International Journal of Parallel Programming, Vol. 17, No. 1, 1988

Two Algorithms for Barrier Synchronization

Debra Hensgen,[1] Raphael Finkel,[1] and Udi Manber[2]

Received March 1988; Revised July 1988

We describe two new algorithms for implementing barrier synchronization on a shared-memory multicomputer. Both algorithms are based on a method due to Brooks. We first improve Brooks' algorithm by introducing double buffering. Our dissemination algorithm replaces Brooks' communication pattern with an information dissemination algorithm described by Han and Finkel. Our tournament algorithm uses a different communication pattern and generally requires fewer total instructions. The resulting algorithms improve Brooks' original barrier by a factor of two when the number of processes is a power of two. When the number of processes is not a power of two, these algorithms improve even more upon Brooks' algorithm because absent processes need not be simulated. These algorithms share with Brooks' barrier the limitation that each of the n processes meeting at the barrier must be assigned identifiers i such that $0 \leqslant i < n$.

KEY WORDS: Barrier; broadcast; distributed; parallel; synchronization; shared memory.

1. INTRODUCTION

This paper presents two new algorithms for barriers. A **barrier** is a tool for synchronizing processes on a shared memory machine. No process may pass the barrier until all other processes have arrived at it. We will call the execution of the barrier synchronization code an **episode.** We define the execution of the code before a barrier to be an **epoch.** Epochs are said to be **adjacent** if they are separated by only one episode. Similarly, adjacent

[1] University of Kentucky, Lexington, Kentucky 40506-0027.
[2] University of Arizona, Tucson, Arizona 85721.

0885-7458/88/0200-0001$06.00/0

828/17/1-1

episodes are separated by a single epoch. These definitions are pictured in Fig. 1.

Episodes prevent any process from proceeding to the next epoch before all processes have made the corresponding barrier call. Thus, a correct implementation of a barrier prevents two processes from executing in adjacent epochs. Two processes may, however, be executing in adjacent episodes. A correct implementation must also ensure that all processes make progress. By progress we mean that once all processes enter an episode, they will all exit the episode in a finite amount of time.

An algorithm due to Brooks makes use of shared memory without locks to implement barriers.[1] We will present this algorithm shortly. Two drawbacks to Brooks' algorithm are that it performs extra work during episodes and is unnecessarily expensive when the number of processes meeting at the barrier is not a power of two.

In the next section, we discuss four algorithms for implementing barriers. The first is a straightforward algorithm that uses locks, the second is Brooks' algorithm, and the third and fourth are our new versions. We compare the algorithms theoretically using a parallel machine model that allows concurrent reads and one write, but not concurrent writes. In Section 3, we conclude with a summary of timings based on our implementation on a Sequent Balance 21000 multicomputer.

2. FOUR ALGORITHMS

2.1. Exclusive Access to Shared Variable

The obvious implementation of a barrier is for each process meeting at the barrier to acquire exclusive write access to a shared counter and

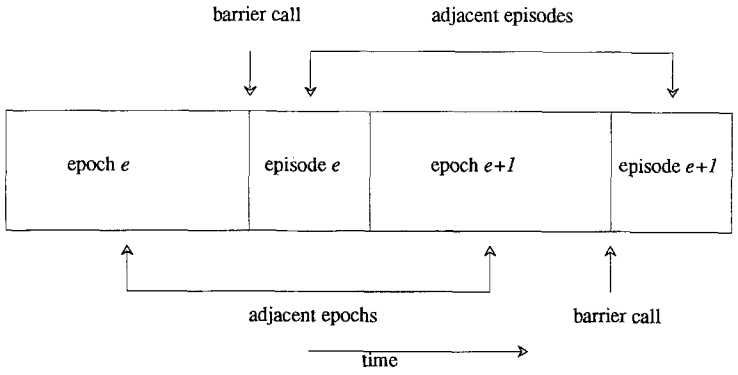

Fig. 1. Basic definitions.

decrement it. Then these processes release exclusive write access and wait until the counter reaches zero, at which point the episode is finished. There are two problems with this algorithm:

- It is centralized, limiting concurrency. If n processes synchronize, $O(n)$ time is required. We call this the **concurrency** problem.
- It is difficult to reinitialize the counter. The process that reinitializes must be sure that all other processes know that the counter has reached the desired value. Otherwise they will remain in the episode, violating the progress criterion. We call this the **reinitialization** problem.

Lubachevsky[2] provides a version of this algorithm meant to run on a shared-memory computer enhanced with hardware fetch-and-add. This version can be executed in $O(\log n)$ time using only one shared variable. Unfortunately, fetch-and-add is not yet available for any computer on the market.

2.2. Brooks' Barrier

Brooks bases the n-process barrier on a two-process barrier using two shared variables.[1] The two-process barrier solves the reinitialization problem and avoids locks. Its algorithm is as follows:

	Process 1	**Process 2**
(1)	**while** SetByProcess1 **do wait**;	**while** SetByProcess2 **do wait**;
(2)	SetByProcess1 := true;	SetByProcess2 := true;
(3)	**while not** SetByProcess2 **do wait**;	**while not** SetByProcess1 **do wait**;
(4)	SetByProcess2 := false;	SetByProcess1 := false;

We will call an execution of these four lines an **instance**. Line (4) reinitializes the barrier for the next episode, solving the reinitialization problem. Line (1) assures a process executing in episode $e+1$ that the other process has completed episode e. Lines (2) and (3) can be understood to say, "I am here; I'll wait for you."

Because we rely on this algorithm later, we provide a correctness proof for it along the lines of Lamport's informal multiprocess correctness proofs.[3] Figure 2 shows the invariants for process 1 in parentheses. The invariants for process 2 are analogous.

Act and Bct are pseudo-variables added to count the number of barriers completed by processes 1 and 2, respectively. Adone and Bdone are pseudo-variables that indicate whether a specific instantiation of the epoch has terminated. The safety criterion is that both processes are either

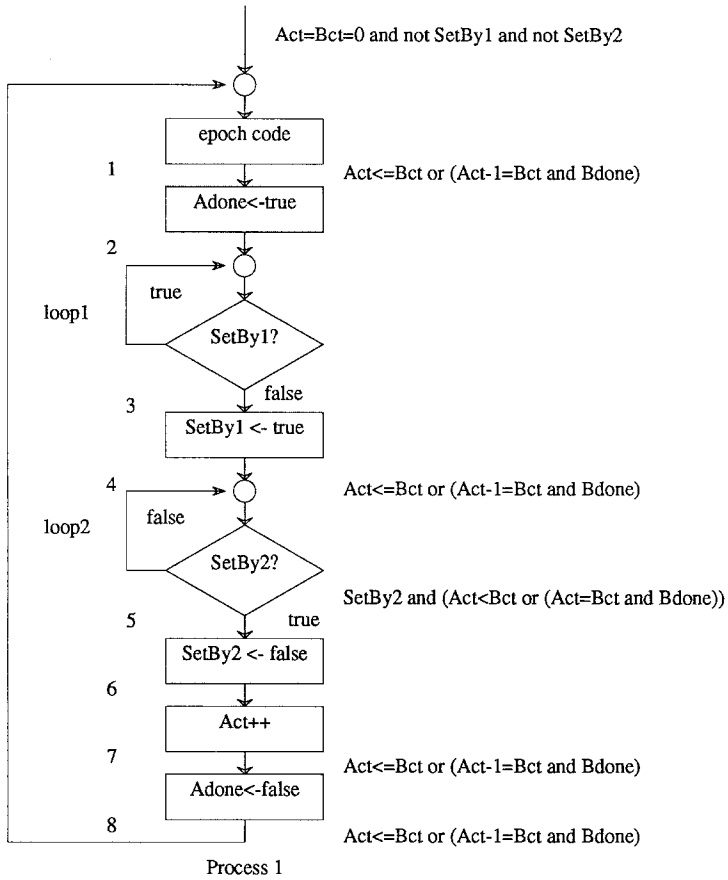

Fig. 2. Invariants for an instance.

in the same epoch or episode ($\mathrm{Act} = \mathrm{Bct}$) or in adjacent episodes. In the latter case, the slower process must have completed the previous epoch (($\mathrm{Act} - 1 = \mathrm{Bct}$) and Bdone) or (($\mathrm{Bct} - 1 = \mathrm{Act}$) and Adone)). The liveness criterion is that both processes will not remain in loops 1 or 2.

Due to symmetry between the processes, to show the safety criterion we need only show that either (($\mathrm{Act} - 1 = \mathrm{Bct}$) and Bdone) or ($\mathrm{Act} \leqslant \mathrm{Bct}$). The initial invariant and the invariant at location **8** imply that the invariant at location **1** holds, since the epoch code does not affect these variables and process 2 may only increase Bct. Process 1 does not change Bct at all and does not change Act between locations **1** and **5**, and process 2 only increments Bct, so the invariant at location **4** holds. To prove the invariant at location **5**, we split the argument into two cases. If

Act = Bct = 0, then process 2 must have set SetBy2, so it must have completed epoch 0, in which case Bdone is true. If Act = n, then process 1 had seen SetBy2 was true n times and cleared it exactly n times before it entered the current episode. Upon reaching location **5**, process 1 sees SetBy2 true for the $n+1^{st}$ time, so process 2 has set SetBy2 $n+1$ times and hence has completed epoch n. The invariants at locations **7** and **8** are a direct consequence of incrementing Act.

The liveness criterion can be verified by showing that no case exists that contains an infinite loop.

- Both processes are in loop1. The fact that they are looping implies that both SetBy1 and SetBy2 are true. However, the last assignments to those variables would have been at location **6**, where they are set to false. So this case is impossible.
- Process 1 is in loop1 and process 2 is in loop2. The fact that process 1 is in loop1 implies SetBy1 is true. The fact that process 2 is in loop2 implies SetBy1 is false. So this case in impossible.
- Process 1 is in loop2 and process 2 is in loop1. This case is identical to the previous case by symmetry.
- Both processes are in loop 2. The fact that they are looping implies that both SetBy1 and SetBy2 are false. However, the last assignments to those variables would have been at location **3**, where they are set to true. So this case is impossible.

Brooks borrows from the butterfly connection startegy to increase concurrency in an n-process barrier. We first present the case where n is a power of two. Each episode is composed of $\log_2 n$ instances, during each of which all processes execute the 4-line algorithm with different variables. Figure 3 shows the pairing of instances for eight processes.

A good way to view an instance is as a message.[4] Each instance conveys information about the sender's experience so far during the current episode. For example, when process 2 communicates with process 6 in instance 2, it is saying "I am in episode e, I have heard from process 3 that it is also, and I have heard from process 0 that both process 0 and process 1 are also. I am waiting for you to send me a message indicating that you have heard from all of the processes that you expected to hear from." By the time process p has finished that last instance, it has heard that all n processes have entered the current episode.

When the number of processes is not a power of two, Brooks' algorithm chooses n, the next larger power of two, and uses existing processes to simulate absent ones. Brooks suggests a method for determining which processes should simulate the absent ones. The most

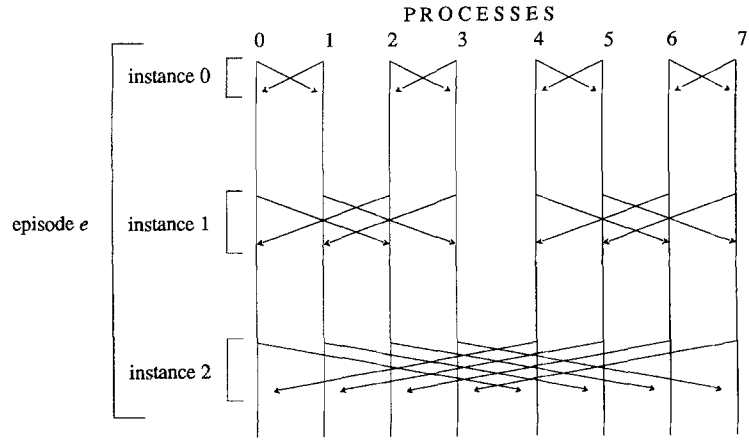

Fig. 3. Communication structure for Brooks' algorithm.

straightforward way is to have process p simulate process $n-1-p$ when process $n-1-p$ is absent. Brooks points out that some instances can be avoided in this case by more carefully choosing the simulators. Unfortunately, Brooks does not give an algorithm for choosing better simulators, and it appears that they must be chosen on a case-by-case basis for each odd n. (Simulations for even n can be derived from simulations for $n/2$.)

The communication pattern requires that each process have a unique identifier in the range 0 to $n-1$. If different processes participate in different episodes, they must agree on such a numbering for each episode. The same warning applies to the new algorithms we will present shortly.

This algorithm requires $O(n \log n)$ space for the shared variables, since each instance in the episode uses a different set of variables. Assuming n concurrent processes and the time for one instance to be I, the total time required is $I \log_2 n$, when n is a power of two; otherwise the total time is between $I\lceil \log_2 n \rceil$ and $2I\lceil \log_2 n \rceil$.

2.3. Shortening an Instance

Before we turn to our first new algorithm, we will improve Brooks' algorithm by shortening the code executed during an instance. Obviously, lines (2) and (3) are necessary because they simulate the actual sending and receiving of the information. Below we discuss the importance of lines (1) and (4) in order to eliminate them.

Line (4) seems necessary for each instance, since without it a process in episode e expecting to hear from process p could see the appropriate variable still set from episode $e-1$. Line (1) seems to be required because

processes might be executing in adjacent episodes. Process p could be in episode $e+1$, instance i, wanting to tell process q that it has entered episode $e+1$. Meanwhile, process q might still be in episode e, instance i, just about to execute line (3). If process p skipped line (1) and executed line (2), then process q would complete episode e, but would be blocked in episode $e+1$.

This discussion shows that the importance of line (1) is to prevent a process executing in one episode from confusing another process executing in the previous episode. Line (4) is needed to prevent a process from viewing an old value previously set by another process.

Line (1) can be obviated by a double-buffering scheme whereby processes use alternate arrays for the shared variables. That is, episode e uses buffer set $e \bmod 2$. Using alternating arrays of shared variables prevents processes in adjacent episodes from blocking each other.

We can eliminate line (4) similarly by having processes alternate between setting the shared variables to true and to false. We call this technique **sense switching**. Assume that all shared variables in one array are initialized to false. During the first episode that uses that array, processes set variables to true and wait to see variables set to true. The next time that array is used in an episode, processes set variables to false and wait to see variables set to false. After that the variables are set to true and so on. Sence switching would not work without double buffering, because then process p could be in episode e setting the variable to be read by process q to false, and process q could be in episode $e-1$ just about to check whether this variable had been set to true. With double buffering the only confusion would be between processes simultaneously in episodes e and $e+2$, but processes may only be up to one episode apart.

We can now remove lines (1) and (4) from each instance in the episode. We must also change the instance so that variables will be alternately set to true and false. We will call this shortened instance a **modified instance**. Brooks' algorithm (and the ones we will present shortly) can use the modified instance only when the number of cooperating processes is fixed; a modified instance cannot be used when a different number of processes meet at subsequent barriers because there is no explicit reinitialization. The following algorithm is for a modified instance.

```
/* per-episode variables */
   OddOrEven = episode mod 2; /* which array should be used */
   signal = (episode div 2) mod 2; /* which signal */
shared AnswersOdd[NPROCS] [LogNPROCS]: integer;
shared AnswersEven [NPROCS] [LogNPROCS]: integer;
OddOrEven = episode mod 2;
```

```
signal = (episode div 2) mod 2;
procedure ModifiedInstance(OddOrEven, signal, instance);
var OddOrEven, signal, instance: integer;
begin
    if OddOrEven = 0 then
        AnswersEven[intended[myid][instance]][instance] := signal;
        while not AnswersEven[myid][instance] = signal do wait;
    else /* OddOrEven = 1 */
        AnswersOdd[intended[myid][instance]][instance] := signal;
        while not AnswersOdd[myid][instance] = signal do wait;
    end
end ModifiedInstance;
```

2.4. The Dissemination Algorithm

An algorithm described by Han and Finkel[5,6] and others[7] achieves complete dissemination of information among n processes in $\lceil \log_2 n \rceil$ synchronized rounds. For our purposes it is sufficient to think of a round as a simultaneous set of instances, one instance being executed by each process. During round i, process p sends all of the information that it knows to process $2^i + p \pmod{n}$. If a process waits to receive the message sent to it during round i and incorporates that message into its own message for all subsequent rounds starting with round $i+1$, then all processes receive

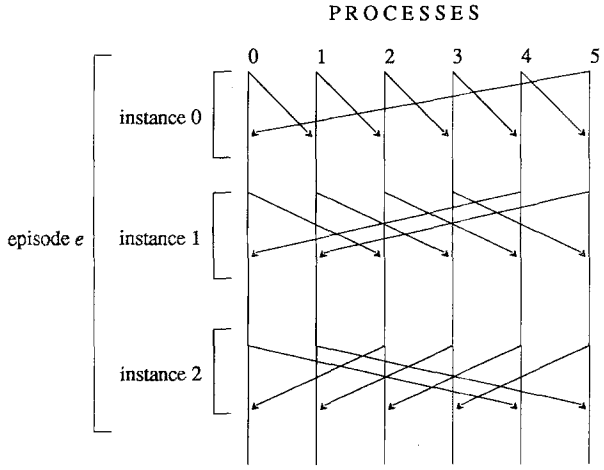

Fig. 4. Communication structure for the dissemination algorithm.

information originating at all other processes in exactly $\lceil \log_2 n \rceil$ rounds. Line (2) of an instance is equivalent to sending information and line (3) to waiting for and receiving information.

We extend the two-process barrier using this algorithm. Figure 4 shows the instances needed in the 6-process case.

Here, the message from process 2 to process 0 in instance 2 says, "I have entered episode *e*, I have heard from process 1 that it has also and I have heard from you that both you and process 5 have also." During this same instance, process 2 hears from 4 that 4, 3, 2, and 1 have also entered the episode. Therefore, process 2 knows that all the processes have entered episode *e* when it completes its last instance.

Pseudo-code implementing this algorithm is shown below. The dissemination algorithm may use a modified instance if the same number of processes participate in each episode. For clarity we use the original, not the modified, instance.

```
# define NPROCS 20 /* or any other size > 0 */
# define LogNPROCS 5
shared Answers[NPROCS][LogNPROCS]: Boolean;
    /* Answers[p, i] is the variable process p waits on in instance i */
    /* this wait appears in line (3) of the two-process barrier */
shared intended[NPROCS][LogNPROCS]: integer;
    /* intended[p, i] is destination for process p in instance i */
procedure InitBarrier;
var power, instance, process: integer;
begin
    power := 1;
    for instance := 0 to LogNPROCS-1 do
        for process := 0 to NPROCS-1 do
            intended[process][instance] := (power + process) mod
              NPROCS;
            Answers[process][instance] := false;
        end
        power := power*2;
    end
end InitBarrier;
procedure Barrier(myid: integer);
var instance: integer;
begin
    for instance := 0 to LogNPROCS-1 do
        while Answers[intended[myid][instance]][instance] do
          wait;
```

```
            Answers[intended[myid][instance]][instance] := true;
            while not Answers [myid][instance] do wait;
            Answers[myid][instance] := false;
        end
    end Barrier;
```

InitBarrier is called only once, after which Barrier may be called any number of times without reinitialization.

Theorem. This algorithm correctly implements a barrier for n processes.

Proof. We assume the correctness of the two-process protocol shown in Section 2.2. We need to show that no process will proceed to a subsequent epoch until all have finished the current epoch and that all processes will make progress.

First we show that if process p is executing in epoch e then no process has finished episode e (which follows epoch e). Since process p is in epoch e, it has not set the variable it was to set in instance i of episode e for any i in $\{0, 1, 2,..., \lceil \log_2 n \rceil - 1\}$. Any process id m between 0 and $n-1$ may be written as $m = p + \sum_{0 \leq i < \lceil \log_2 n \rceil} a_i 2^i \pmod n$ where a_i is chosen from the set $\{0, 1\}$. Let m be a process id and define $m_{Waiters} = \{j \mid a_j = 1\}$. If j is in $m_{Waiters}$, then there will be process waiting in instance j on a variable to be set by process p or by another process that is waiting in an earlier instance k also in $m_{Waiters}$. Finally, if $m \neq p$ then m will be waiting in instance $\max(m_{Waiters})$. Since it is waiting, m has not finished the current episode.

Now we must show that once all processes enter an episode, they will all exit it. We proceed by induction on the instance number i. Process p can continue past instance $i = 0$ because its partner, process $p - 1 \pmod n$, has also arrived at the episode. Similarly, process p will eventually continue past an arbitrary instance i because its partner, $p - 2^i \pmod n$, will also eventually finish instance $i-1$ (by induction) and will therefore arrive at instance i. □

2.5. The Tournament Algorithm

We can view the barrier requirement as a tournament. Only one process from each two-process game will continue to the next round. This asymmetry and the fact that we can decide in advance which process will win allow us to write a fairly simple two-process game:

	Process Winner	**Process Loser**
(1)	**while not** SetByLoser **do wait**;	SetByLoser := true;
(2)	SetByLoser := false;	**while not** SetByChampion **do wait**;

The barrier is a tournament with n processes. The overall winner announces the end of the contest by setting a shared variable.

Process OverallWinner

(3) SetByChampion := true;

After losing, a process continues to read this variable until it is set. At this point, all the contestants are finished with the tournament. Reinitializing the global variable after each episode can be achieved by double buffering or eliminated by sense switching.

There are $\lceil \log_2 n \rceil$ rounds, as in the dissemination algorithm. As before, the instances that constitute each round involve different variables. However, there are only $n-1$ games (after losing, a process sits out the rest of the tournament), so only $n-1$ variables are needed. Only $O(n)$ instructions are performed overall, as opposed to $O(n \log n)$ in the dissemination algorithm.

The communication pattern of the tournament algorithm is shown in Fig. 5 for 7 processes.

Below is pseudo-code for this n-process barrier.

```
# define NPROCS 20 /* or any other size > 0 */
# define LogNPROCS 5
# define CHAMPION 0
shared Answers[NPROCS][LogNPROCS]: Boolean;
    /* Answers are the variables waited on during the games */
    /* This wait appears in line (1) of the two-process game */
```

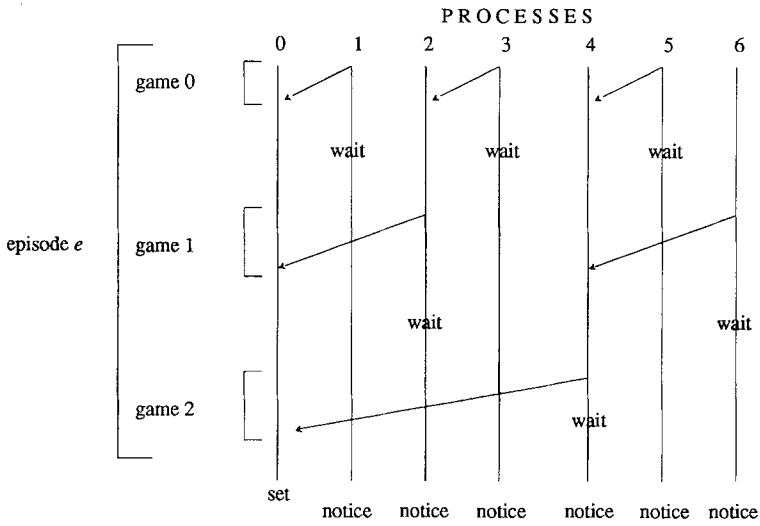

Fig. 5. Communication structure for the tournament algorithm.

```
shared Announcement [2]: Boolean;
    /* global announcement that tournament is over */
shared Opponent[NPROCS][LogNPROCS]: integer;
    /* Opponent[p, i] is opponent of process p in game i. */
shared BarrierCount: 0..1;
    /* which announcement? */
procedure InitBarrier;
var power, instance, process: integer;
begin
    power := 1;
    for instance := 0 to LogNPROCS-1 do
        for process := 0 to NPROCS-1 do
            Opponent[process][instance] := process xor power;
            Answers[process][instance] := false;
        end
        power := power*2;
    end
    Announcement[0] := Announcement[1] := false;
    BarrierCount := 0;
end InitBarrier;
procedure Barrier(myid: integer);
var instance: integer;
    LocalBarrierCount: 0..1;
begin
    LocalBarrierCount := BarrierCount;
    for instance := 0 to LogNPROCS-1 do
        if (myid mod 2^instance ≠ 0) then
            break; /* break out of loop; we are no longer active */
        end;
        if (myid > Opponent[myid][instance]) then /* loser */
            Answers[Opponent[myid][instance]][instance] :=
              true;
        elsif Opponent[myid][instance] ≥ NPROCS then
            /* win by default if no opponent */
        else /* winner */
            while not Answers[Opponent[myid][instance]]
              [instance] do wait;
            Answers[Opponent[myid][instance]][instance] :=
              false; /* reinit */
        end;
    end; /* for all instances */
```

```
        if (myid = CHAMPION) then
            BarrierCount := (BarrierCount + 1) mod 2;
            Announcement[BarrierCount] := false; /* reinit */
            Announcement[LocalBarrierCount] := true; /* all may
            proceed */
        else /* not the champion */
            while not Announcement[LocalBarrierCount] do wait;
        end;
    end Barrier;
```

Once again, InitBarrier must be called only once, after which Barrier may be called any number of times without explicit reinitialization.

We now argue that this algorithm correctly implements a barrier. Two invariants hold whenever any process exits Barrier or InitBarrier.

- The elements of array Answers are all false.
- The Announcement element to be used in the next episode is false.

The safety criterion is that no two processes may be in adjacent epochs. Suppose one process remains in epoch e. We need to show that no other process can finish the following episode e. If process 0 is the process in epoch e, then all other processes will eventually wait in episode e for Announcement[LocalBarrierCount] to be set. Suppose process m remains in epoch e, where $m = \sum_{0 \leqslant i < \lceil \log_2 n \rceil} a_i 2^i$, and at least one of the a_i is nonzero. Let $b_0, b_1, \ldots, b_k$ be the values of i for which a_i is nonzero, listed in increasing order. In game b_0, m's partner $m_1 = m - 2^{b_0}$ will block waiting for m. In game b_1, m_1's partner $m_1 - 2^{b_1}$ will block waiting for m_1. More and more processes will wait; at game b_k, process 0 will itself block waiting for 2^{b_k}. Therefore, process 0 will never set Announcement [LocalBarrierCount], and no other process will leave the current episode.

We now need to show that once all processes enter an episode, they will all exit it in a finite amount of time. If all processes exit the **for** loop in a finite amount of time, then all processes will complete the episode because one will set the Announcement variable to true and all of the rest will see it. We will show that all processes $p = \sum_{0 \leqslant i < \lceil \log_2 n \rceil} a_i 2^i$, $p \neq 0$, exit the **for** loop by induction on L, the smallest i such that a_i is nonzero. Then we will show that process 0 exits. All processes with $L = 0$ exit the loop because they set a variable to true and exit. Suppose all processes with $L = m$ have exited the loop. Then all processes with $L = m + 1$ will be looking at only variables set by processes with $L \leqslant m$ before they can set a variable to true and exit instance $m + 1$. But these watched variables have already been set to true (and cannot have been set back to false since process identifiers are unique), so those processes with $L = m + 1$ will exit

the loop. Since process $p = 2^{\lceil \log_2 n \rceil - 1}$ exits the loop by the above argument, then process $p = 0$ will also.

If the same number of processes participate in each barrier, then the second line of the winner's game can be removed by sense switching. We will call the resulting game a modified instance to maintain consistency with our previous terminology.

2.6. Discussion

Given a computer architecture with fetch-and-add, the best algorithm costs $O(\log n)$ time and uses only 1 variable. The following discussion assumes a model that allows concurrent reads and one write to the same variable, but not concurrent write to the same variable. (This model is weaker than CRCW, which allows concurrent writes.) The solution that uses exclusive access to a shared variable requires $O(n)$ time and 1 variable. The other solutions all require $\lceil \log_2 n \rceil$ instances, each of a constant time, so they (like fetch-and-add) require $O(\log n)$ time. They differ with respect to the cost of an instance. Table I shows the number of reads and writes needed by the two partners to an instance under various situations. This number dictates how long an instance will take. The total instances column indicates the product of instances and active processes summed over an episode.

3. TIMINGS

Figures 6 and 7 show the timings on a Sequent Balance 21000 for $2 \leqslant n \leqslant 20$ under the various methods. We wrote generator programs to construct the programs used in making the timings. These generator

Table I

Situation	Algorithm	Reads	Wites	Total instances
ordinary instance	Brooks, $n = 2^i$	2/2	2/2	$n\lceil \log_2 n \rceil$
	Brooks, $n \neq 2^i$	4/4	4/4	$n\lceil \log_2 n \rceil$
	Dissemination	2/2	2/2	$n\lceil \log_2 n \rceil$
	Tournament	1/1	1/1	$n-1$
modified instance	Brooks, $n = 2^i$	1/1	1/1	$n\lceil \log_2 n \rceil$
	Books, $n \neq 2^i$	2/2	2/2	$n\lceil \log_2 n \rceil$
	Dissemination	1/1	1/1	$n\lceil \log_2 n \rceil$
	Tournament	1/1	0/1	$n-1$

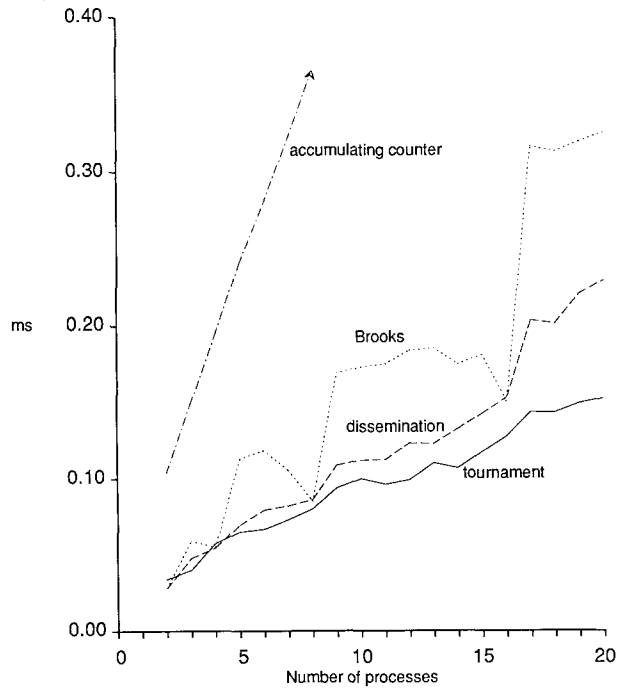

Fig. 6. Timings for the original instance.

programs take the number of processes to meet at the barrier and produce appropriate barrier code with all indices of the shared arrays reduced to constants. Loops are unrolled and macros used as in Brooks' timings. The code for method 1, exclusive access, uses double buffering to solve the reinitialization problem: every time an episode finishes using one counter, the other counter is reinitialized.

It is worth noting that Brooks' algorithm is identical in cost to the dissemination algorithm when the number of processes is a power of two. A glance at the code explains this result. Processes access the same number of variables in this case, but those variables have different array indices.

The curves for our algorithms appear more linear than logarithmic. As expected, the curves rise most sharply as the number of processes rises past a power of two due to the extra instance. The gradual linear rise can be attributed to linear-cost bus contention due to the write-through cache on the Sequent Balance 21000.

When different numbers of processes meet at different barriers, we must use an original instance. In this case, the tournament algorithm is the clear choice. When the same number of processes meet at all of the

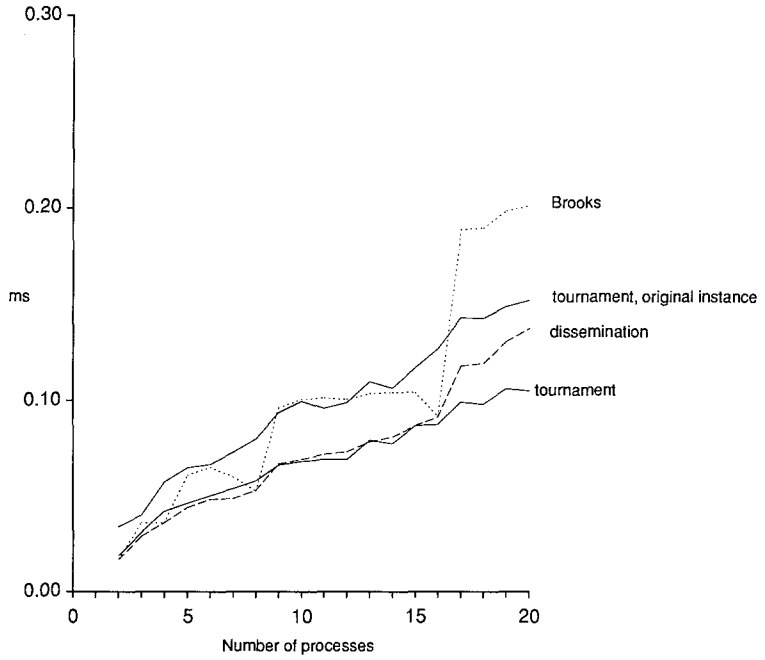

Fig. 7. Timings for the modified instance.

barriers, the modified instance may be used. When the number of processes is less than 16, the dissemination algorithm is competitive with the tournament algorithm. The extra code at the end of the tournament algorithm to notice that the global variable has been set incurs a small cost. This cost is not offset by improvement in bus contention when n is small. However, the difference between the number of writes in the dissemination algorithm $(2n\lceil \log_2 n \rceil)$ and the tournament algorithm $(n-1)$ becomes more significant for larger n.

REFERENCES

1. Eugene D. Brooks III, The Butterfly Barrier, *International Journal of Parallel Programming* **15**:295–307 (1986).
2. Boris D. Lubachevsky, An Approach to Automating the Verification of Compact Parallel Coordination Programs, *Acta Informatica*, pp. 125–169 (1984).
3. Leslie Lamport, On the Correctness of Multiprocess Programs, *IEEE Transactions on Software Engineering* **SE-3**(2):125–143 (Match 1977).
4. Hugh C. Lauer and Roger M. Needham, On the Duality of Operating System Structures, *Operating Systems Review*, pp. 3–19 (April 1979). Originally printed in the Proceedings of the Second International Symposium on Operating Systems, *IRIA* (October 1978).

5. Yijie Han and Raphael Finkel, An Optimal Scheme for Disseminating Information, Technical Report 106-88, Computer Sciences Department, University of Kentucky (1988).
6. Y. Han and R. Finkel, An optimal scheme for disseminating information, *Proceedings of the 1988 International Conference on Parallel Processing* **II**:198–203 (August, 1988).
7. N. Alon, A. Barak, and U. Manber, On Disseminating Information Reliably without Broadcasting, *The 7th International Conference on Distributed Computing Systems*, pp. 74–81 (September 1987).