International Journal of Parallel Programming, Vol. 15, No. 4, 1986

The Butterfly Barrier[1]

Eugene D. Brooks III[2]

Received December 1986; Accepted March 1987

We describe an algorithm for barrier synchronization that requires only read and write to shared store. The algorithm is faster than the traditional *locked counter* approach for two processors and has an attractive $\log_2 N$ time scaling for larger N. The algorithm is free of hot spots and critical regions and requires a shared memory bandwidth which grows linearly with N, the number of participating processors. We verify the technique using both a real shared memory multiprocessor, for numbers of processors up to 30, and a shared memory multiprocessor simulator, for number of processors up to 256.

KEY WORDS: Barrier; synchronization; butterfly; multiprocessor; shared memory.

1. INTRODUCTION

In this paper, we describe an algorithm to perform *barrier synchronization.* In this form of synchronization, all of the participating processors are required to *meet at the barrier* before any are allowed to proceed. The traditional approach to barrier synchronization has been to use a counter that records the arrival of processors. The waiting processors are released when the correct number N have arrived. The traditional approach has several critical regions and hot spots which cause the execution time for the barrier to grow linearly in the number of participating processors N. The butterfly barrier, in contrast, uses a distributed control structure that contains no hot spots or critical regions. The execution time for the butterfly barrier grows like $\log_2 N$.

[1] Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

[2] Parallel Processing Project, Lawrence Livermore National Laboratory, Livermore, California 94550.

0885-7458/86/0800-0295$05.00/0

828/15/4-2

The purpose of this paper is to describe the implementation of the butterfly barrier algorithm and to present its performance for both real and simulated multiprocessors. An analysis of the advantages $\log_2 N$ time scaling has been made by Axelrod[1] and we refer interested readers to that paper. We use the butterfly barrier on the Balance[3] 8000 and Balance 21000 parallel processors and have found it to be faster than the traditional approach for any number of processors. We have also run the butterfly barrier on a simulated shared memory multiprocessor, verifying its scaling properties for systems with up to 256 processors.

The organization of the paper is as follows. In Section 2, we describe the basic two processor barrier algorithm from the butterfly barrier is built. In Section 3, we describe the butterfly barrier algorithm itself, giving an argument for its correctness which depends on the correctness of the basic two processor barrier. In Section 4, we describe how cases for which N is not a power of two can be handled. We compare, in Section 5, the performance of the traditional barrier with that of the butterfly barrier on a bus based shared memory multiprocessor. The butterfly barrier always performs better than the traditional approach, but it's logarithmic scaling properties are violated for numbers of processors that saturate the available memory bandwidth. In Section 6, we examine the performance of the butterfly barrier, and the effects of accidentally introducing a hot spot, using a multiprocessor simulator. The simulator models a machine with a memory bandwidth that grows linearly with N, the number of processors, and the butterfly barrier takes advantage of the extra bandwidth. We consider some optimizations of the butterfly barrier in Section 7 and end with a discussion.

2. THE TWO PROCESSOR BARRIER

The butterfly barrier is constructed using a two processor barrier as the basic building block. Following the style of Lamport[2] we consider constructing a two processor barrier using a minimum of read and write operations of shared memory.[4] In order to effect a two processor barrier, each processor must first announce that it has arrived and then wait for its partner to arrive. This can be done using only read and write to two shared memory locations. The needed operations are described in the *C* code.

[3] *Balance* is a trademark of Sequent Computer System, Inc.

[4] If both processors arrive at the same time, each processor will execute two reads an two writes. More memory operations are executed if continued polling occurs.

```
                Processor 0              Processor 1
(1)     while(l0 != 0);          While(l1 != 0);
(2)     l0 = 1;                  l1 = 1;
(3)     while(l1 != 1);          while(l0 != 1);
(4)     l1 = 0;                  l0 = 0;
```

The shared memory locations required for the two processor barrier are $l0$ and $l1$. The workings of the code can be understood as follows, considering it from the perspective of processor 0. The algorithm is symmetric under interchange of the two processors. In statement (1) processor 0 asserts that $l0$ is zero. This is simply an assertion that the other processor has completed statement (4) from any previous barrier call. Without statement (1) two succeeding uses of the barrier could race with a set of $l0$ being lost. In practice this can happen quite frequently if the amount of work performed between barriers is small enough. In statement (2) processor 0 announces that it has arrived at the barrier. In statement (3) processor 0 waits for its partner to arrive. In statement (4) processor 0 resets its partner's notice of arrival and exits the barrier. The race condition that is being avoided by statement (1) is possible overlap of the set of $l0$ by processor 0 on barrier entry and the reset of $l0$ by its partner on barrier exit. The correctness of the algorithm may be proved using the methods of Lamport[3] and the proof is left as an exercise to the reader.

The basic two processor barrier has several desirable features that make it attractive as a building block for a larger N barrier. If the two shared memory locations $l0$ and $l1$ are implemented in separate physical memory banks the algorithm contains no hot spots or critical regions. If the shared memory is cache based, with a suitable cache coherence scheme, the while loops make effective use of cache polling. For a bus based shared memory system, this minimizes the effects of memory traffic on processors which are still performing real work and have not yet arrived at the barrier.

3. THE BUTTERFLY BARRIER

Considering the two processor barrier previously described as a basic building block, we can construct a barrier for any N that is a power of two using $\log^2 N$ stages of two processor parings. We show the individual two processor barrier interactions for an eight processor barrier in Fig. 1. The crossed arrows represent the individual two processor barrier interactions. The name *butterfly barrier* arises from the similarity of the interaction diagram to the butterfly diagram for the Fourier transform.

The correctness of the butterfly barrier, assuming the correctness of the basic two processor component, is easily shown. Consider the interac-

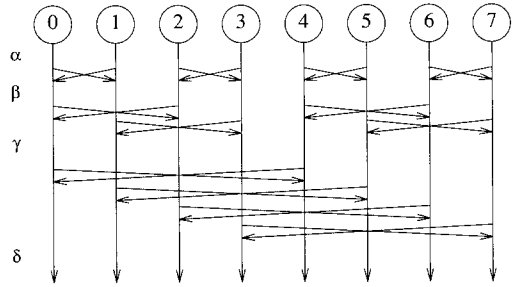

Fig. 1. Individual two processor barrier interactions for the 8 processor butterfly barrier.

tions of processors 0, 1, 2, and 3 as they proceed to point γ. We again consider things from the perspective of processor 0 noting the symmetries of Fig. 1. Processor 0 will not be allowed to arrive at point β until processor 1 arrives at point α, by virtue of its simple two processor barrier interaction with procesor 1 between points α and β. Processor 0 will not be allowed to arrive at point γ until processor 2 arrives at point β, by virtue of its interaction with processor 2 between points β and γ. Processor 2 will not be allowed to arrive at point β until processor 3 arrives at point α by virtue of its interaction with processor 3 between points α and β. This shows that processor 0 will not be allowed to arrive at point γ until processors 1, 2 and 3 arrive at point α. Considering the symmetry of the diagram, none of processors 0, 1, 2, and 3 will reach point γ until they all reach point α and we have, at point γ, a 4 processor barrier for this subset of the 8 processors.

To show the correctness of the 8 processor barrier at point δ, one considers the arrival of processor 0 at point δ. By virtue of its interaction with processor 4 between point γ and point δ it may not reach point δ until processor 4 reaches point γ. Processor 4 may not reach point γ until processors 5, 6, and 7 reach point α. We now have that processor 0 may not reach point δ and exit the 8 processor barrier until all of the other 7 processors have reached point α, that is, they have arrived at the barrier. Considering symmetry again, the same argument follows for other 7 processors and we have that the two processor barrier interactions depicted in Fig. 1 effect an 8 processor barrier.

By induction, we can build a correct N processor barrier for any number of processors equal to a power of 2. We can in fact give an argument for a correct $2M$ processor barrier given a correct M processor barrier for any M. This provides a hint for improving the efficiency of the implementation for N not a power of 2 that we will describe later. As each of the individual 2 processor barriers must be independent, the butterfly barrier requires $N \log_2 N$ shared memory locations which must be appropriately

indexed by the different 2 processor barriers. These shared memory locations are accessed without hot spots [see Ref. 4] if they are appropriately arranged in N memory banks.

4. THE ALGORITHM FOR N NOT A POWER OF 2

To construct a means of barrier synchronization for N not a power of 2 we use the data structures for the next power of 2 greater than N and arrange for existing processors to substitute for missing ones. Consider the simplest case, the 3 processor barrier. The computational flow graph for the three processor barrier is shown in Fig. 2.

Here the *missing processor* is processor 3 and we arrange that the actions of this missing processor are substituted for by processor 0. A general and easy to implement, but not the most efficient, scheme is to have a missing processor substituted for by the one obtained by reflecting the diagram. In this case processor 0 substitutes for processor 3 in each of the individual two processor barrier interactions it needs to participate in. In order to avoid deadlock, a processor that is performing substitutions must first perform the two notifications of arrival, one for the missing processor it is substituting for and one for itself, and then wait for its partner and the partner of the processor it is substituting for. This is done at each stage of the algorithm. In the case of the 3 processor barrier, at the first stage processor 0 will first indicate the arrival of the missing processor 3, then its own arrival, then assert that procesor 1 has arrived and finally assert that processor 2 has arrived for the missing processor 3. Although this careful treatment is not really needed to avoid deadlock in the 3 processor barrier, it is needed for other N, for instance the final stage of a 5 processor barrier.

5. PERFORMANCE ON A BUS BASED MULTIPROCESSOR

To show how the butterfly barrier can significantly reduce synchronization overhead, we compare the performance of the butterfly barrier

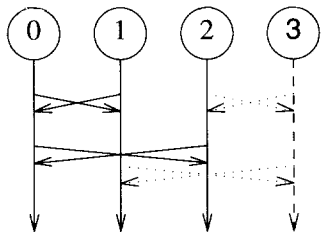

Fig. 2. The computation flow graph for the 3 processor barrier.

with the vendor supplied counter based barrier on the Balance 21000 multiprocessor. The Balance 21000 is a bus based shared memory multiprocessor that is manufactured by Sequent Computer Systems, Inc. of Portland Oregon. This machine can have up to 30 processors, each with its own cache, attached to a shared memory. The caches are kept coherent using a *write through* design in combination with bus watching to invalidate incoherent entries in other caches. The polling loops of the basic two processor barrier described in Section 2 take advantage of this cache design. The polling is performed within a cache and the bus is not subjected to the read requests until another processor writes to the polled location. Without this feature, the butterfly barrier would saturate the bus with read requests and the performance of processors that have not yet reached the barrier would be affected.

In Fig. 3 we compare the performance of the butterfly barrier with Sequent's counter based barrier implementation. Note that the execution time for the counter based implementation grows linearly in the number of processors. The butterfly barrier is faster than the traditional counter based implementation for any choice of the number of processors and shows a much more attractive scaling behavior. The execution time for the butterfly barrier for 2, 4, and 8 processors falls on a straight line in the semi-log plot.

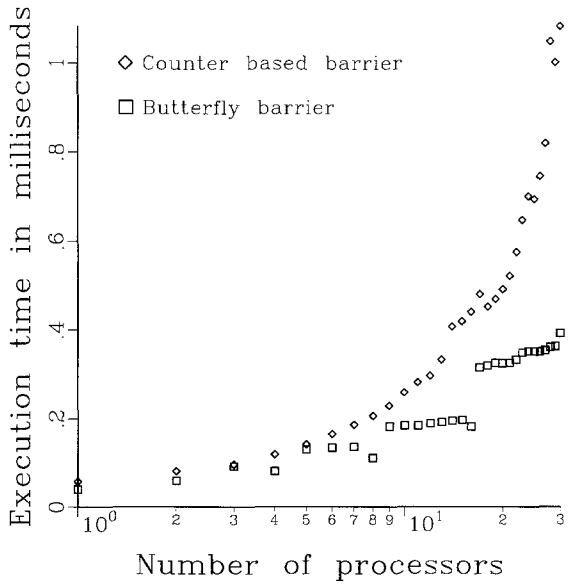

Fig. 3. Performance of the butterfly and counter based barriers as the number of processors is varied.

This confirms the logarithmic time scaling predicted in Section 3. For N not a power of two, the execution time for the butterfly barrier is greater than the time for the next N that is a power of 2. This is caused by the extra overhead of substitutions being performed as described in Section 4. These execution times could be improved using the methods described later. One should note that the execution time for the 16 processor barrier does not fall on the line obtained by extrapolating the timings for 2, 4, and 8 processors. This violation of the logarithmic scaling is being caused by bus loading. We will confirm this using the Cerberus multiprocessor simulator in the next section.

6. PERFORMANCE USING A MULTIPROCESSOR SIMULATOR

In addition to the performance masurements made on the Balance 21000 multiprocessor, we simulated the execution of the butterfly barrier using the Cerberus multiprocessor simulator. This simulator does a complete one pass simulation of a compiled program, using a pipelined RISC[5] instruction set for the processors and the indirect k-ary n-cube packet switched network[6,7] for the shared memory server. In Fig. 4, we show the execution time for the butterfly barrier as a function of the number of

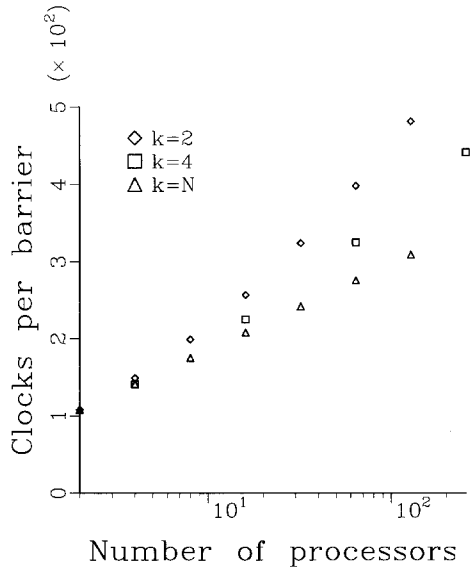

Fig. 4. Performance of the butterfly barrier using the Cerberus multiprocessor simultor.

processors. The results for $k = N$ are for a packet switched memory server which is a "full crossbar". The memory latency for this network choice, in the absence of memory bank conflicts, is independent of the number of processors. The execution times fall on a straight line for the semi-log plot. This unambiguously confirms the logarithmic scaling of the execution time for large N, if memory latency is a constant. The curves labeled by $k = 2$ and 4 are for shared memory servers constructed using 2×2 and 4×4 switch nodes, respectively. The exact architecture of these memory servers is described in Ref. 7. For these multistage memory server networks, the barrier execution time is affected by the increased memory latency which grows like $\log_2 N$ and $\log_4 N$ respectively.

It is interesting to investigate the consequences of inadvertently introducing a *hot spot*[4] into the execution of the butterfly barrier algorithm. Our implementation of the butterfly barrier is in the form of a subroutine call that all of the participating processors enter when they need to synchronize. The routine *barrier(iproc, nproc)* takes two arguments; *iproc*, which is used to identify the processor and *nproc*, which is used to indicate how many processors there are. *Iproc* must be in the range (0, *nproc*–1) and *nproc* must be identical for all of the processors. If one does not think carefully about the consequences of doing so, one might be tempted to put *nproc* in a shared memory location. Doing this introduces a

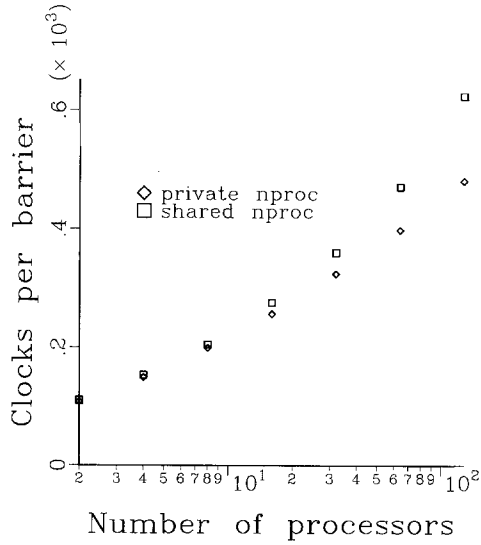

Fig. 5. Performance of the butterfly barrier with and without a hot spot.

hot spot into the execution of the code as each processor fetches the value of *nproc* to provide its value as an argument to the barrier call.

In Fig. 5 we show the effects of making this error. The execution time for the butterfly barrier, using the Cerberus multiprocessor simulator and an indirect binary n-cube network for shared memory server, is shown for *nproc* residing in both *shared* and *private* memories. As seen in Fig. 5, the hot spot introduced by making *nproc* shared has a significant effect on performance for numbers of processors beyond 32. The hot spot in fact dominates for large N and would cause the barrier call to take a time linear in N to complete. In spite of an algorithm that is free of hot spots and critical regions, a careless programmer can still degrade its performance by placing *nproc* in a shared storage location.

7. OPTIMIZATIONS

The performance of the butterfly barrier can be optimized by making careful choices for processors to do substitutions. Consider again a 3 processor barrier, referring to Fig. 2. If we substitute for the missing processor 3 with processor 2, processor 2 handshakes with *itself* in the first stage. This interaction can be removed from the 3 processor barrier and the resulting algorithm will be faster. Given the more efficient 3 processor barrier, a more efficient 6 and 12 processor barrier can be constructed by following the arguments of Section 3. This optimization strategy can be employed for other values of N.

Our implementation of the butterfly barrier uses the standard substitution scheme of Section 4 and the code for this implementation, written in *C* is presented in the Appendix. For applications where performance is critical, the optimizations previously described may be important.

DISCUSSION

We have presented a software algorithm for an efficient barrier synchronization. The algorithm is a practical one, and is the method of choice on our shared memory multiprocessor. The algorithm is free of critical regions and hot spots, and has an execution time that scales like the shared memory latency times $\log_2 N$. We have shown this scaling behavior using both an existing shared memory multiprocessor and a multiprocessor simulator. The butterfly barrier should be useful on any shared memory multiprocessor as long as there is enough memory bandwidth to support its execution.

Even though a bus represents a bottleneck, the butterfly barrier makes effective use of cache polling and its performance is still quite good on bus based systems, provided there are effective caches. On a bus based system without caches to support the polling, the increased memory traffic of the butterfly barrier might negatively impact the execution of processors that have not yet reached the barrier. Of course, one might use a more clever handshaking scheme, supported in hardware, that does not require polling. The object of this work is to create a high performance barrier algorithm without any special hardware support.

The butterfly barrier algorithm itself is free of hot spots and critical regions, but the user must be careful in using it. A hot spot can be introduced in the argument list to the barrier call and this can severely impact performance for large N. The butterfly barrier algorithm shows that we can implement synchronization primitives that are free of hot spots, but Murphy's law is always around the corner. Let the user beware!

ACKNOWLEDGMENTS

The author wishes to thank Gregory Darmohray who performed the simulations shown in Figs. 4 and 5. The author also wishes to thank Sequent Computer Systems, Inc. for access to the 30 processor Balance 21000 used to benchmark the butterfly barrier.

REFERENCES

1. T. S. Axelrod, Effects of Synchronization Barriers on Multiprocessor Performance, *Parallel Computing* **3**:129–140 (1986).
2. L. Lamport, The Mutual Exclusion Problem (to appear in *JACM*).
3. L. Lamport, A New Approach to Proving the Correctness of Multiprocess Programs, *ACM Transactions on Programming Languages and Systems*, **1**(1):84–97 (1979).
4. G. F. Pfister and V. A. Norton, Hot Spot Contention and Combining in Multistage Interconnection Networks, *IEEE Trans. Comput.* **34**(10):943–948 (1985).
5. D. A. Patterson, Reduced Instruction Set Computers, *Commun. ACM.* **28**(1):8–21 (1985).
6. E. D. Brooks III, A Butterfly Processor-Memory Interconnection for a Vector Processing Environment, *Parallel Computing* **4**:103–110 (1987).
7. E. D. Brooks III, The Indirect k-ary n-cube for a Vector Processing Environment (to appear in: *Parallel Computing*, LLNL, Livermore, UCRL 94529 (1986).

DISCLAIMER

APPENDIX

/* Barrier(whoami, howmany) implements a barrier for howmany processes. You tell it who you think you are and how many processes will participate. Whoami must be unique for each entering processor and must be in the range (0, howmany-1). Howmany must be identical for all of the calling processors, but beware of putting it in a shared memory location. This code is for up to 8 processors and does not contain any error handling.

```
        */
#define LNMAXPROC     3
#define MAXPROC       (1<<LNMAXPROC)
static shared int locks[LNMAXPROC][MAXPROC];
        /* Array for handshaking. */
#define UNLOCK(i, j)  {while(locks[i][j]);   locks[i][j] = 1;}
#define LOCK(i, j)    {while(!locks[i][j]);  locks[i][j] = 0;}
```

/* The WORK#(i) macro implements the work for an # level barrier. The argument i is who you are. The WORK#M(i) macro is the work section for a processor that has to perform substitutions.

```
        */
#define WORK1(i)      UNLOCK(0, (i^(1<<0))); LOCK(0, i);
#define WORK2(i)      UNLOCK(0, (i^(1<<0))); LOCK(0, i);\
                      UNLOCK(1, (i^(1<<1))); LOCK(1, i);
#define WORK2M(i)     UNLOCK(0, (i^0x3)^(1<<0));\
                      UNLOCK(0, (i^(1<<0)));\
                      LOCK(0, i); LOCK(0, (i^0x3));\
                      UNLOCK(1, ((i^0x3)^(1<<1)));\
                      UNLOCK(1, (i^(1<<1)));\
                      LOCK(1, i); LOCK(1, (i^0x3));
#define WORK3(i)      UNLOCK(0, (i^(1<<0))); LOCK(0, i);\
                      UNLOCK(1, (i^(1<<1))); LOCK(1, i);\
                      UNLOCK(2, (i^(1<<2))); LOCK(2, i);
```

```
#define WORK3M(i)       UNLOCK(0, ((i^0x7)^(1<<0)));\
                        UNLOCK(0, (i^(1<<0)));\
                        LOCK(0, i); LOCK(0, (i^0x7));\
                        UNLOCK(1, ((i^0x7)^(1<<1)));\
                        UNLOCK(1, (i^(1<<1)));\
                        LOCK(1, i); LOCK(1, (i^0x7));\
                        UNLOCK(2, ((i^0x7)^(1<<2)));\
                        UNLOCK(2, i^(1<<2)));\
                        LOCK(2, i); LOCK(2, i^0x7));
void barrier(whoami, howmany)
int whoami;     /* Who am I? (0 <= whoami < howmany) */
int howmany;    /* The number participating. (howmany >= 1) */
{
        swith(howmany) {
        case 1:
                return;
                break;
        case 2:
                WORK1(whoami);
                break;
        case 3:
                /* Space can be traded for speed if you want
                to turn this "if else" into a switch and have
                all the index computation done at compile time.
                        */
                if(whoami < 1) {
                        WORK2M(whoami);
                }
                else {
                        WORK2(whoami);
                }
                break;
        case 4:
                WORK2(whoami);
                break;
        case 5:
                if(whoami < 3) {
                        WORK3M(whoami);
                }
                else {
                        WORK3(whoami);
                }
                break;
```

```
case 6:
        if(whoami < 2) {
                WORK3M(whoami);
        }
        else {
                WORK3(whoami);
        }
        break;
case 7:
        if(whoami < 1) {
                WORK3M(whoami);
        }
        else {
                WORK3(whoami)
        }
        break;
case 8:
        WORK3(whoami);
        break;
```