GLOBAL SYNCHRONIZATION FOR OPTIMISTIC PARALLEL DISCRETE EVENT SIMULATION

David M. Nicol*
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795

Abstract

A number of optimistic synchronization schemes for parallel simulation rely upon a global synchronization. The problem is to determine when every processor has completed all its work, *and* there are no messages in transit in the system that will cause more work. Most previous solutions to the problem have used distributed termination algorithms, which are inherently serial; other parallel mechanisms may be inefficient. In this paper we describe an efficient parallel algorithm derived from a common "barrier" synchronization algorithm used in parallel processing. The algorithm's principle attraction is speed, and generality—it is designed to be used in contexts more general than parallel discrete-event simulation. To establish our claim to speed, we compare our algorithm's performance with the standard barrier algorithm, and find that its additional costs are not excessive. Our experiments are conducted using up to 256 processors on the Intel Touchstone Delta.

1 Introduction

A number of algorithms for parallel simulation are based upon the notion of a window in simulation time, where activity associated with time-stamps in the window may be executed in parallel, but successive windows are processed serially. Examples include bounded-lag[4], conservative time windows[2], YAWNS[5], MTW [7],SPEEDES [9], and Bounded Time Warp [10]. The last three of these protocols are optimistic. In a conservative protocol, at the point a processor has simulated up to the end of the window it knows that it will not get a message in its past. In this case, in order to establish that all processors have finished the window it suffices to use a standard barrier synchronization algorithm, such as the butterfly barrier [1]. Such algorithms are efficient and parallelizable (usually enjoying a $\log(P)$ complexity on P processors), and are provided as a system call in most parallel computer systems. However, consider a processor running an optimistic protocol. At the point it has simulated up to the end of the window it may still be rolled back by the arrival of a straggler message. Standard barrier algorithms will not work in this case, as they have no provision for a processor needing to back out of the barrier.

The fore-mentioned optimistic windowing algorithms use different approaches for determining when a window has completed. MTW's scheme is to monitor the idleness of LPs, and to initiate a GVT calculation once the fraction of idle LPs exceeds some threshold. Synchronization is detected when the GVT is greater than the upper window edge. SPEEDES uses parallel operations to compare the total number of messages sent with the total received, and upon equality applies a GVT calculation. Such a calculation is the heart of our approach as well; however, the mechanics and frequency of SPEEDES' computations is not documented in the literature. Our algorithm performs the computation essentially once. Bounded Time-Warp uses an algorithm from the distributed termination literature. A token is passed serially among processors. Any "busy" processor receiving the token marks it. Once the token has circulated twice without being colored, then the processor who generated the token knows the computation has reached the end of the window; an additional communication phase is needed to notify all processors of this fact.

As recognized by SPEEDES and (various GVT algorithms) the key to the problem is to compare the number of messages sent and received in the system. If no processor will execute an event past time t, we

*This research was supported in part by NASA grants NAG-1-1060, NAG-1-1132, and NAG-1-995, and NSF Grants ASC 8819373 and CCR-9201195.

know that *all* the processors have completed once every processor is idle and the total number of messages sent is equal to the total number received. The problem is to perform this calculation efficiently, and to do so in such a way that every processor quickly knows when it is synchronized with every other processor.

In this paper we show how modification of a standard algorithm (the butterfly barrier[1]) permits the use of barrier synchronization in an optimistic computation. 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. 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 address is not original; others have had to tackle it, notably [9, 10]. Furthermore, most of the key ideas in this algorithm originated elsewhere. For instance, the notion of comparing message send/receive counts lies at the heart of standard GVT algorithms; combining trees (hardware or software) are as old as parallel processing itself, as is the butterfly communication pattern; optimistic execution has been well explored in parallel simulation, and database research. Our contribution is to *combine* these ideas to produce an *efficient* and *general* algorithm for global optimistic synchronization. The efficiency derives from our identification of $O(\log P)$ sized quantities (our shell counts) which support the send/receive analysis; the efficiency is proven by comparing our algorithm with a highly optimized non-optimistic synchronization barrier synchronization on a large-scale multiprocessor. The generality follows from design; the support for optimistic processing is an integral part of the algorithm, and does not assume any optimistic simulation kernal. The problem of needing to synchronize in the face of uncertainty is not unique to parallel simulation, e.g. it has also appeared in the context of parallel numerical algorithms[8]. Our algorithm works equally well in this more general context.

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 argues for the algorithm's correctness. Section §4 evaluates the performance of our algorithm on large scale multiprocessors. Section §5 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. In the context of parallel simulation, it means that processors synchronize at a logical simulation time, say s—no processor is to execute any event, even optimistically, until every processor has simulated all events up to time s. 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 algorithm we propose, and yet do not automatically 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.

To be sure, one can rig standard optimistic simulation approaches to provide a barrier-like mechanism. A reviewer of this paper suggested that one create events to "freeze" the state, and then apply a global state-detection algorithm to determine whether every processor's state is frozen. The point of this paper is not to solve a new problem. The point is to solve it quickly,

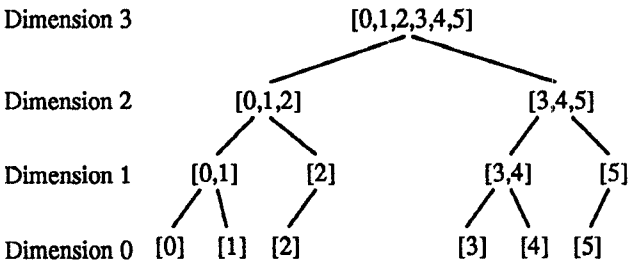

Figure 1: Balanced tree created by splitting sets of processors ids.

and generally. The fact that standard optimistic algorithms worry much about memory management, and yet invoke GVT calculations infrequently suggests that standard approaches to problems related to ours are viewed as having high overhead.

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. As shown in [8], the problem can also arise in numerical contexts, when convergence determines termination. A processor whose subdomain has converged enters the optimistic barrier, but can receive a message containing boundary interface values from a neighboring processor whose subdomain has not.

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 its 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. 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 $P = 6$.

$i \backslash j$	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 and specifying the barrier algorithm.

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

[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.

$S_m(k)$, $m \geq k$.

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 $m \geq k$, $TotalSend_m(C_k(i)) = TotalSend_m(\{i\})$ for $k = 0$ and $TotalSend_m(C_k(i)) = TotalSend_m(C_{k-1}(i)) + TotalSend_m(C_{k-1}(n_{k-1}(i)))$ for $k > 0$; while $TotalRecv_m(C_k(i)) = TotalRecv_m(\{i\})$ for $k = 0$ and $TotalRecv_m(C_k(i)) = TotalRecv_m(C_{k-1}(i)) + TotalRecv_m(C_{k-1}(n_{k-1}(i)))$ for $k > 0$.

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:

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

and

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

Figure 2 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 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.

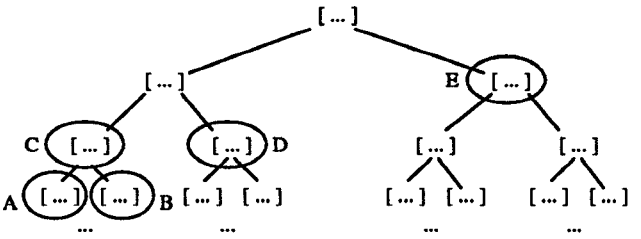

Figure 2: Graphical depiction of information passed when i (in A) synchronizes with $n_k(i)$ (in B). i gives $n_k(i)$ send/receive counts between processors in A and B, A and D, A and E. $n_k(i)$ gives i send/receive counts between processors in B and A, B and D, B and E.

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 3 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

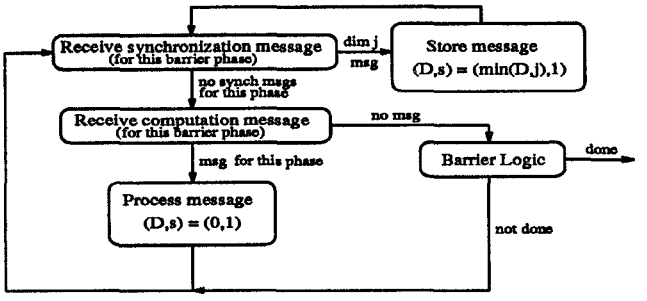

Figure 3: Flow diagram of processing logic using an optimistic barrier

there are no known computation messages to process. Entering the barrier logic in state (D, s), the processor pushes through as many dimensions as it can, beginning with dimension D. 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).

Lazy cancellation [6] can be applied to the optimistic barrier. 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 the barrier will reach the point where their transmission is logically appropriate again. However, upon reaching a logical transmission point again, the processor may find that the vector it ought to send is identical to the vector it previously send, in which case retransmission is unnecessary.

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.

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.

First consider termination. For the sake of contradiction, suppose that the algorithm does not terminate. Now every processor eventually performs all of the work required of it prior to the upper window edge, and every message that causes work has been generated, sent, and received; hence every processor i is in the barrier logic with some state (D_i, s_i). Without loss of generality suppose that $D_0 \leq D_i$ for all $i = 1, \ldots, P-1$. The only reason processor 0 cannot advance is because it does not agree with its neighbor in dimension D_i. But this is impossible, because non-agreement is possible only if there is a generated message that has not yet been received. This established the contradiction.

Finally, we need to argue that no processor leaves the barrier prematurely. In order for a processor i to depart the barrier, it is necessary that it agree with its neighbor in dimension $p-1$. To even reach that stage in the barrier, it is necessary that the processor be in agreement with all its neighbors in dimensions $j = 0, 1, \ldots, p-2$. This can only happen if there is no unprocessed message sent from a processor in $C_{p-2}(i)$ to a processor in the same set. Similarly, for $n_{p-1}(i)$ to synchronize with i it is necessary that there be no unreceived message generated by a processor in $C_{p-2}(n_{p-1}(i))$ for a processor in that same set. Now i cannot depart the barrier unless it agrees with $n_{p-1}(i)$, which can only happen if there is no as-yet-unprocessed message generated by a processor in $C_{p-1}(i)$ for a processor in $C_{p-1}(n_{p-1}(i))$, or vice versa. But every processor is in one of these two sets, implying that processor i cannot leave the barrier before all necessary messages have been generated, sent, and received.

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

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

Table 2: Comparison of time required to execute optimistic barrier vs. time required to execute **gsync()** on Intel Touchstone Delta.

the Intel Touchstone Delta[3]. The Delta is a basically a 16 × 32 PE mesh architecture. The global synchronization provided with the system—gsync()—does not work precisely like the standard barrier we described earlier, being optimized for Delta architecture and NX operating system.

In the first experiment we 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()**. Considering all of the extra costs involved and the fact that **gsync()** is optimized for the Delta while our algorithm is not, 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 $(i+1)$ mod 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 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 4 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

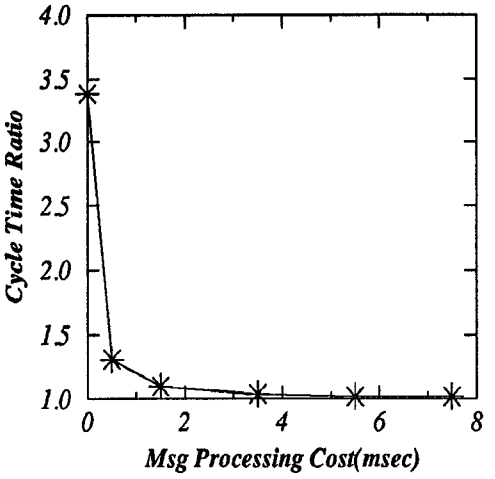

Figure 4: Ratio of time required to complete a cycle using an optimistic barrier, to that required using `gsync()`

relative difference between our algorithm and `gsync()` was observed to be nearly identical to that observed on the Delta, implying that the network bandwidth of the Delta is sufficient to support our algorithm's "artificial" tree construction without significant cost to performance.

5 Summary

Many optimistic parallel simulation algorithms require the global synchronization of processors, where a processor ought not pass through the synchronization if there is any chance that it will be rolled back to a simulation time prior to the synchronization point. The use of an optimistic protocol complicates the problem over that experienced using a conservative algorithm, because standard global synchronization algorithms do not work. In this paper we extend a standard barrier synchronization algorithm by endowing it with optimism. We study the performance of our algorithm with experiments on the Intel Touchstone Delta, using up to 256 processors. Comparisons with a standard barrier algorithm show that the extensions inflate the cost of the barrier by only a factor of 2-3. If any significant amount of processing is associated with a message, this additional cost is quickly amortized.

Acknowledgements

This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Sandia National Labs. 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] T.S. Axelrod. Effects of synchronization barriers on multiprocessor performance. *Parallel Computing*, 3(2):129–140, May 1986.

[2] R. Ayani. A parallel simulation scheme based on distances between objects. In *Distributed Simulation 1989*, pages 113–118. SCS Simulation Series, 1989.

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

[4] B.D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123, 1989.

[5] D.M. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. ICASE Technical report 90-20. To appear in *Journal of the ACM*, 1993.

[6] 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.

[7] 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.

[8] Jianjian Song. A distributed-termination experiment on a mesh-connected array of processors. *Parallel Computing*, 18(2):779–791, July 1992.

[9] 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.

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