzziness and adaptation pay off. On the TC2000, the fuzzy local-spinning adaptive combining tree barrier outperforms all known alternatives when the amount of fuzzy computation exceeds about 10% of the average time between barriers. On the KSR 1, the arrival phase of the local-spinning adaptive combining tree barrier, combined with a central wakeup flag, outperforms all known alternatives with reasonable inter-barrier delays on more than about 50 processors. Its fuzzy version outperforms the non-fuzzy version when the amount of fuzzy computation exceeds about 5% of the average time between barriers. On smaller numbers of processors, when contention is not as serious a concern, the centralized barriers (regular and fuzzy) provide the best performance.

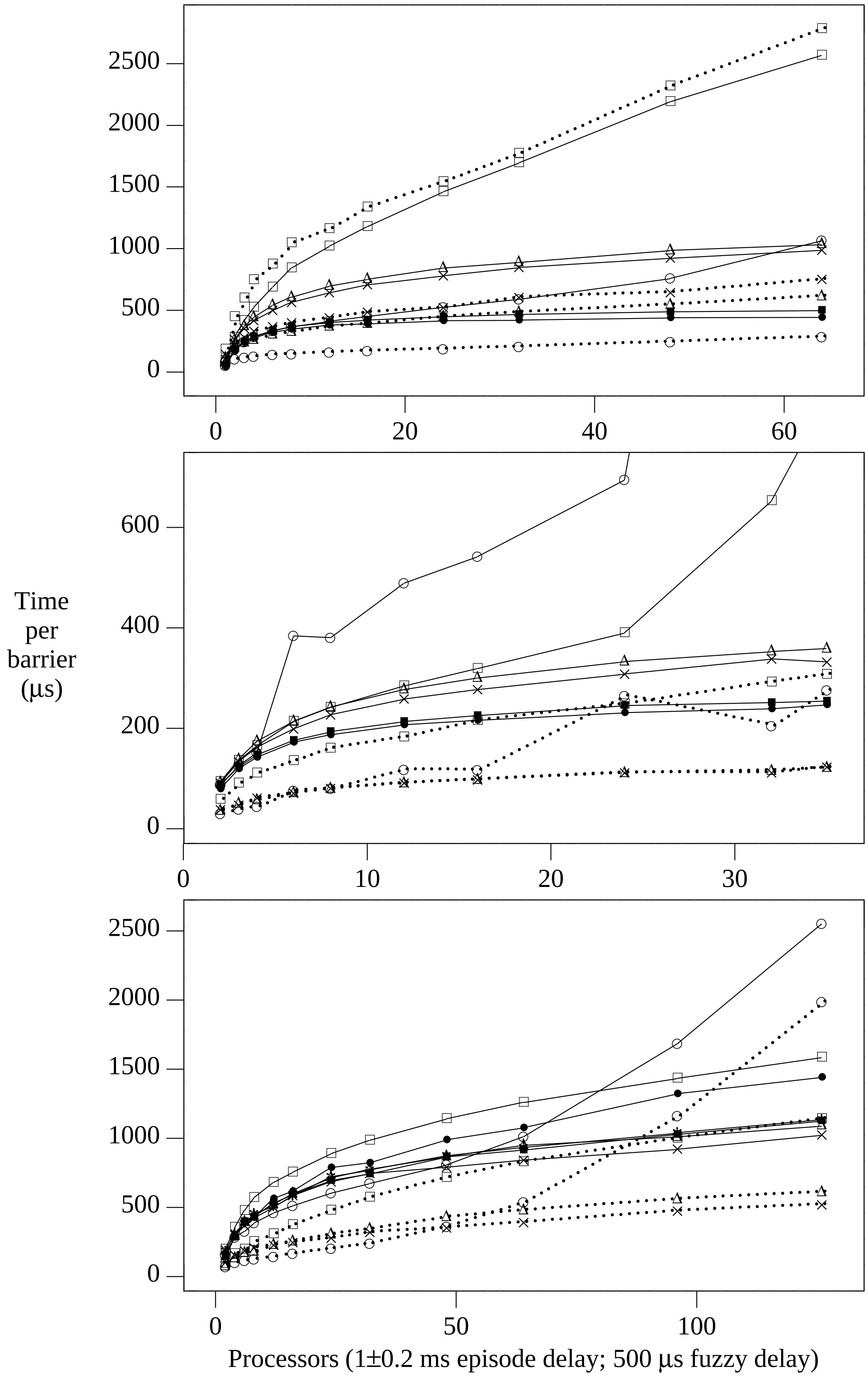

Figure 10: Barrier performance with a large amount of fuzzy computation on the Butterfly 1 (top), TC2000 (middle), and KSR 1 (bottom).

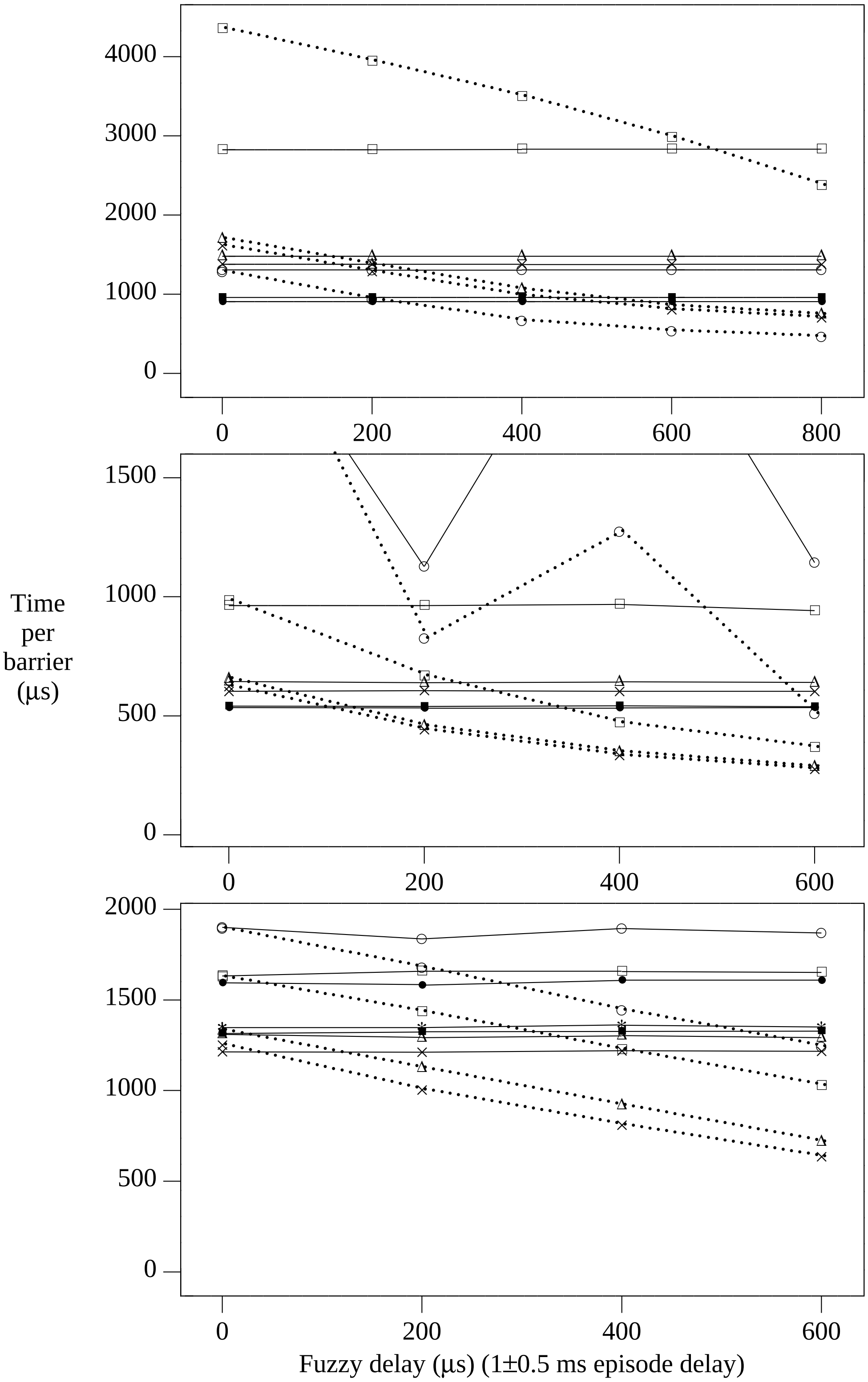

Figure 11: Dependence of performance on amount of fuzzy computation on the Butterfly 1 (top), TC2000 (middle), and KSR 1 (bottom).

5. Conclusions

On the basis of the current study, as well as previous work [15], we offer the following recommendations:

(1) Use a fuzzy barrier whenever the application can be structured with a significant amount of fuzzy computation. As shown in figure 11, the pay-off can be substantial.

(2) On modest numbers of processors, regardless of architecture, use the centralized barrier with proportional backoff (fuzzy or non-fuzzy version, as appropriate). Exactly how many processors constitute a ''modest'' number depends on the machine architecture, the arrival time skew, and the amount of fuzzy computation. In our experiments the cross-over point varies from a low of less than 10 to a high of over 40.

(3) On a large cache coherent machine (with unlimited replication and fast broadcast or multicast), combine a central wakeup flag with the arrival phase of either the static tree barrier, the local-spinning adaptive combining tree barrier, or preferably, if appropriate, the latter's fuzzy variant. (See the bottom [KSR] graphs in figures 7 and 10; see also figure 8.)

(4) On a large NUMA machine (or a cache-coherent machine that does not support unlimited replication and fast broadcast or multicast), use either the dissemination barrier, the static tree barrier, or preferably, if appropriate, the fuzzy local-spinning adaptive combining tree barrier. (See the middle [TC2000] graphs in figures 7 and 10.)

These recommendations apply primarily to modern machines, in which atomic `fetch_and_Φ` operations are comparable in speed to loads and stores, but in which all remote operations take scores of cycles. On the older Butterfly 1, the dissemination and static tree barriers achieve a substantial advantage over the competition by relying only on ordinary (fast) reads and writes. The only algorithm that ever outperforms them is the fuzzy centralized barrier (with proportional backoff), and only for applications with a significant amount of fuzzy computation, and with process arrival times that are skewed enough by load imbalance to keep contention under control (see figure 10).

On the somewhat more modern TC2000, atomic operations are nearly as fast as loads and stores, but remote operations are still only three times as expensive as an access to local memory (the hardware does not cache shared memory coherently). The centralized barriers suffer more from memory contention than they did on the Butterfly 1, and the fuzzy versions of the local-spinning combining tree barriers are fast enough to outperform the competition by a substantial margin in applications with fuzzy computation (see figures 10 and 11). Adaptation is not a major win, however, and the dissemination and static tree barriers still perform best for non-fuzzy computations.

On the most recent of the machines, the cache-coherent KSR 1, remote operations are some 300 times as expensive as a cache hit, and shared memory can be cached. The centralized barriers suffer less from contention than they did on the TC2000, but still enough to cause serious problems when the load is well balanced across a large number of processors (see figure 9). The best all-around strategy would appear to combine a log-time arrival phase with a central wakeup flag. For applications with a significant amount of fuzzy computation, the arrival phase should be taken from the fuzzy local-spinning adaptive combining tree barrier (see figure 10). For other applications, it should be taken from the static tree barrier or the (non-fuzzy) local-spinning adaptive combining tree barrier (see figure 7).

It is not yet clear whether cache coherence should be provided in hardware. Our experiments suggest that it is not required for high-performance synchronization. In fact, the Butterfly 1, while ten years older than the KSR 1, is able to achieve a 64-node barrier in one fourth the absolute time! In all cases, the key to scalable synchronization is to avoid situations that require inter-processor coherence traffic. The principal advantage of cache coherence, for synchronization purposes, is that it enables a process to spin locally on a variable that was not statically allocated in local main memory. One can achieve essentially the same effect on hardware without coherent caches by using `fetch_and_store` or `compare_and_swap` to replace a flag in an arbitrary location with a pointer to a flag in a local location, on which a process can then spin locally. This technique is a key part of the algorithms in figures 4 and 5, and should permit variants of most synchronization algorithms to run on NUMA machines.

Since even the fastest software barriers require time logarithmic in the number of participating processes, a more promising possibility for hardware support is to implement barriers directly, as in the Thinking Machines CM-5 or the Cray T3D. Hardware implementations can exploit large fan-in logic to dramatically reduce this logarithmic factor, effectively producing a constant-time barrier for machines of realistic size. It may be difficult, however, for hardware to match the flexibility of software barriers in such dimensions as variable numbers of participants, or concurrent, independent execution in separate machine partitions. Further experiments are needed to determine whether the performance gains outweigh these potential disadvantages.

Our work has demonstrated that log-length critical paths, adaptation, fuzziness, and local-only spinning are desirable and mutually compatible, and that their combination is competitive with the best known alternative techniques for busy-wait barrier synchronization. Since improvements in processor performance are likely to outstrip improvements in memory and interconnect performance for the foreseeable future, and since hardware designers now routinely implement fast atomic instructions, the local-spinning adaptive combining tree barriers, both regular and fuzzy, are of serious practical use.

Acknowledgments

Our thanks to Tom LeBlanc, Evangelos Markatos, Ricardo Bianchini, and Bob Wisniewski for their comments on this paper, and to Mark Crovella for his help in learning to use the KSR 1. Our thanks also to editor Gary Lindstrom and to the anonymous referees — referee B1 in particular. For the use of the TC2000, we thank the Advanced Computing Research Facility, Mathematics and Computer Science Division, Argonne National Laboratory. For the use of the KSR 1, we thank the Cornell Theory Center and its staff, especially Lynn Baird, Donna Bergmark, and Marty Faltesek.

References

[1] T. E. Anderson, ‘‘The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors,’’ *IEEE Transactions on Parallel and Distributed Systems 1*:1 (January 1990), pp. 6-16.

[2] E. D. Brooks III, ‘‘The Butterfly Barrier,’’ *International Journal of Parallel Programming 15*:4 (1986), pp. 295-307.

[3] T. H. Dunigan, ‘‘Kendall Square Multiprocessor: Early Experiences and Performance,’’ ORNL/TM-12065, Oak Ridge National Laboratory, May 1992.

[4] G. Graunke and S. Thakkar, ‘‘Synchronization Algorithms for Shared-Memory Multiprocessors,’’ *Computer 23*:6 (June 1990), pp. 60-69.

[5] R. Gupta and C. R. Hill, ‘‘A Scalable Implementation of Barrier Synchronization Using An Adaptive Combining Tree,’’ *International Journal of Parallel Programming 18*:3 (June 1989), pp. 161-180.

[6] R. Gupta, ‘‘The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors,’’ *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 54-63.

[7] D. Hensgen, R. Finkel, and U. Manber, ‘‘Two Algorithms for Barrier Synchronization,’’ *International Journal of Parallel Programming 17*:1 (1988), pp. 1-17.

[8] M. Herlihy, ‘‘Wait-Free Synchronization,’’ *ACM Transactions on Programming Languages and Systems 13*:1 (January 1991), pp. 124-149.

[9] W. C. Hsieh and W. E. Weihl, ‘‘Scalable Reader-Writer Locks for Parallel Systems,’’ MIT/LCS/TR-521, Laboratory for Computer Science, MIT, November 1991.

[10] T. Johnson and D. Shasha, ‘‘A Framework for the Performance Analysis of Concurrent B-tree Algorithms,’’ *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, April 1990, pp. 273-287.

[11] C. A. Lee, ‘‘Barrier Synchronization over Multistage Interconnection Networks,’’ *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, December 1990, pp. 130-133.

[12] P. L. Lehman and S. B. Yao, ‘‘Efficient Locking for Concurrent Operations on B-Trees,’’ *ACM Transactions on Database Systems 6*:4 (December 1981), pp. 650-670.

[13] B. Lubachevsky, ‘‘Synchronization Barrier and Related Tools for Shared Memory Parallel Programming,’’ *Proceedings of the 1989 International Conference on Parallel Processing II* (August 1989), pp. 175-179.

[14] E. P. Markatos and T. J. LeBlanc, ‘‘Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performance,’’ TR 420, Computer Science Department, University of Rochester, May 1992.

[15] J. M. Mellor-Crummey and M. L. Scott, ‘‘Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,’’ *ACM Transactions on Computer Systems 9*:1 (February 1991), pp. 21-65.

[16] J. M. Mellor-Crummey and M. L. Scott, ‘‘Synchronization Without Contention,’’ *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 269-278.

[17] J. M. Mellor-Crummey and M. L. Scott, ‘‘Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors,’’ *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, April 1991, pp. 106-113.

[18] Y. Sagiv, ‘‘Concurrent Operations on B*-Trees with Overtaking,’’ *Journal of Computer and System Sciences 33*:2 (October 1986), pp. 275-296.

[19] 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 (April 1987), pp. 388-395.