p. 23

NASA Contractor Report 189684
ICASE Report No. 92-34

ICASE

OPTIMISTIC BARRIER SYNCHRONIZATION

David M. Nicol

28P

Contract No. NAS1-18605
July 1992

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

OPTIMISTIC BARRIER SYNCHRONIZATION

David M. Nicol[1]
Department of Computer Science
College of William and Mary
Williamsburg, VA 23185

ABSTRACT

Barrier synchronization is a fundamental operation in parallel computation. In many contexts, at the point a processor enters a barrier it knows that it has already processed all work required of it prior to the synchronization. This paper treats the alternative case, when a processor cannot enter a barrier with the assurance that it has already performed all necessary pre-synchronization computation. The problem arises when the number of pre-synchronization messages to be received by a processor is unknown, for example, in a parallel discrete simulation or any other computation that is largely driven by an unpredictable exchange of messages. We describe an optimistic $O(\log^2 P)$ barrier algorithm for such problems, study its performance on a large-scale parallel system, and consider extensions to general associative reductions, as well as associative parallel prefix computations.

[1]This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665. Research was also supported in part by NASA grants NAG-1-1060, NAG-1-1132, and NAG-1-995, and NSF Grants ASC 8819373 and CCR-9201195.

1 Introduction

Consider a computation where the processing is driven in whole or in part by the receipt, processing, and generation of messages. An important motivating example is parallel discrete-event simulation, where a message represents an event whose eventual execution may lead to the generation of further events, possibly on other processors; however, distributed algorithms in general are often characterized this way. The run-time behavior of such computations may be highly unpredictable, which creates a problem if one desires to employ a barrier synchronization. A processor ought not synchronize until it has processed all messages sent to it by processors prior to their own synchronization, yet this is difficult if the message generation activity is unpredictable. In a parallel simulation whose synchronization is based on windows [13, 11, 5, 14], for example, one synchronizes all processors at the upper edge of the time window, say time t. If the simulation within the window is managed optimistically (e.g., using Time Warp[6]), then a processor that has simulated all known workload up to time t may receive a message associated with simulation time $s < t$ and be forced to roll back. In the course of re-executing events in time interval $[s, t]$ the processor may send new messages to other processors who also appear to have already simulated to t, causing them to roll back as well. Traditional barrier algorithms presume that a processor entering the barrier has completed all work required of it; to call a barrier routine such as **gsync()** on an Intel iPSC multiprocessor is to lose the thread of control until all processors have entered the barrier. This is clearly undesirable if the number of messages to be received is unknown.

We show how modification of a standard algorithm (the butterfly barrier [3]) permits the use of barrier synchronization when the total number of messages to be processed by a processor prior to the barrier is unknown. There are two important elements to the algorithm. One is to permit a processor to enter the barrier optimistically, before it is certain that it is finished with its pre-synchronization work. In this our algorithm incorporates ideas from optimistic synchronization in parallel discrete event simulation. The second important element is to have each processor keep track of the number of messages it has sent to and received from each of $\log P$ sets of processors we call *shells* (there are P processors). Then, like a standard barrier algorithm, a processor advances through $\log P$ steps, where at each step it synchronizes with a specific processor. Unlike a standard barrier, two synchronizing processors exchange send/receive counts tabulated for each shell, and from this information decide whether to advance to the next synchronization step, or wait to receive and process further messages. At any time, receipt of a new computation message can roll a processor back out of the barrier altogether, or a repeated synchronization message from a previous step can also roll the synchronization processing back to that step. Our algorithm requires $O(\log P)$ space on each of P processors, and requires $O(\log^2 P)$ parallel time to execute.

The problem we pose already has at least three solutions. The concept of "virtual time" underlying optimistic synchronization in parallel discrete-event simulations provides the first. Most optimistic simulation methods employ a background calculation of the "Global Virtual Time (GVT)"[4]), essentially a point in simulation time behind which it is guaranteed that no processor will ever to be required to roll back again. A barrier of the type we desire at simulation time t can be implemented by simply requiring that a processor not proceed past time t until the GVT advances up to t. However, most optimistic simulations invoke the GVT calculation infrequently (e.g. every few seconds), as it is relatively expensive. Furthermore, the issues of who invokes GVT, when it invokes GVT, and how often it invokes GVT loom large in such an approach. One should note that barrier synchronization is not usually employed by optimistic simulations (with the exceptions of Moving Time Windows[13] and Bounded Time Warp [15]), as such systems are capable of rolling a processor back to the barrier point in the event it proceeds past it prematurely. Our algorithm has the advantages of being responsive to the immediate synchronization demands of the computation, of supporting window-based simulation approaches, and of being applicable when the computation does not otherwise synchronize on the basis of virtual time. We note in passing that our algorithm can be used to compute GVT, for example, by synchronizing globally every Δ units of simulation time. Emerging from the barrier at simulation time t a processor knows the GVT is t. A second solution is hardware-based. A synchronization network is presently under development[10] where every processor stores in a network register the least timestamp among all known events; the network computes and distributes to all processors the minimum such. This minimum provides instantaneous GVT information, so that a processor can synchronize at t by merely waiting until the GVT reaches t. A third solution is really a family of solutions. One can view completion of a barrier as the termination of a distributed algorithm. Many termination algorithms already exist [8]; however, these algorithms generally view the system as being much more loosely coupled than parallel systems. Furthermore, the complexity of these algorithms is measured in terms of numbers of messages passed, rather than time to execute. In a parallel system there is a huge performance difference between a computation that passes P messages serially, and one that passes P messages in parallel. In fact, the Bounded Time Warp algorithm [15] employs a global synchronization point in simulation time, and uses a linear-time token-passing distributed termination algorithm to implement the barrier. Nevertheless, our algorithm has similarities to the "vector algorithm" proposed by Mattern [9], in that both track the difference between messages sent and messages received. However, there are substantial differences between our approach and Mattern's. His algorithm relies on a circulating control vector with P components, that serially traverses processors; ours accumulates counts in a logarithmic fashion, and has no serial component.

The fundamental communication pattern we use is based on the butterfly barrier [3]. Students of synchronization should also read the comparative study of barriers on shared memory machines reported in [2].

The principle contribution of this paper is to identify and solve a general synchronization problem by bringing together ideas from optimistic parallel simulation, deterministic parallel synchronization, and distributed termination detection. We demonstrate that our solution has a relatively small cost, by comparing it with the barrier synchronization routines provided by a large-scale multiprocessor.

The remainder is structured as follows. Section §2 introduces some notation, and uses it to describe a standard barrier synchronization algorithm. Section §3 describes our modifications, and proves the algorithm's correctness. Section §4 evaluates the performance of our algorithm on large scale multiprocessors, Section §5 extends the method to general associative reductions, and general associative parallel prefix operations, and Section §6 summarizes this paper.

2 Background

Suppose that we can view every processor's behavior in terms of its response to messages. For example, a processor might receive one or more messages, perform some computation, and possibly send new messages as a result. The notion is quite general, encompassing scientific computations where the messages communicate data at domain partition boundaries, to parallel discrete-event simulations, where a message represents an event. A key difference between these two examples is that in the former case the message passing behavior is predictable, whereas in the latter case it is not.

A barrier synchronization is introduced into the computation when we desire that the processors synchronize globally. When the computation is performed correctly, this means that every processor will have received and processed all messages for it prior to synchronizing, and *no processor leaves the barrier until all processors have received and processed all messages for which they are responsible prior to synchronization.* A processor leaving the barrier is assured that every other processor has already received all messages, performed all work, and sent all messages that are logically required by the computation prior to the global synchronization. This point is important: optimistic parallel simulations are very closely related to the

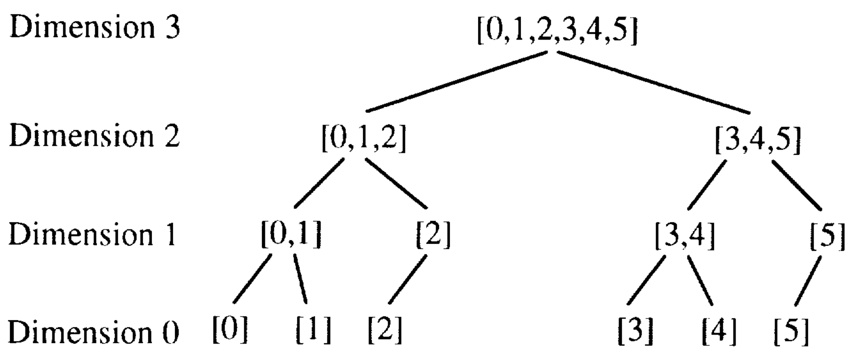

Figure 1: Balanced tree created by splitting sets of processors ids.

algorithm we propose, and yet do not provide this assurance. While our solution permits optimistic entry into the barrier, our problem forbids an optimistic departure. Upon emerging from a barrier a processor can be certain that its present state is correct.

Our problem arises in contexts other than parallel simulation. For example, consider a parallel searching algorithm that performs load balancing by having a processor generate some nodes to evaluate, select some for itself, and distribute the rest. We might wish to use a barrier to establish termination, yet a processor must be concerned about receiving additional workload *after* entering the barrier.

Next we introduce some notation. Consider a system of P processors, for any $P > 1$. Define p, the *system dimension*, to be the smallest integer such that $P \leq 2^p$. Our solution involves a balanced binary tree whose elements are sequences of processor ids. The root node is $T_0 = [0, 1, \ldots, P-1]$. Given tree node $T_c = [i, \ldots, j]$, $i \leq j$, we define T_c's left child $T_{2c+1} = [i, \ldots, \lceil (i+j)/2 \rceil]$, and its right child (applicable only if $i < j$) $T_{2c+2} = [\lceil (i+j)/2 \rceil + 1, \ldots, j]$. Thus, children sets are defined by evenly splitting a parent sequence, with the "extra" member (if any) placed in the left child. Also, we define the "dimension" of T_0 to be p, and the dimension of a child to be one less than it's parent's. The splitting process is applied until the dimension 0 sequences are defined. Figure 1 illustrates the tree associated with $P = 6$.

Let T_{2c+1} and T_{2c+2} in dimension k be children of a common parent. As these sequences are nearly balanced, we can pair their elements as follows. We say that processors i and j are *neighbors* in dimension k if for some m, i is the m^{th} largest element of T_{2c+1}, and j is the m^{th} largest element of T_{2c+2}. We denote this relationship by a function n, writing $n_k(i) = j$ and $n_k(j) = i$. For example, in Figure 1, the neighbors in dimension 2 are 0 and 3, 1 and 4, 2 and 5. When the size of two sibling sequences differs, the largest member (say j) of the left sibling has no neighbor. In this case we say that j is a *hermit* in that dimension. Also, we call the least member of any sequence the *leader* of that sequence.

Most scalable barrier algorithms employ a tree of some kind, where processors representing sibling nodes synchronize locally, and a processor representing a parent node is enabled to synchronize as soon as its own children have synchronized. One approach is to require the leader of a sequence to represent the sequence in this synchronization process. In our example, in dimension 0 we'd have 0 synchronize with 1, and 3 synchronize with 4; in dimension 1 we have 0 synchronize with 2, and 3 synchronize with 5; in dimension 2, we have 0 synchronize with 3. At any point in the barrier algorithm, if the leader of a sequence $\mathcal{S}$ is attempting to synchronize with some other processor, then we know that all processors in $\mathcal{S}$ have entered the barrier. Observe that only the processors representing T_1 and T_2 will know when all processors have entered the barrier. In this case, a broadcast step is required to notify the remaining processors. This is usually accomplished by having the leader of a tree node release the leaders of its children, who in turn release the leaders of their children, and so on.

Another approach avoids the broadcast step by requiring every processor in a tree node to determine for itself when that tree node is synchronized with its sibling. A processor synchronizes with its neighbor in dimension 0, then its neighbor in dimension 1, and so on through dimension $p-1$. If a processor i successfully synchronizes with its dimension $k-1$ neighbor, then we know that all processors in the dimension k sequence $\mathcal{S}$ containing i have entered the barrier. Thus, a processor is free to leave the barrier once it is synchronized with its neighbor in dimension $p-1$. One minor difficulty occurs if processor i in sequence $\mathcal{S}$ in dimension k is a hermit there. A solution is to have i wait to be notified by the leader of $\mathcal{S}$'s sibling, which is $i+1$. In our example, in dimension 1 we have processor 1 wait for a message from 2, and processor 4 wait for a message from 5. When this occurs, we call the leader a *messenger* in dimension k, and define $n_k(i) = i+1$. A messenger doesn't need to receive a synchronization message from its hermit, as it will synchronize with its own neighbor.

In the remainder we will call the algorithm above the *standard* barrier algorithm. A high level description is given in Figure 2. Our solution involves modification of this algorithm.

A little more notation will aid our discussion. For any processor i and dimension k, let $C_k(i)$ denote the sequence in dimension k that contains i. For any two processors i and j, define their *distance* $d(i,j) = k$ if k is the largest dimension in which i and j are not in the same sequence. The table below gives $d(i,j)$ for the case of $P = 6$.

```
Standard Barrier Synchronization Algorithm (viewed from Processor $i$)

1. $D = 0$;
2. If $i$ is a messenger in dimension $D$, send a synchronization message to $i - 1$.
3. If $i$ is not a hermit in dimension $D$, send a synchronization message to $d_D(i)$.
4. Wait until a synchronization message is received from $n_D(i)$.
5. $D = D + 1$; If $D = p$, exit;
6. goto (2)
```

Figure 2: Standard Barrier Synchronization Algorithm

$i \backslash j$	0	1	2	3	4	5
0	—	0	1	2	2	2
1	0	—	1	2	2	2
2	1	1	—	2	2	2
3	2	2	2	—	0	1
4	2	2	2	0	—	1
5	2	2	2	1	1	—

For every processor i and dimension k, define $S_k(i)$ to be the set of all processors j with $d(i,j) = k$. We call the collection of $S_k(i)$ ($k = 0, \ldots, p-1$) processor i's *shell* sets. An intuitive understanding of $S_k(i)$ is as the set of processors represented by the sibling of $[i]$'s dimension k ancestor. Another view is that $S_k(i)$ is the set of processors with whom i establishes synchronization in dimension k.

3 An Optimistic Barrier Synchronization Algorithm

The problem we pose has two components. First, we must ensure that the thread of control is not lost by calling a barrier routine, as we may have to roll back out of the barrier. Secondly, we have to ensure that no processor believes it has completed the barrier before it is certain that the processor has received all pre-synchronization messages eventually destined for it.

Even with provision for rollback, simple optimistic execution of a barrier synchronization will not ensure that a processor not leave a barrier prematurely. For example, consider a four processor system where at some time t processor 0 sends a message to processor 3 and heads into the barrier. It is quite possible for the processors to exchange synchronization messages (0 with 1 then 3, 1 with 0 then 2, 2 with 3 then 0, 3 with 2 then 1) and appear to be globally synchronized *before* the computation message from 0 is recognized by 3. Our problem formulation forbids these processors to depart the barrier, yet this is precisely what they will do if we rely only on rollback to enforce the synchronization. This example highlights the fact that a correct barrier algorithm must account for messages that are sent, but not yet received. The modifications we make to the standard algorithm do precisely that.

The remainder of the section separately addresses the problems of managing message counts, specifying the barrier algorithm, and proving its correctness.

3.1 Managing Message Counts

Our solution requires that every processor i maintain, for every shell $S_k(i)$ ($k = 0, \ldots, p-1$), a count of messages it has sent to processors in $S_k(i)$, and a separate count of messages it has received from $S_k(i)$[1]. These counts (called $Send_k(\{i\})$ and $Recv_k(\{i\})$, $k = 0, \ldots, p-1$) should include all messages relevant to the computation, but should not include the synchronization messages sent as part of the barrier implementation. Between barriers these counts increase monotonically, they are never reset as a result of rollback. Immediately following successful completion of a barrier the counts are cleared.

In the standard barrier algorithm, a single step synchronization between i and $n_k(i)$ serves to establish synchronization of two disjoint collections of processors, $C_k(i)$ and $C_k(n_k(i))$. Now suppose that processors i and $n_k(i)$ additionally exchange counts of messages sent to and received from these two sets of processors (if i is a hermit it does not send counts to $n_k(i)$). For example, suppose they detect that the total number of messages sent by processors in $C_k(i)$ to processors in $C_k(n_k(i))$ is larger than the total number of messages received by processors in $C_k(n_k(i))$ from processors in $C_k(i)$. Processors in $C_k(n_k(i))$ will eventually receive the missing messages, and be rolled back out of the barrier. Consequently neither processor i nor processor $n_k(i)$ ought to advance to the next dimension. If the two pairs of send/receive counts match as required, we will say that i and $n_k(i)$ are "in agreement" at step k.

How then can i and $n_k(i)$ have available counts of messages between $C_k(i)$ and $C_k(n_k(i))$? Observe that $S_k(i) = C_k(n_k(i))$, and that the $Send_k$ and $Recv_k$ counts in processor i and every other processor in $C_k(i)$ tabulate the number of messages sent to and received from $S_k(i)$. When i and $n_0(i)$ synchronize, they can exchange their counts relating to this set, and combine them. When i synchronizes with $n_1(i)$ it can send the combined i and $n_0(i)$ counts, and receive the combined $n_1(i)$ and $n_0(n_1(i))$ counts. Continuing in this fashion, by the time i reaches dimension k, it will have accumulated the send/receive counts of all processors

[1]Actually, one need only maintain the difference between these two counts. This optimization reduces the communication load of our algorithm; however, it is easier to explain in terms of separate counts.

in $C_k(i)$ relating to $S_k(i)$. For that matter, it can have accumulated the send/receive counts relating to all shells $S_m(k)$, $m \geq k$.

Our modified barrier algorithm hinges on the observation above. For all $k = 0, \ldots, p-1$ and $m = k, \ldots, p-1$ define $TotalSend_m(C_k(i))$ to be the total number of messages sent by processors in $C_k(i)$ to processors in $S_m(i)$; similarly define $TotalRecv_m(C_k(i))$ to be the total number of messages received by processors in $C_k(i)$ from processors in $S_m(i)$. These counts are defined to describe the situation after all pre-synchronization messages have been generated, received, and processed. Since $C_k(i)$ is the union of disjoint sequences $C_{k-1}(i)$ and $C_{k-1}(n_k(i))$, it is evident that whenever $m \geq k$:

$$TotalSend_m(C_k(i)) = \begin{cases} TotalSend_m(\{i\}) & \text{for } k = 0 \\ TotalSend_m(C_{k-1}(i)) + TotalSend_m(C_{k-1}(n_{k-1}(i))) & \text{for } k > 0 \ , \end{cases}$$

and

$$TotalRecv_m(C_k(i)) = \begin{cases} TotalRecv_m(\{i\}) & \text{for } k = 0 \\ TotalRecv_m(C_{k-1}(i)) + TotalRecv_m(C_{k-1}(n_{k-1}(i))) & \text{for } k > 0 \ . \end{cases}$$

In the course of synchronization, a processor will not necessarily know these final send/receive counts. It can only tally the numbers of messages it has seen itself with similar counts reported by other processors. We will approximate each $TotalSend_m(C_k(i))$ count with a count $Send_m(C_k(i))$ that is computed using the aggregation equations specified above (replacing each instance of *TotalSend* with a corresponding *Send*); we similarly approximate each $TotalRecv_m(C_k(i))$ with a count called $Recv_m(C_k(i))$. When processor i attempts to synchronize in dimension k, it includes in its synchronization message to $n_k(i)$ (and to $i-1$, if i is a messenger) two vectors that estimate completed send/receive counts:

$$SendVec_k(i) = \quad [Send_k(C_k(i)), \ldots, Send_{p-1}(C_k(i))],$$

and

$$RecvVec_k(i) = \quad [Recv_k(C_k(i)), \ldots, Recv_{p-1}(C_k(i))].$$

Figure 3 illustrates the information exchanged by two processors i and $n_k(i)$. Here we suppose that i is a member of the sequence labeled A, and $n_k(i)$ is in the sequence labeled B, both in some dimension k. Sets D and E are $S_{p-2}(i) = S_{p-2}(n_k(i))$, and $S_{p-1}(i) = S_{p-1}(n_k(i))$, respectively. The components of $SendVec_k(i)$ are the counts of messages sent by processors in A to processors in B, D, and E; $RecvVec_k(i)$ contains the number of messages received by processors in A from B, D, and E. Similarly, the components of $SendVec_k(n_k(i))$ are the counts of messages sent by processors in B to processors in A, D, and E; $RecvVec_k(n_k(i))$ contains the number of messages received by processors in B from A, D, and E. When i

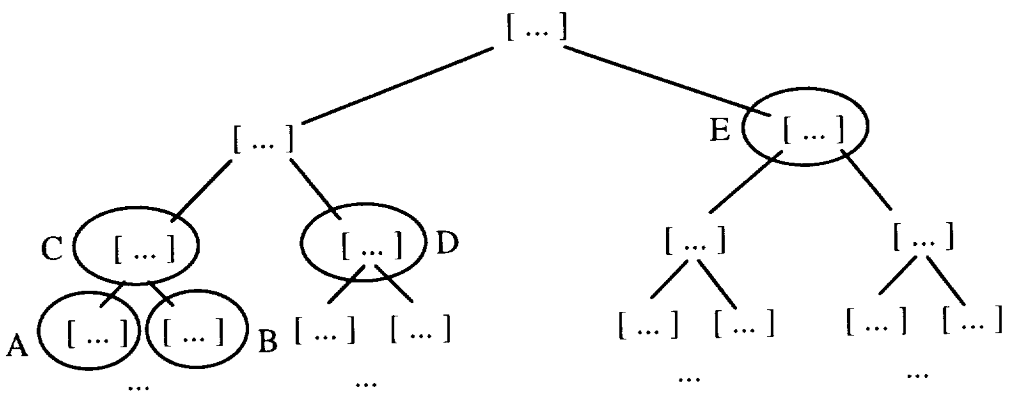

Figure 3: Graphical depiction of information passed when i (in A) synchronizes with $n_k(i)$ (in B). i gives $n_k(i)$ send/receive counts between processors in A and B, A and D, A and E. $n_k(i)$ gives i send/receive counts between processors in B and A, B and D, B and E.

and $n_k(i)$ are in agreement they may combine these values to determine the send/receive counts between processors in C and D, and between C and E.

3.2 Algorithm Specification

In our solution processor i enters the barrier logic and passes through as many dimensions as possible until it either completes, reaches a dimension k for which there is yet no synchronization message from $n_k(i)$ (or a messenger), or the message fails to indicate agreement. Upon completion failure processor i exits the barrier logic to permit receipt of further messages (either computation and synchronization messages). When a processor reenters the barrier logic it may not need to step through dimensions it has already passed through; for example, if i leaves the barrier logic because $n_k(i)$ has not yet sent its synchronization message, on reentry it may return directly to the dimension k step. However, if i is rolled back in the meantime it may need to start over in dimension 0, or possibly in some other dimension $j < k$. The proper point of entry is given by *state* of the barrier, a pair (D, s). D is the current working dimension, and s is 1 or 0, depending on whether the processor needs to send a synchronization message to $n_k(i)$ (and to $i-1$, if i is a messenger) or not. For example, if i leaves the barrier on failure to find a synchronization message from $n_k(i)$, the barrier state on departure is $(k, 0)$. If the barrier state is not altered by a rollback, then on i's reentry it need not resend the synchronization message—it just checks again for the synchronization message from $n_k(i)$. On the other hand, if i's barrier is rolled back due to receipt of a computation message or re-receipt of a synchronization message in some dimension $j < k$, then the barrier state is reset to $(j, 0)$, (use $j = 0$ if the rollback is due to a computation message).

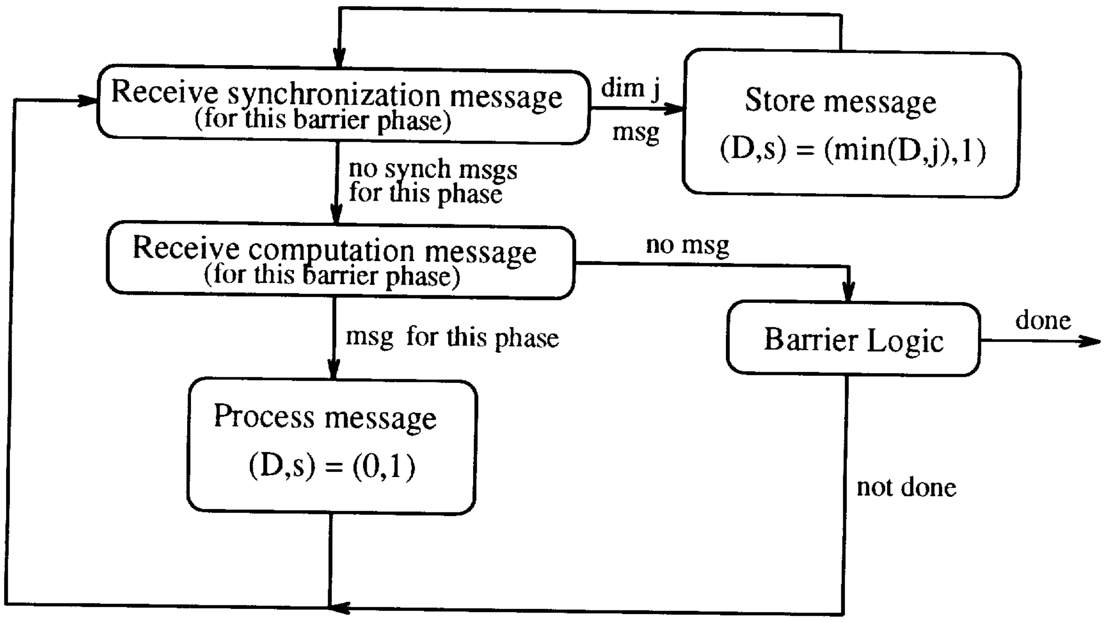

Figure 4: Flow diagram of processing logic using an optimistic barrier

Figure 4 illustrates a flowchart of processing that uses an optimistic barrier. Synchronization messages are given highest priority (although this is not absolutely necessary), and barrier processing is attempted only if there are no known computation messages to process. Entering the barrier logic, the processor pushes through as many dimensions as it can. On passing through dimension $p-1$ the processor may leave the barrier. Otherwise, the thread of control is returned to the user program to receive and process any computation messages that may have arrived since the processor last checked.

The processing shown assumes that messages from processor i to j are delivered in the order in which they are sent (a condition usually satisfied by parallel machines). If this condition cannot be guaranteed (or if synchronization messages are not given highest receipt priority), our algorithm works provided that synchronization and computation messages are tagged with a "phase" identifier, e.g., the number of global barriers completed so far. Phase identification prohibits a processor from accepting a phase k synchronization or computation message before it has completed its phase $k-1$ barrier. In practice, only one bit of phase identification is needed (odd or even phase); in our Intel iPSC implementation we add this bit to the message type identifier.

Before specifying the barrier logic in more detail, we consider some important implementation issues. First there is the problem of determining whether a given neighbor or messenger has sent a synchronization message, and making sure that we access the last such sent from a given processor. To address this, it is straightforward to maintain an array of synchronization messages, indexed by dimension. If we can assume that the system always delivers messages between two processors in the order they are sent, then a synchronization message from $n_k(i)$ (or a messenger in dimension k) is simply copied into the k^{th} element of this array, overwriting whatever may be there already. If the system may deliver messages out-of-order, we just include another count field in the message. A sender records the number of synchronization messages it has sent to the receiver thus far; the receiver can compare the field of a newly received message with that of its current copy. Late messages are simply discarded.

Another issue concerns definition of the barrier "state". Processor i can enter the barrier at dimension $k > 0$ and be required to send vectors $SendVec_k(i)$ and $RecvVec_k(i)$. These vectors depend on synchronization messages from neighbors and messengers in lower dimensions; rigorously there is no need for further barrier state, since those messages are available to support recomputation of the vectors. However, if the barrier is at dimension k, then we know that $SendVec_k(i)$ and $RecvVec_k(i)$ have already been computed. Rather than recompute them on reentry, it is convenient to include them as part of the barrier state. Thus we save a copy of these vectors once they are computed and sent. Each reentry to the barrier will then be able to recover the saved vectors.

Rollback processing deserves special mention. Either the receipt of a computation message or the receipt of a synchronization message in dimension $j < D$ causes a rollback. The rollback consists entirely of resetting the barrier state (D, s) as appropriate. It is not necessary to cancel the synchronization messages already sent in dimensions j through D, for they will be resent, and in being resent may cause rollback.

Another optimization is also possible. Before sending a synchronization message in dimension k we can compare the present vectors $SendVec_k(i)$ and $RecvVec_k(i)$ with their counterparts (if any) the last time we visited dimension k—recall that we save these vectors after transmission. If we observe absolutely no difference between the vectors we are about to send and those we last sent, then there is no need to resend the synchronization message. This idea (called lazy cancellation[12]) has been developed in the parallel simulation world, and has proven to be effective.

Since $O(\log P)$ counts are transmitted and analyzed at each of $\log P$ steps, the algorithm's time complexity is $O(\log^2 P)$. In addition, $O(\log P)$ is space required at every processor to store the shell counts, and synchronization vectors. Figure 5 gives the barrier logic (after the receipt of synchronization messages).

Optimistic Barrier Synchronization Algorithm (viewed from Processor i)

1. If $D = 0$ and $s = 1$, initialize

$$SendVec_0(i) = [Send_0(\{i\}), \ldots, Send_{p-1}(\{i\})]$$

   and

$$RecvVec_0(i) = [Recv_0(\{i\}), \ldots, Recv_{p-1}(\{i\})].$$

   Otherwise recover the saved copies of $SendVec_D(i)$ and $RecvVec_D(i)$. After initialization, we call these vectors *working copies*.

2. If $D = 0$ and i is a hermit, save $SendVec_0(i)$ and $RecvVec_0(i)$ (as $SendVec_1(i)$ and $RecvVec_1(i)$), set $D = 1$, goto (1).

3. Send the working copies of $SendVec_D(i)$ and $RecvVec_D(i)$ to $n_D(i)$ (and to $i - 1$, if i is a messenger in dimension D), provided that $s = 1$ *and* either the working copy of $SendVec_D(i)$ is not identical to the last saved copy of $SendVec_D(i)$, or the working copy of $RecvVec_D(i)$ is not identical to the last saved copy of $RecvVec_D(i)$. If a message is sent at this step, then save $SendVec_D(i)$ and $RecvVec_D(i)$.

4. Set $s = 0$.

5. If a synchronization message in dimension D has not been received, return **false**. Otherwise, if $SendVec_D(i)$ and $RecvVec_D(i)$ are not in agreement with the synchronization message, return **false**.

6. If $D = p - 1$ then set $D = 0$ and $s = 1$, release all saved messages and vectors, then return **true**.

7. Compute working copies of $SendVec_{D+1}(i)$ and $RecvVec_{D+1}(i)$ as

$$\begin{aligned} SendVec_{D+1}(i) &= [Send_{D+1}(C_D(i)) + Send_{D+1}(C_D(n_D(i))), \ldots \\ &\qquad Send_{p-1}(C_D(i)) + Send_{p-1}(C_D(n_D(i)))] \end{aligned}$$

   and

$$\begin{aligned} RecvVec_{D+1}(i) &= [Recv_{D+1}(C_D(i)) + Recv_{D+1}(C_D(n_D(i))), \ldots \\ &\qquad Recv_{p-1}(C_D(i)) + Recv_{p-1}(C_D(n_D(i)))]. \end{aligned}$$

8. Set $D = D + 1$, $s = 1$, goto (3).

Figure 5: Optimistic Barrier Synchronization Algorithm

3.3 Correctness

Finally, we establish the correctness of the algorithm. We need to show both that the algorithm terminates, and that no processor leaves the barrier prematurely. The lemma below establishes termination.

Lemma 1 *For every dimension k there exists a time t_k such that after time t_k no processor reenters the barrier logic with barrier state value $D = k$.*

Proof: We induct on the dimension, k. Consider the base case of $k = 0$. Eventually the last computation message associated with this barrier phase will be sent, and received, say at time T. We may assume that measures (described earlier) are taken to prevent receipt of a computation message from any subsequent barrier phase. Thus, after time T it is not possible for any processor to be rolled back due to the arrival of a computation message. Furthermore, after time T each processor i's individual $\mathit{Send}_m(\{i\})$ and $\mathit{Recv}_m(\{i\})$ counts will equal $\mathit{TotalSend}_m(\{i\})$ and $\mathit{TotalRecv}_m(\{i\})$, respectively. Consequently, any synchronization vectors sent in dimension 0 after time T will reflect completed send/receive totals, and any two processors synchronizing in dimension 0 after time T must find themselves in agreement. Thus, the only way a processor can enter and exit the barrier logic in dimension 0 after time T is if it fails to find a message from its dimension 0 neighbor (or a messenger). Clearly that message must eventually arrive, since by definition of T it must eventually be sent. Consequently, every processor must eventually advance to dimension 1; t_0 is the time at which the last one does. For the induction hypothesis suppose there exists a dimension $k-1$ and time t_{k-1} such that after time t_{k-1} no processor reenters the barrier in a dimension smaller than $k-1$. The proof of the induction step is entirely similar to that of the base case, with t_{k-1} playing the role of T.

∎

The final step is to show that no processor leaves the barrier before every processor has received and processed all of its messages from the barrier phase.

Lemma 2 *For each processor i let s_i be the time at which it completes processing of its last computation message in the current phase, and let e_i be the time at which it departs the barrier. Then for every i, $e_i \geq \max_j\{s_j\}$.*

Proof: We induct on p, the dimension of the system. The base case of $p = 0$ is trivially satisfied. Suppose then that the result holds for all systems of dimension $p-1$. Consider a system of dimension p, choose any processor i and consider the time e_i at which i departs the barrier. Now for i to depart it must be true that both i and $n_{p-1}(i)$ passed through dimension $p-2$, say at times u and v respectively. We can view $C_{p-1}(i)$ and $C_{p-1}(n_{p-1}(i))$ (or $C_{p-1}(i+1)$, as appropriate) as separate systems of dimension $p-1$, and consider u and v to be departure times in the smaller systems (respectively). For every processor j in $C_{p-1}(i)$ let a_j be the time at which j completes processing of its last message *from another processor in* $C_{p-1}(i)$; similarly define b_k for any processor k in $C_{p-1}(n_{p-1}(i))$. By the induction hypothesis we have $u \geq \max_j\{a_j\}$ and $v \geq \max_k\{b_k\}$. Observe that

$$e_i \geq \max\{u, v\} \geq max_{j,k}\{a_j, b_k\}.$$

We claim that $a_j = s_j$ and $b_k = s_k$, for all j and k, for suppose not. $s_k > b_k$ only if processor k eventually receives a message from some processor in $C_{p-1}(i)$. This message must be accounted for as part of i's vector $SendVec_{p-1}(i)$ (since i does no message processing after time u), but is not accounted for in $RecvVec_{p-1}(n_{p-1}(i))$. This is a contradiction however, for i to depart the barrier it must first be in agreement with $n_{p-1}(i)$. Thus $s_k = b_k$ for all processors k in $C_{p-1}(n_{p-1}(i))$ (and similarly $s_j = a_j$ for all processors j in $C_{p-1}(i)$), completing the induction. ∎

4 Empirical Results

Our optimistic barrier provides more flexibility than a conventional barrier, but at a cost. Our algorithm sends vectors of data at each synchronization, it compares vectors prior to transmission in an effort to avoid unnecessary retransmission, and it implements message passing logic at the user level. All of these activities exact costs not suffered by an optimized conventional barrier. In this section we endevour to quantify these costs, by comparing the performance of our barrier with that of the conventional barrier provided on a large-scale parallel architecture.

We first quantify the relative cost of our algorithm in the absence of rollbacks. Table 1 presents timings from the Intel Touchstone Delta[7]. The Delta is a mesh architecture, with 560 total processors. The global synchronization provided with the system—`gsync()`—does **not** work precisely like the standard barrier we described earlier, as it is optimized for a mesh, not a hypercube.

These experiments simply call the barrier algorithms repeatedly. The numbers presented are averages taken over thousands of calls. Since there is no other message passing, our algorithm does not rollback. Even so, our algorithm experiences memory copy and comparison costs at every step. These measurements show that that on large architectures, the cost of our barrier is only slightly more than twice that of `gsync()`.

Size	opt barrier	gsync	Size	opt barrier	gsync
3 × 3	1.14 ms	0.56 ms	10 × 10	2.10 ms	1.00 ms
4 × 4	1.30 ms	0.56 ms	11 × 11	2.19 ms	1.05 ms
5 × 5	1.40 ms	0.65 ms	12 × 12	2.29 ms	1.09 ms
6 × 6	1.57 ms	0.74 ms	13 × 13	2.38 ms	1.14 ms
7 × 7	1.73 ms	0.82 ms	14 × 14	2.49 ms	1.17 ms
8 × 8	2.0 ms	0.92 ms	15 × 15	2.53 ms	1.20 ms
9 × 9	2.0 ms	0.94 ms	16 × 16	2.71 ms	1.24 ms

Table 1: Comparison of time required to execute optimistic barrier vs. time required to execute **gsync()** on Intel Touchstone Delta.

Size	opt barrier	gsync	Size	opt barrier	gsync
3 × 3	0.41 ms	0.13 ms	10 × 10	0.26 ms	0.08 ms
4 × 4	0.30 ms	0.09 ms	11 × 11	0.26 ms	0.07 ms
5 × 5	0.29 ms	0.08 ms	12 × 12	0.25 ms	0.07 ms
6 × 6	0.29 ms	0.08 ms	13 × 13	0.25 ms	0.07 ms
7 × 7	0.29 ms	0.08 ms	14 × 14	0.24 ms	0.07 ms
8 × 8	0.28 ms	0.08 ms	15 × 15	0.24 ms	0.07 ms
9 × 9	0.26 ms	0.08 ms	16 × 16	0.24 ms	0.07 ms

Table 2: Comparison of optimistic barrier and **gsync()** on Intel Touchstone Delta; the experiment measures time-per-hop when cycling a message.

Considering all of the extra costs involved and the fact that **gsync()** is optimized for the mesh architecture, we view this as very encouraging. So long as the cost of the computation of interest is not dominated by the barrier, the relative expense of using an optimistic barrier is not large.

A second set of experiments is designed to measure relative costs in the presence of rollbacks. In these experiments each processor is to receive, and send, one message. A cycle begins with processor 0, who sends a message to processor 1. Upon receipt of a message, processor i ($i \neq 0$) sends a message to processor $(i+1) \bmod P$. The cycle completes when 0 receives a message. Implementation using an optimistic barrier lets the barrier logic determine when all messages to be generated have been (after 0 reenters the barrier after receiving a message). Observe that receipt of every message will cause a rollback in the receiving processor. Implementation using **gsync()** simply has a processor block waiting for its single message, send a message upon its receipt, and then call **gsync()**. Table 2 gives the average times required to complete a cycle, divided by the number of processors used.

Now we find that the cost of using an optimistic barrier is over three times that of using **gsync()**. This

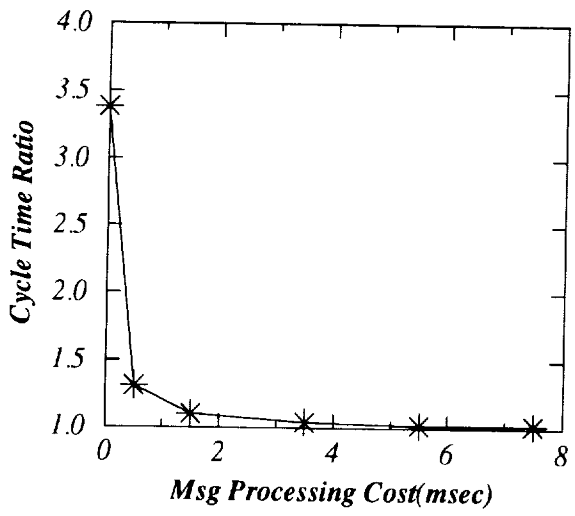

Figure 6: Ratio of time required to complete a cycle using an optimistic barrier, to that required using gsync()

ought to be viewed as an upper bound, since any computation related to message passing will be the same in both versions, and will serve to lessen the ratio of their running times. A final set of experiments illustrates this point, by modeling the cost of message processing. These experiments are identical in structure to the previous set, save that upon receiving a message, a processor waits for a specified period of time before sending the message on. The parameter in these experiments is the average number of milliseconds a processor waits. Figure 6 plots the ratio of time required by our algorithm to complete a cycle, to the time required using gsync(), using 256 processors. Here we see that even under a modest half millisecond message processing time, use of our optimistic barrier is only 30% more expensive than gsync(); at higher message processing costs the relative difference is well under 5%.

We also examined the cost of our algorithm vs gsync() on an Intel iPSC/860 multiprocessor. This architecture has a hypercube topology. In these experiments processor counts were always powers of two, and synchronization messages were always exchanged between processors that are directly connected. The relative difference between our algorithm and gsync() was observed to be nearly identical to that observed on the Delta, implying that the the network bandwidth of the Delta is sufficient to support our algorithm's "artificial" tree construction without significant cost to performance.

5 Extensions

Next we consider extending the optimistic barrier algorithm to include the optimistic computation (with global barrier synchronization) of any reduction operation of an associative operator $\otimes$, as well as an optimistic parallel prefix computation of $\otimes$. The fact that this is possible is evident from our algorithm's basis in a tree structure; the computation of reductions and prefix operations on trees is already well-understood [1]; the point of this section is give enough details it show where our algorithm can be modified to support these operations.

Let $\mathcal{S}$ be a set, and $\otimes : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ be an associative operator. Imagine that after processing all messages, each processor i has computed some $m_i \in \mathcal{S}$, and we desire that every processor learn the reduced element $m_0 \otimes m_1 \otimes \ldots \otimes m_{P-1}$. This is easily accomplished with a small modification to our barrier algorithm. First we introduce some new definitions.

Processors in a sequence $C_k(i)$ have contiguous ids; define $l_k(i)$ to be lowest element, and $u_k(i)$ to be the greatest element of this sequence. Also define

$$M_k(i) = m_{l_k(i)} \otimes m_{l_k(i)+1} \otimes \ldots m_{u_k(i)}.$$

When i and $n_k(i)$ synchronize, suppose that i knows $M_k(i)$, $n_k(i)$ knows $M_k(n_k(i))$, and these elements are exchanged. This is clearly possible for $k = 0$, as $M_0(i) = m_i$ and $M_0(n_0(i)) = m_{n_0(i)}$. At an arbitrary dimension k, given $M_k(i)$ and $M_k(n_k(i))$ i and $n_k(i)$ can compute $M_{k+1}(i) = M_{k+1}(n_k(i)) = M_k(i) \otimes M_k(n_k(i))$ (assuming here that $i < n_k(i)$). This is merely a percolation of partial sums up the tree, which is standard practice in reduction algorithms. Continuing in this fashion, at the point processor i leaves the barrier it will have computed $M_P(i)$, which is the desired reduced value. To incorporate reduction in the barrier, all we need to do is include the element $M_k(i)$ with $Send_k(i)$ and $Recv_k(i)$ as part of the synchronization vector, saving it and restoring it when the synchronization vector is saved and restored, and to add the additional logic needed to implement $\otimes$ in the right order.

The barrier can also be extended to provide parallel prefix computations. In these, we compute $m'_0 = m_0$, $m'_1 = m_0 \otimes m_1$, $m'_2 = m_0 \otimes m_1 \otimes m_2$, and so on, all the way up to $m'_{P-1} = m_0 \otimes m_1 \ldots \otimes m_{P-1}$. Processor i receives one element of this sequence, m'_i. We assume that the elements $M_k(i)$ described for the reduction operation are being computed, and will use them in such a way that processor i can construct its prefix element.

Given processor id i, for $k = 0, \ldots, p-1$ define b_k to be 1 if $C_k(i)$ is the right child of its parent, and to be 0 otherwise. Observe that if $P = 2^p$, then the bits $\{b_k\}$ describe i's id in the base 2 number system. In the general case, i is uniquely identified by these bits. We exploit the following result, based on this definition.

Lemma 3 *For any processor i, define bit code b_k ($k = 0, \ldots, p-1$) to be 1 if $C_k(i)$ is the right child of its parent, and to be 0 otherwise. Let the dimensions in which $b_k = 1$ be enumerated as $d_0, \ldots, d_s$, in ascending* *order. Then*

$$m_i' = M_{d_s}(n_{d_s}(i)) \otimes M_{d_{s-1}}(n_{d_{s-1}}(i)) \otimes \ldots \otimes M_{d_0}(n_{d_0}(i)) \otimes m_i.$$

Proof: Induct on the system dimension p. The base case is immediate. For the induction hypothesis, suppose the result holds for any system of dimension $k-1$, and consider processor i in a system of dimension k. If $C_k(i)$ is the left child of it's parent we are done, by the induction hypothesis, for then i is a member of a system of dimension $k-1$. Otherwise, we must have $b_k = 1$, and $d_k = k$. We may write $i = l_k(i) + \hat{i}$, and consider $\hat{i}$ as a member of a system (rooted in $C_k(i)$) of dimension $k-1$. Let $u = m_{l_k(i)} \otimes \ldots \otimes m_i$; this is the prefix element for $\hat{i}$ in the reduced system. Let $\hat{d}_0, \hat{d}_1, \ldots, \hat{d}_t$ enumerate in ascending order the dimensions in which $\hat{i}$'s bit codes are non-zero. Then by the induction hypothesis

$$u = M_{\hat{d}_t}(i) \otimes \ldots \otimes M_{\hat{d}_0}(i) \otimes m_i.$$

The induction is completed by the observation that $m_i' = M_k(n_k(i)) \otimes u$, and that $d_s = k$. ∎

Lemma 3 shows how each processor i can combine the elements $M_k(i)$ it computes to form m_i'. Define $F_k(i)$ be the "working" value of the prefix at dimension k. Initially $F_0(i) = m_i$. We have assumed already that $M_k(i)$ is computed upon passing through dimension k. Then, if i's bit code b_k is set, it computes $F_{k+1}(i) = M_k(n_k(i)) \otimes F_k(i)$; otherwise $F_{k+1}(i) = F_k(i)$. As an illustration, consider the computation of m_4' in the six processor system depicted in Figure 1. Processor 4's bit codes are $b_0 = 1$, $b_1 = 0$, and $b_2 = 1$. Synchronizing in dimension 0, processor 3 sends $m_3 = M_0(n_0(4))$ to processor 4. Since $b_0 = 1$, processor 4 computes $F_1(4) = M_0(n_0(4)) \otimes F_0(4) = m_3 \otimes m_4$. After synchronizing in dimension 1, $F_2(4) = F_1(4)$ because $b_1 = 0$. Synchronizing in dimension 2, processor 1 sends $M_2(C_2(n_2(4)) = m_0 \otimes m_1 \otimes m_2$ to processor 4, who computes $m_4' = F_3(4) = M_2(C_2(n_2(4)) \otimes F_2(4)$.

Although the $F_k(i)$ elements are not exchanged with other processors, they do form part of the barrier's state, and ought to be saved and restored in the same fashion as the send/receive vectors, and the $M_k(i)$ elements.

6 Summary

Barrier synchronization is an integral part of many parallel algorithms. All barrier algorithms of which we are aware assume that a processor knows when it is safe to enter the barrier. However, for some applications is it difficult to determine when a processor has completed all work that might be required of it prior to synchronization. We first encountered this problem in the context of parallel discrete event simulation, yet we believe the problem may occur for any computation whose behavior is driven by the receipt and processing of messages.

We propose a solution based on optimistic execution of a modified standard barrier algorithm. Our algorithm differs from the standard technique in that (i) it permits a processor to back out of a barrier when a computation message is received, (ii) the barrier computation is performed optimistically (complete with state-saving, rollback, and cancellation optimizations), and (iii) counts of messages send and received between certain sets of processors are included in the synchronization messages, and are used to determine when all processors have reached the barrier and have executed all workload necessary prior to the barrier. Despite its seeming complexity, experiments on a large-scale multiprocessor show that the algorithm is only 2-3 times slower than the optimized deterministic barrier provided with the system, and that the relative additional cost disappears when any significant computation is associated with handling a message.

Intel iPSC source code for the optimistic barrier is available by anonymous ftp to host `cs.wm.edu`, in file pub/outgoing/opt_barrier.c (a listing of the contents of `pub/outgoing` cannot be read, but an ftp `get` on the file will work).

Acknowledgements

We thank Albert Greenberg of AT&T Bell Labs for running our code (with the support of Sandia National Labs) on the Intel Touchstone Delta machine. We also acknowledge the usefulness of discussions on optimistic barrier synchronization with Phillip Dickens, Paul Reynolds, Richard Fujimoto, Lisa Sokol, and Harry Jordan.

References

[1] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[2] N.S. Arenstorf and H.F. Jordan. Comparing barrier algorithms. *Parallel Computing,* 12(2):157–170, November 1989.

[3] T.S. Axelrod. Effects of synchronization barriers on multiprocessor performance. *Parallel Computing,* 3(2):129–140, May 1986.

[4] S. Bellenot. Global virtual time algorithms. In *Distributed Simulation 1990*, volume 22, pages 122–127. SCS Simulation Series, 1990.

[5] S. Eick, A. Greenberg, B. Lubachevsky, and A. Weiss. Synchronous relaxation for parallel simulations with applications to circuit-switched networks. In *Proceedings of the 1991 Workshop on Parallel and Distributed Simulation*, pages 151–162, Jan. 1991.

[6] D. R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and Systems*, 7(3):404–425, 1985.

[7] Sigurd L. Lillevik. The Touchstone 30 gigaflop DELTA prototype. In *Distributed Memory Computer Conference 91*, pages 671–677. IEEEPRESS, April 1991.

[8] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[9] F. Mattern. Experience with a new distributed termination detection algorithm. In *Distributed Algorithms*. Springer-Verlag, New York, 1987.

[10] P.F. Reynolds, Jr. An efficient framework for parallel simulations. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 167–174. SCS Simulation Series, Jan. 1991.

[11] D.M. Nicol. Conservative parallel simulation of priority class queueing networks. *IEEE Trans. on Parallel and Distributed Systems*, 3(3):294–303, May 1992.

[12] Peter L. Reiher, Richard Fujimoto, Steven Bellenot, and David Jefferson. Cancellation strategies in optimistic execution systems. In *Distributed Simulation 1990*, pages 112–121. Society for Computer Simulation, 1990.

[13] L.M. Sokol, D.P. Briscoe, and A.P. Wieland. MTW:A strategy for scheduling discrete simulation events for concurrent execution. In *Distributed Simulation 1988*, pages 34–42. SCS Simulation Series, 1988.

[14] J.S. Steinman. Speedes: Synchronous parallel environment for emulation and discrete event simulation. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 95–103. SCS Simulation Series, Jan. 1991.

[15] S. Turner and M. Qu. Performance evaluation of the bounded time warp algorithm. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, volume 24, pages 117–126. SCS Simulation Series, 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE July 1992	3. REPORT TYPE AND DATES COVERED Contractor Report

4. TITLE AND SUBTITLE
OPTIMISTIC BARRIER SYNCHRONIZATION

5. FUNDING NUMBERS
C NAS1-18605
WU 505-90-52-01

6. AUTHOR(S)
David M. Nicol

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 92-34

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-189684
ICASE Report No. 92-34

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
Submitted to Journal of Parallel & Distributed Computing

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 59, 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Barrier synchronization is a fundamental operation in parallel computation. In many contexts, at the point a processor enters a barrier it knows that is has already processed all work required of it prior to the synchronization. This paper treats the alternative case, when a processor cannot enter a barrier with the assurance that it has already performed all necessary pre-synchronization computation. The problem arises when the number of pre-synchronization messages to be received by a processor is unknown, for example, in a parallel discrete simulation or any other computation that is largely driven by an unpredictable exchange of messages. We describe an optimistic $O(\log^2 P)$ barrier algorithm for such problems, study its performance on a large-scale parallel system, and consider extensions to general associative reductions, as well as associative parallel prefix computations.

14. SUBJECT TERMS
synchronization; parallel simulation; optimistic computation; parallel prefix

15. NUMBER OF PAGES
22

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT