International Journal of Parallel Programming, Vol. 18, No. 3, 1989

A Scalable Implementation of Barrier Synchronization Using an Adaptive Combining Tree

Rajiv Gupta[1] and Charles R. Hill[1]

Barrier synchronization is commonly used for synchronizing processors prior to a join operation and to enforce data dependencies during the execution of parallelized loops. Simple software implementations of barrier synchronization can result in memory hot-spots, especially in large scale shared-memory multiprocessors containing hundreds of processors and memory modules communicating through an interconnection network. A software combining tree can be used to substantially reduce memory contention due to hot-spots. However, such an implementation results in $O(\log n)$ latency in recognition of barrier synchronization, where n is the number of processors. In this paper an *adaptive software combining tree* is used to implement a scalable barrier with $O(1)$ recognition latency. The processors that arrive early at the barrier adapt the combining tree so that it has a structure appropriate for reducing the latency for the processors that arrive later. We also show how adaptive combining trees can be used to implement the fuzzy barrier. The fuzzy barrier mechanism reduces the idling of processors at the barriers by allowing the processors to execute useful instructions while they are waiting at the barrier.

KEY WORDS: Memory hot spots; software combining tree; fuzzy barrier; interconnection networks; processor synchronization.

1. INTRODUCTION

Barrier synchronization is a commonly used mechanism for synchronizing the flow of control of two or more parallel threads of execution. Upon reaching a barrier a processor must wait until all processors reach the

[1] Philips Laboratories, North American Philips Corporation, 345 Scarborough Road, Briarcliff Manor, New York.

0885-7458/89/0600-0161$06.00/0

828/18/3-1

barrier. Barriers may be automatically introduced by a parallelizing compiler or may be introduced explicitly by the programmer. They can be used for synchronizing processors to perform a join operation when exploiting parallelism in a fork-join fashion. During the parallel execution of loops, a barrier placed at the end of a DOALL loop enforces data dependencies between successive executions of the DOALL. Barriers can be implemented in software using one or more shared variables and several algorithms have been proposed to do so.[1–3] In this paper, we address the problem of developing a software implementation of barrier synchronization suitable for a large scale shared-memory multiprocessor system. A large scale shared-memory multiprocessor contains hundreds of processors and memory modules. A multistage interconnection network, such as the omega network,[4] is typically provided as a means for communication between the processors and memory modules. The Cedar machine at the University of Illinois,[5] the NYU Ultracomputer,[6] BBN Butterfly, and the IBM RP3[7] are examples of such machines.

The barriers can be implemented by an atomic counter decrement followed by a busy wait. Although such an implementation may be adequate in a system with a small number of processors it suffers from two major drawbacks if used in a large scale system. First, it results in memory hot-spots, which prevents its use for synchronizing a large number of processors. Second, the busy waiting of processors at the barrier wastes processor resources. A software solution to the hot-spot problem was proposed by Yew *et al.*[3] Instead of using a single variable, a combining tree consisting of several shared memory variables is built. Each of these variables is only accessed by a subset of processors during synchronization. By distributing the variables among different memory modules in the system the problem of memory hot-spots is greatly reduced. Thus, this approach is suitable for synchronizing a large number of processors. However, such an implementation involves latency in detecting the occurrence of synchronization. If one of the processors arrives at the barrier later than all other processors it requires $O(\log_d N)$ time to recognize that it is the last processor, to arrive at the barrier, where d is the number of children of each node in the tree and N is the number of processors. In this paper, we present a software implementation of the barrier that eliminates the latency in the recognition of barrier synchronization by employing an *adaptive combining tree.*

A solution to reduce idling of processors at the barriers has been proposed.[8] The compiler finds useful instructions that can be executed by the processor while it is waiting for notification of synchronization. After having finished executing these instructions a processor checks whether all other processors have arrived at the barrier. If this is true the processor can

continue execution and thus avoid busy waiting at the barrier. The reduction in the busy waiting not only leads to better processor utilization but also reduces the memory traffic due to busy waiting. A hardware implementation of the fuzzy barrier was proposed to avoid synchronization overhead and memory contention.[8] We present a software implementation of the fuzzy barrier using adaptive combining trees that results in an implementation that is scalable, involves low latency, and reduces idling at barriers.

In the subsequent sections we first describe the machine model used in this work and the implementation of barrier synchronization using the traditional combining tree. Next the low latency implementation of barrier synchronization based upon an adaptive combining tree is discussed in detail. Finally the implementation of the fuzzy barrier using adaptive combining trees is described.

2. THE MACHINE MODEL

In this section we specify the machine model of a large scale shared-memory multiprocessor system. The system consists of a large number of identical processors and memory modules. The memory modules are shared by all the processors in the system. The processors and the shared memory modules are connected by a multistage interconnection network. We assume that the interconnection network does not perform combining as combining networks are slow and expensive to build. Each processor also has its local memory where local variables are stored. A processor does not have to go through the interconnection network to access its local memory.

The machine also supports synchronization instructions that are executed atomically. These instructions perform read-test-modify operations on variables in shared memory. The implementations of the adaptive combining tree and the fuzzy barrier described in this paper require the use of spinlocks that can be implemented using read-test-modify operations. For convenience we use the synchronization instructions supported by the Cedar machine[5] to express synchronization operations. The Cedar synchronization instructions have the following form:

$$\langle \text{syncvar; test; oper} \rangle$$

where *syncvar* is an integer synchronization variable allocated in shared memory, *test* is a condition that is tested prior to performing the operation *oper* on the synchronization variable. If the test fails the operation is not performed and the outcome of the test is sent to the processor. The

processor can issue the same instruction again or proceed with execution. A star on the test condition (*test**) is used to indicate that the processor will continue to issue the instruction till it succeeds. The following instructions perform a lock and unlock operation on a spinlock *spin*.

lock: ⟨spin; (=0)*; Increment⟩

unlock: ⟨spin; Null; Decrement⟩

The integer synchronization variable *spin* is initialized to zero. The lock instruction repeatedly tests the value of *spin* till it is zero at which point it increments *spin*. The unlock instruction decrements *spin* unconditionally.

3. BARRIER IMPLEMENTATION USING A SOFTWARE COMBINING TREE

In this section we present a barrier implementation based upon the software combining trees proposed by Yew *et al.*[3] Figure 1 shows a combining tree for synchronizing N processors. The nodes of the tree represent variables allocated from different memory modules in the system. Each node contains a *parent* pointer, a *counter* that is initialized to d, which is the number of children of each node in the tree, and a *notify* field used during the notification of synchronization. For simplicity we assume that the number of processors (N) synchronizing at the barrier is an integral power of d (i.e., $N = d^k$). A processor upon arriving at the barrier goes to the leaf node assigned to it and decrements the *counter*. If the counter is not zero there are other processors that have not reached the barrier and the processor remains at that node and busy waits on the *notify* field. If the counter is zero then it is the last processor to arrive at the node and it goes to the parent node and repeats the above process. When a processor decrements the counter at the root node to zero, barrier synchronization

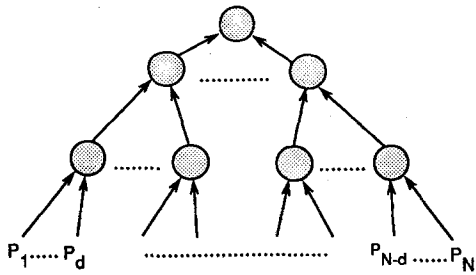


Fig. 1. Combining tree.

has occurred and the notification through the *notify* fields is carried out. Each node in the combining tree is accessed by at most d processors and the nodes are assigned to different memory modules. Thus, the memory traffic required for barrier synchronization is distributed among the memory modules. The pseudocode for the barrier implementation follows.

```
type node = record {
                counter: syncvar;/* initial value = d */
                notify: syncvar; /* initial value = 0 */
                parent: ^node;
              }

Procedure Barrier ( node );
  {
   <node⟶counter; Null; Fetch(last)&Decrement>
   if ( last == 1 ) {
     /* d processors have arrived at node */
     if ( node != root ) Barrier(node.parent);
     /* all processors have arrived at the barrier - begin notification */
     node⟶notify = d-1; /* notify siblings */
     /* wait for all siblings to notice */
     while (node⟶notify != 0);
     /* reinitialize the current node */
     node⟶counter = d;
    }
   else {
        /* wait for notification and indicate receipt of notification */
        <node⟶notify; (>0)*; Decrement>
        /* wait for all siblings to notice */
        while (node⟶notify != 0);
       }
  }
```

A processor upon reaching the barrier calls the procedure *Barrier* passing as parameter the node at the lowest level in the combining tree assigned to it. In this implementation the combining tree is reinitialized when the processors leave the barrier. Thus, it can be reused repeatedly as is often required when barriers are used during parallel execution of loops. To ensure that a processor does not encounter an uninitialized node upon re-entry in to the barrier, the processor waits at a node until all processors busy waiting at that node have received the notification. The operation *Fetch(last)&Decrement* decrements the *counter* and returns the value of the *counter* prior to the decrement in the parameter *last*. The caller can then check the value of *last* to determine whether it was the last processor to arrive at the node. Decrementing of the *notify* field to indicate receipt of notification is also carried out atomically. It should be noted that for $d = 2$,

i.e., a binary tree, the counter and notify variables can be replaced by a flag and an atomic test-and-set operation suffices.

3.1. Time Complexity Analysis

Barrier synchronization can be considered to consist of two phases, *recognition* and *notification.* During the recognition phase the arrival of processors is noted to determine whether all processors have reached the barrier. During the notification phase all processors are notified about the occurrence of synchronization so that they can continue execution.

Recognition: Two cases need to be considered here. The first case deals with the situation in which all processors arrive at the barrier simultaneously and the second case arises when one of the processors arrives later than all other processors. Two processors are considered to have arrived at the barrier *simultaneously* if one of them arrives before the other enters the busy waiting stage.

(a) *Simultaneous arrival*—In this case the time to achieve barrier synchronization is $O(d \log_d N)$. This is because the d processors arriving at a node must decrement *counter* one at a time. Since there are $\log_d N$ levels in the tree the total time spent in synchronizing is $O(d \log_d N)$.

(b) *Non-simultaneous arrival*—If all but one of the processors has already arrived at the barrier then the last processor must decrement the counters from the lower most level to the root of the tree to detect synchronization. This takes $O(\log_d N)$ time. Thus, there is a latency of $O(\log_d N)$ in the detection of barrier synchronization after the last processor has arrived at the barrier.

Notification: Notification to all processors takes $O(d \log_d N)$ time. This is because the processor that reaches the root of the tree has to go through $\log_d N$ levels notifying the processors and at each level it ensures that each of the $d-1$ processors receives the notification.

From this analysis it is clear that the choice of d involves a trade-off between optimizing the recognition time for the simultaneous arrival case and the non-simultaneous arrival case. By making the value of d small, the recognition time in the simultaneous arrival case can be reduced. But at the same time, the number of levels in the tree increases, thus causing the latency for detection of barrier synchronization in the non-simultaneous arrival case to go up. This trade-off was also reported by Yew *et al.*[3] in their work.

4. BARRIER IMPLEMENTATION USING AN ADAPTIVE COMBINING TREE

In this section, we present a barrier implementation based upon an adaptive combining tree. This implementation eliminates the latency for barrier recognition in the non-simultaneous arrival case. Therefore, a binary tree is used to minimize the recognition time in the simultaneous arrival case to $O(\log_2 N)$. The notification process is also modified, resulting in a barrier implementation that requires $O(\log_2 N)$ time each for recognition in the simultaneous arrival case and performing the notification. The barrier is correctly reinitialized, thus allowing its repeated use in synchronization.

To understand the approach to elimination of latency, let us study the cause of the latency in the implementation described in the previous section. Consider the situation in which a binary combining tree is being used to synchronize four processors. In addition let us assume that the order in which the processors arrive at the barrier is P_1, P_2, P_3, and P_4 respectively. The state of the combining tree before the arrival of P_4 is shown in Fig. 2(a). The nodes of the tree are labeled by the processors busy-waiting at the nodes. When processor P_4 arrives, starting from the bottom of the tree, it has to go to the root of the tree to recognize synchronization. If the order of arrival was known prior to recognition the tree in Fig. 2(b) would have been more suitable. Since no processors arrived in parallel, the tree need not be organized to exploit parallelism. Instead it is organized so that no processor has to visit multiple levels in the tree. When processor P_4 arrives it recognizes synchronization immediately after decrementing the counter at the root node. The trees shown in Fig. 2 represent the two extreme cases. The tree in Fig. 2(a) is organized to exploit maximum parallelism during synchronization and the tree in Fig. 2(b) is organized to minimize latency. For a given arrival pattern the most suitable tree is one

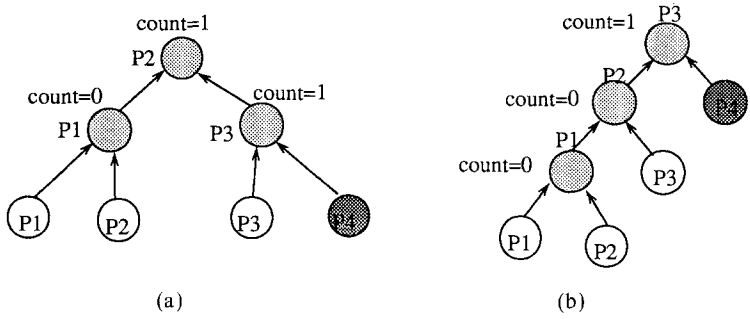


Fig. 2. Reason for latency.

which exploits the parallelism in synchronizing the processors that arrive simultaneously and minimizes the latency for the processors that arrive serially.

The order in which the processors arrive at the barrier is not known prior to execution. Furthermore, this order may not be the same during every execution. Thus, the appropriate tree structure cannot be constructed statically. However, we propose the use of an *adaptive combining tree* to achieve the appropriate tree structure dynamically. The combining tree is originally organized as a binary tree so that it can exploit maximum parallelism during synchronization if the processors arrive simultaneously. However, the processors that arrive early modify the tree so that it has a structure appropriate for reducing the latency for the processors that arrive later. For the processor arrivals in the example of Fig. 2, we will start out with the tree in Fig. 2(a) but achieve the effect of having the tree in Fig. 2(b) if the processors arrive serially.

The adaptive binary tree is illustrated through an example in Fig. 3. Figure 3(a) shows the initial combining tree. The parent link of the root node is *nil*. Thus, all processors have reached the barrier when a processor trying to go up the tree encounters a *nil* parent link. Figure 3(b) shows the tree after P_1 and P_4 have arrived at the barrier. The links have been modified so that P_2 and P_3 have node 2 as their parent. Let us assume P_2 arrives next. Processor P_2 goes directly to node 2 and modifies the tree so that the node corresponding to processor P_3 points to the root node, as shown in Fig. 3(c). Thus, when P_3 arrives it goes directly to the root of the tree and modifies the tree so that the parent of node 3 is *nil*. When processors P_5, P_6, P_7, and P_8 arrive at the barrier they do not have to go all the way to the original root of the tree. A processor that arrives at the barrier after all other processors have modified the tree will have a *nil* parent and therefore will recognize the occurrence of synchronization immediately. The example presented illustrates how processors that arrive earlier adapt the tree so that the processors that arrive later can go directly to higher levels in the tree. At the same time starting out with a binary tree will allow for maximum parallelism during synchronization if all processors do arrive at the barrier simultaneously.

The adaptive combining tree is implemented as a binary tree. Each node in the tree contains two sets of fields, the *binary* fields that form the binary tree and the *current* fields which form the current tree. The current fields, *parent*, *left*, and *right*, are provided for the purposes of traversing the tree in the recognition phase. The binary fields, *bin_parent*, *bin_left*, and *bin_right*, always form a binary tree. The current fields are initialized to form a binary tree and are modified during the synchronization process. After barrier synchronization is completed the current fields are

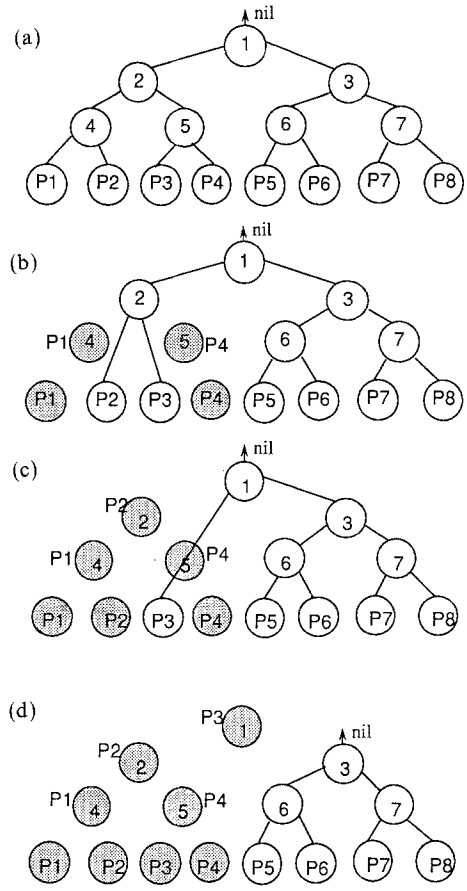


Fig. 3. Adapting the combining tree to avoid latency.

reinitialized using the binary fields so that the combining tree can be used repeatedly.

The basic rule that was being used to adapt the current combining tree is shown in Fig. 4. In Fig. 4 the *current* fields of the nodes are shown inside the node and the *binary* fields are shown by dotted lines. Figure 4(a) shows a situation in which the last processor from the subgraph S arrives at the barrier and visits node n_j, which has not been visited by another processor. The other child node of n_j (i.e., the node not belonging to S), denoted by n_k, has not been visited earlier. The parent link of node n_k is modified to point to node n_i, the parent of node n_j. Also node n_k is now made the new left child of n_i. Fig. 4(b) shows that n_k may not necessarily

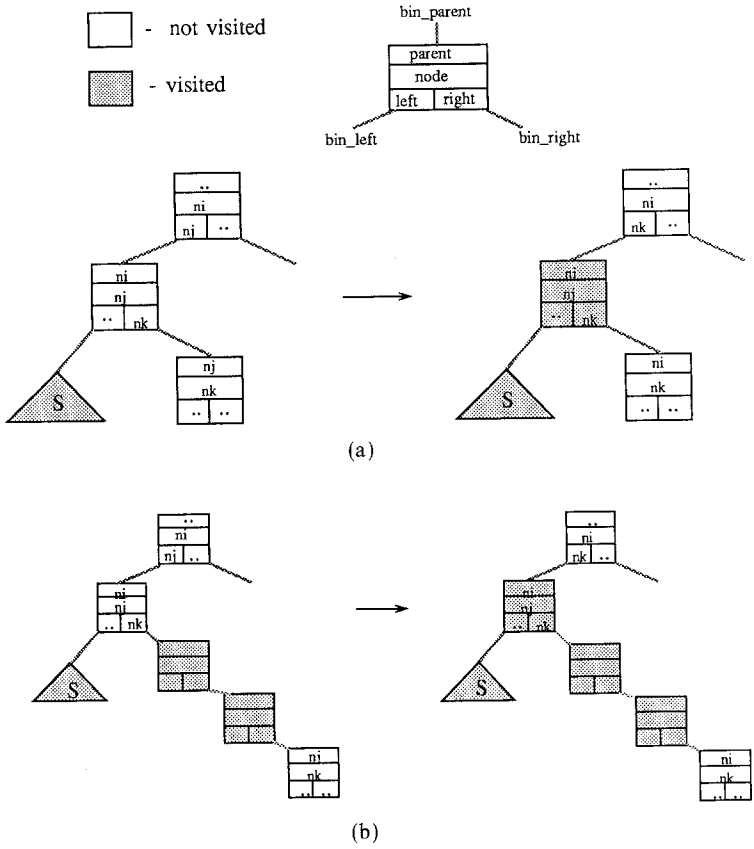


Fig. 4. Tree adaptation.

be the immediate right child of n_j in the binary tree. However, the nodes between n_j and n_k must have been visited earlier and bypassed through the earlier modifications to the current tree.

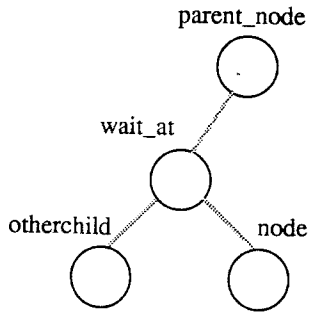
Next, we discuss how notification of synchronization is carried out. After a processor visits an unvisited node, it marks the node visited, carries out the modifications to the current combining tree and then busy waits at the node for notification. Thus if all but one of the processors have arrived at the barrier then there is a single processor busy waiting at each of the internal nodes of the current combining tree. The last processor upon arrival will have a *nil* parent link, indicating that synchronization has occurred. At this point it notifies the processor waiting at the root of the

binary tree by setting the *notify* flag of the root node. Each processor is provided with a pointer to the root node so that it can set the *notify* flag for the root node. The processor busy waiting at the root notifies the processors waiting at its child nodes in the binary tree. This notification continues down to the leaves. Thus using the binary combining tree the notification process is carried out in parallel.

As in the previous algorithm the tree must also be reinitialized so that it can be reused. Each processor is responsible for reinitializing the node at which it busy waits. Thus, after receiving notification it reinitializes the node. However, if this approach is taken a processor can leave and hence reenter the barrier before all the nodes have been reinitialized. To avoid this problem we propose the use of two combining trees (say T_1 and T_2) to implement a single barrier. Each tree is used during alternate synchronizations. If a processor has exited T_1 and is entering T_2, then T_2 must have been reinitialized. This is because a processor can only leave T_1 after all processors have entered T_1, which guarantees that all processors have exited T_2. Thus T_2 must have been reinitialized. Each processor keeps track of the tree it is currently using. When the processor exits a tree it switches its current tree.

The pseudocode for the implementation of the adaptive combining tree is given next. The three main parts to the synchronization process are indicating arrival at the barrier, modifying the combining tree, and waiting for notification. As mentioned earlier, each node contains current fields and binary fields which are used during the recognition phase and notification phases, respectively. A lock is provided that allows mutual exclusion when a node is being marked visited and during the modification of the parent link. A flag *notify* is provided which is continually checked by the processor busy waiting at the node for notification. The *visited* field indicates whether the node has not yet been visited by a processor (=no), or it has been visited by a processor from the left/right subtree (=left/right). The indication as to whether a node has been visited from the left subtree or the right subtree is saved so that the child node whose parent pointer is to be modified can be conveniently determined. The function *Reinitialize(node)* is used to reinitialize *node* prior to exiting the barrier.

```
type node = record {
            lock: syncvar; /* Initially 0 */
            visited: (no, left, right);
            root, bin_left, bin_right, bin_parent: ^node;
            left, right, parent: ^node; /* Initially bin_left, bin_right, and
                                           bin_parent respectively */
            notify: boolean; /* Initially false */
          }
```




```
Procedure Barrier ( node )
 {
    /* Arrival Phase */
    Loop {  /* Loop until parent field of node is nil or unvisited */
      wait_at = node⟶parent;
      if (wait_at == nil) {
         node⟶root⟶notify = true; /* Last to arrive - begin notification */
         Reinitialize(node); /* reset left, right, parent, visited, and notify */
         return;
      }
      <wait_at⟶lock; (=0)*; increment>
      if (wait_at⟶visited == no) break;
      <wait_at⟶lock; Null; Decrement>
    }
    Reinitialize(node);
    /* Now wait_at is unvisited. Mark it visited, and drop the lock */
    wait_at⟶visited = if (wait_at⟶left==node) left; else right;
    <wait_at⟶lock; Null; Decrement>


    /* Tree Modification Phase */
    Loop {
      otherchild = if (wait_at⟶visited=left) wait_at⟶right;
                   else wait_at⟶left;
      <otherchild⟶lock; (=0)*; Increment>
      if (otherchild⟶visited==no) break;
      <otherchild⟶lock; Null; Decrement>
    }
    /* Now otherchild is unvisited, and its lock is held */
    parent_node = wait_at⟶parent;
    if (parent_node != nil) {
      if (parent_node⟶left==wait_at)
        parent_node⟶left = otherchild;
      else parent_node⟶right = otherchild;
    }
    otherchild⟶parent = parent_node;
    <otherchild⟶lock; Null; Decrement>
    while (not wait_at⟶notify); /* Wait for notification */
```

```
    /* Notification Phase */
    if (not Leaf (wait_at→bin_left)) {
       /* Notify the binary children */
       wait_at→bin_left→notify = true;
       wait_at→bin_right→notify = true;
     }
    Reinitialize(wait_at); /* reset left, right, parent, visited, and notify */
}
```

In the arrival phase a processor (P) continually examines the *visited* field of its current parent node. If the node is unvisited the processor marks the node visited and moves to the tree modification phase. If the parent node has already been visited processor P waits for its parent node to change to an unvisited node or nil. This will eventually happen because the processor that visited the parent node must be in the process of modifying the parent pointer of the node at which P is waiting. In the tree modification phase a processor determines the node *otherchild* whose parent pointer is to be modified. If *otherchild* is an unvisited node its parent pointer is immediately modified. If this is not the case the processor must wait till its *otherchild* is modified to point to a unvisited node. This will eventually happen because the processor that visited *otherchild* will modify the tree appropriately in its tree modification phase. After tree modification a processor waits for notification. The last processor to arrive finds its parent pointer to be *nil* and starts notification at the root of the tree.

4.1. Time Complexity Analysis

The time for synchronization using an adaptive combining tree is as follows:

Recognition: (a) *Simultaneous arrival*—The time for recognition is $O(\log_2 N)$. This is because in the worst case a processor that discovers synchronization has its pointer changed $\log_2 N$ times till it is *nil*. (b) *Non-simultaneous arrival*—The parent pointer for the last processor is already *nil* when it arrives at the barrier. Thus it takes constant ($O(1)$) time to recognize synchronization.

Notification: This is being carried out in parallel using the binary tree and therefore it takes $O(\log_2 N)$ time.

5. FUZZY COMBINING TREE

The waiting of processors at barriers can be reduced by using the fuzzy barrier mechanism.[8] The compiler constructs barrier regions consisting of

several instructions such that a processor is ready to synchronize upon reaching the first instruction in this region and must synchronize before exiting the region. When synchronization occurs the processors could be executing at any point in their respective barrier regions. The processors can potentially arrive at the barrier at different times and exit the barrier at different times without busy waiting or stalling. If a processor reaches the end of the barrier region and tries to execute a non-barrier region instruction before synchronization has taken place, the processor is stalled. The larger the barrier region the more likely it is that none of the processors will have to stall. Compile-time techniques to find useful instructions that can be executed by a processor after it is ready to synchronize also exist.[8]

A software fuzzy barrier can be made available by providing two procedure calls *EnterBarrier* and *ExitBarrier*. The example shown in Fig. 5 demonstrates the use of the fuzzy barrier in comparison to the fixed barrier for executing iterations of a loop in parallel. The barrier is required due to the cross iteration data dependency between statements $S1$ and $S4$. Since statements $S2$ and $S3$ are not involved in cross iteration dependencies they can be executed prior to synchronization as part of the barrier region.

Next we present an implementation of the fuzzy barrier based upon the adaptive combining tree discussed in the previous section. At first it may seem that the adaptive tree implementation may be used for implementing the fuzzy barrier by simply having each processor execute the instructions in the barrier region prior to busy-waiting for notification. The problem with this approach is illustrated in Fig. 6. The combining tree in Fig. 6 shows a situation in which all processors have reached the barrier and P_4, being the last to arrive, has to start the notification process. It will do so by informing P_2 of the synchronization. However, it may be the case that P_2 is still busy executing its barrier region and therefore will not notice the notification till later. At the same time processors P_1 and P_3 may have completed execution of their barrier regions and thus they will have to wait unnecessarily for notification. The cause of this problem is that the solution

```
Doall I = 1 to N
  S1: A(I) = B(I) + C(I);
  Barrier;
  S2: D(I) = B(I) - C(I);
  S3: E(I) = D(I) + 1;
  S4: F(I) = A(I-1) + A(I);
Enddoall
```

```
Doall I = 1 to N
  S1: A(I) = B(I) + C(I);
  EnterBarrier;
  S2: D(I) = B(I) - C(I);
  S3: E(I) = D(I) + 1;
  ExitBarrier;
  S4: F(I) = A(I-1) + A(I);
Enddoall
```

Fig. 5. Fixed versus fuzzy barrier.

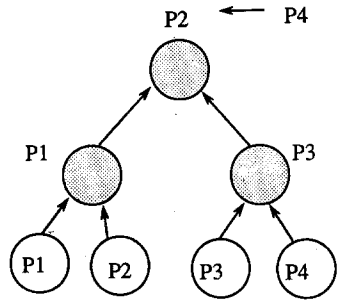


Fig. 6. Problem with notification.

assumes that all processors will exit the barrier at the same time. This is true for the fixed barrier but not in the case of the fuzzy barrier. In a fuzzy barrier synchronization, the processors can enter the barrier in any order and also exit the barrier in any order. Furthermore, the order in which processors enter the barrier may not be the same as the order in which they exit the barrier.

The solution to this problem becomes evident if we use separate tree traversals for implementing of the fuzzy barrier operations *EnterBarrier* and *ExitBarrier*. *EnterBarrier* performs the recognition and tree adaptation phases and *ExitBarrier* performs the notification phase. In each of the operations the processors must start at the bottom of the tree. Instead of busy waiting at the internal node reached by a processor during the recognition phase, it must start again at the bottom of the tree and go up the tree looking for notification. We illustrate this process through a four processor example. Let us assume that the processors arrive in the order P_1, P_2 and P_3 respectively. Figure 7(a) shows the combining tree after it has been visited by processors P_1, P_2, and P_3. The nodes are labeled with

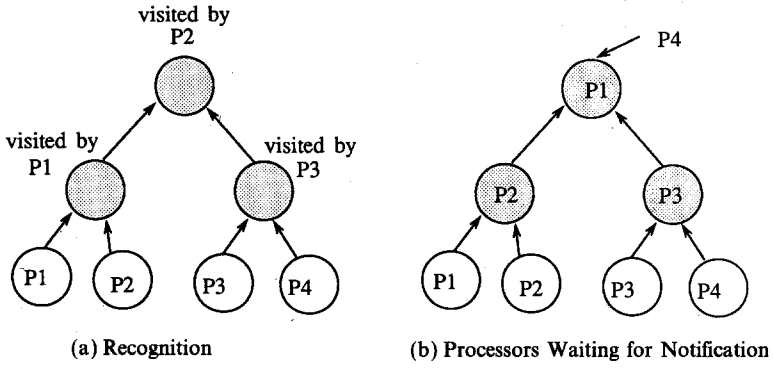
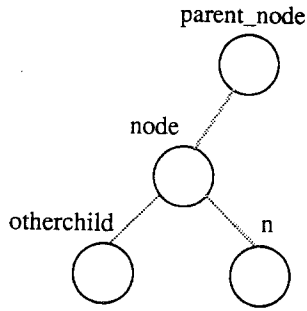


Fig. 7. Notification.

the processors that visited the nodes. After visiting the nodes the processors do not busy-wait at the nodes but instead return from the procedure *EnterBarrier* and start executing their respective barrier regions. Processor P_4, after discovering that it was the last processor to reach the barrier, sets the *notify* flag at the root node and returns to execute its barrier region. After completion of their barrier regions the processors start at the bottom of the original tree looking for notification. The processor that arrives first reaches the root node and discovers that synchronization has occurred. It then goes down the tree setting the notify flags of both binary child nodes and leaves the barrier. The processors that arrive later have to go up fewer levels and they too set the notify flags for the child nodes. Note that tree traversal during the notification phase is carried out using the pure binary tree.

If the processors arrive before synchronization has occurred they go to the highest unoccupied node and busy wait. At most one processor busy-waits at an internal node. Thus, there is no hot spot problem during the notification phase. Figure 7(b) shows the combining tree in the situation where prior to the arrival of P_4 at the barrier, the remaining processors have executed their barrier regions and are busy waiting for notification. It should also be noted that nodes at which the processors are busy waiting are not the same as the nodes they visited in the notification phase. This is because the processors may not execute *ExitBarrier* in the same order as they executed *EnterBarrier*, as they may take varying amounts of time to execute their barrier regions. After P_4 arrives at the barrier notification proceeds in the usual manner. The pseudocode for the implementation of the fuzzy barrier is given here.

```
type node = record {
            lock: syncvar; /* Initially 0 */
            visited: (no, left, right);
            LastVisitBy: integer; /* processor that reinitializes the node */
            root, bin_left, bin_right, bin_parent: ^node;
            left, right, parent: ^node; /* Initially bin_left, bin_right, and
                                           bin_parent, respectively */
            notify: boolean; /* Initially false */
          }
```



```
Procedure EnterBarrier ( n )
 {
    Loop {  /* Loop until parent field of n is nil or unvisited */
      node = n⟶parent;
      if (node == nil) {
         n⟶root⟶notify = true; /* Last to arrive - begin notification */
         return;
      }
      <node⟶lock; (=0)*; increment>
      if (node⟶visited == no) break;
      <node⟶lock; Null; Decrement>
    }
    /* Now node is unvisited. Mark it visited, and drop the lock */
    node⟶visited = if (node⟶left == n) left; else right;
    <node⟶lock; Null; Decrement>
    Loop {
      otherchild = if (node⟶visited=left) node⟶right;
                   else node⟶left;
      <otherchild⟶lock; (=0)*; Increment>
      if (otherchild⟶visited == no) break;
      <otherchild⟶lock; Null; Decrement>
    }
    /* Now otherchild is unvisited, and its lock is held */
    parent_node = node⟶parent;
    if (parent_node != nil) {
      if (parent_node⟶left == node) parent_node⟶left = otherchild;
        else parent_node⟶right = otherchild;
    }
    otherchild⟶parent = parent_node;
    <otherchild⟶lock; Null; Decrement>
 }
Procedure ExitBarrier ( node )
    {
       /* the processor to visit a node last will reinitialize it */
       if Leaf(node) <node⟶lock; (=0)*; Increment>;
       node⟶LastVisitBy = ProcessorId();
       if (node⟶notify) <node⟶lock; Null; Decrement>;
       else {
            /* climb the tree looking for notification */
            parent_node = node⟶bin_parent;
            if (parent_node == nil) {
              /* reached the root, cannot climb further */
              node⟶occupied = true;
              <node⟶lock; Null; Decrement>
              while (not node⟶notify);
             }
            else {
                 <parent_node⟶lock; (=0)*; Increment>
                 if (not parent_node⟶occupied) {
                   /* drop lock on node continue to climb */
                   <node⟶lock; Null; Decrement>
                   ExitBarrier(parent_node);
                  }
```

```
            else {
                /* cannot climb further, wait at node */
                <parent_node⟶lock; Null; Decrement>
                node⟶occupied = true;
                <node⟶lock; Null; Decrement>
                while (not node⟶notify);
              }
          }
      }
    if (node⟶LastVisitBy != ProcessorId()) while (node⟶notify);
    if (not Leaf(node)) {
        /* notify binary child nodes */
        <node⟶bin_left⟶lock; (=0)*; Increment>
        if (node⟶LastVisitBy == ProcessorId()) node⟶bin_left⟶notify = true;
        <node⟶bin_left⟶lock; Null; Decrement>
        <node⟶bin_right⟶lock; (=0)*; Increment>
        if (node⟶LastVisitBy == ProcessorId()) node⟶bin_right⟶notify = true;
        <node⟶bin_right⟶lock; Null; Decrement>
      }
    if (node⟶LastVisitBy == ProcessorId()) Reinitialize( node )
}
```

The procedure *EnterBarrier* implements the recognition phase and the tree modification in exactly the same manner as the fixed barrier implementation of the previous section. However, the *ExitBarrier* code is different from the notification phase of the barrier as the processors must retraverse the tree to receive notification. Prior to leaving the barrier each processor reinitializes the node at which it received notification. During the *EnterBarrier* operation the tree is adapted and therefore it is traversed using the *current* fields of the nodes. During the notification phase the *binary* fields are used to traverse the tree. The *current* fields are not needed for traversal during *ExitBarrier* because the tree is not being adapted. The function *ProcessorId*( ) returns the calling processor's *id.* During the notification phase in *ExitBarrier*, the processor that visits a node last is responsible for reinitializing the node. The identification of the last processor to visit a node is saved in the field *LastVisitBy*.

5.1. Time Complexity Analysis

The time for synchronization using the fuzzy barrier is as follows:

Recognition: The implementation of the recognition and tree adaptation phases is exactly the same as the algorithm described in the previous section. Therefore in the *simultaneous arrival* case the time for recognition is $O(\log_2 N)$ and in the *non-simultaneous arrival* case recognition takes constant time.

Notification: The time for notification depends upon whether the synchronization has already been detected prior to the completion of barrier regions by the processor.

(a) *Synchronization has Occurred*—If synchronization has already occurred then the processor that completes its barrier region first will require $O(\log_2 N)$ to receive notification as it must climb to the root of the tree. The processor that completes the execution of its barrier region last will require constant time to receive notification.

(b) *Synchronization has not Occurred*—If synchronization has not occurred the processors must wait at the nodes in the tree. After synchronization occurs the processor waiting at the root of the tree receives the notification immediately, while the processors at lower levels of the tree have to wait longer for notification. Since the notification is carried out using the binary tree, the processors at the bottom of the tree have to wait for $O(\log_2 N)$ time.

6. SUMMARY

In this paper we introduced *adaptive combining trees* to allow efficient implementation of barrier synchronization. An adaptive tree enables the exploitation of parallelism, available in the synchronization process, in situations where the processors arrive at the barrier simultaneously. At the same time if the processors do not arrive at the barrier simultaneously, the tree is adapted so that the latency in recognizing synchronization for the late arriving processor is avoided. We also presented an implementation of the fuzzy barrier that reduces the idling of processors at a barrier. This provides us with an implementation that avoids latency and reduces processor idling, the two main problems with existing scalable implementations of the barrier mechanism.

REFERENCES

1. E. D. Brooks, The Butterfly Barrier, *International Journal of Parallel Programming*, **15**(4):295–307 (August 1986).
2. D. Hansgen, R. Finkel, and U. Manber, Two Algorithms for Barrier Synchronization, *International Journal of Parallel Programming*, **17**(1):1–18 (February 1988).
3. 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 (April 1987).
4. D. H. Lawrie, Access and Alignment of Data in an Array Processor, *IEEE Transactions on Computers*, **C-24**:1145–1155 (December 1975).
5. D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, Parallel Supercomputing Today and the Cedar Approach, *Science*, **231**:967–974 (February 1986).

6. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, The NYU Ultracomputer-Designing a MIMD Shared Memory Parallel Machine, *IEEE Transactions on Computers*, **C-32**(2):175–189 (February 1983).
7. G. F. Pfister, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, In *Proc. of the International Conf. on Parallel Processing*, pp. 764–771 (August 1985).
8. R. Gupta, The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors, In *Proc. of the Third International Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 54–64 (April 1989).