A DISTRIBUTED HARDWARE BARRIER IN AN OPTICAL BUS-BASED DISTRIBUTED SHARED MEMORY MULTIPROCESSOR*

Martin H. Davis, Jr. and Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
davism@cc.gatech.edu and rama@cc.gatech.edu

Abstract

After defining our distributed shared memory multiprocessor architecture which uses an optically based interconnection network, we give a pure hardware optical barrier synchronization mechanism. Because of the current state of optical technology, we introduce a more realistic optical barrier mechanism, called the Distributed Shared Hardware Barrier *(DSHB), which is a combination of software and hardware (electronic and optical). We give a brief analysis of the cost of the DSHB scheme and show how this scheme combines the flexibility of software barrier techniques with the efficiency of hardware barrier techniques.*

Introduction

A necessary aspect of parallel programming is *synchronization*, the coordination of parallel processes and their activities. A frequently used synchronization construct is the *barrier*, first proposed by Jordan [10]. A barrier is a rendezvous point for some set of processes. The barrier operation comprises two parts, the *arrival* and *notification* phases. The arrival phase consists of the processes making their presence known at the barrier. When all the processes arrive at the barrier, then *barrier completion* has occurred. The notification phase entails the processes recognizing that barrier completion has happened and that they may proceed past the barrier.

Various software and hardware techniques have been proposed to implement barriers. Software techniques include the original centralized counter [10] and multiple counters (or flags) with various combining patterns [1, 7, 11, 14]. Hardware techniques include the original wired-`AND` gate [10], a tree of `AND` gates [12], combining trees embedded in the interconnection network switching hardware [4, 9], a bit-addressable register [13], wired-`NOR` synchronization lines [8], and a tree of `AND` gates augmented with "clearing" latches [5].

Because the barrier can conveniently implement various styles of parallel programming (e.g., the *fork-join* construct or `DOALL` loop parallelism [13]), it is imperative, as with all synchronization primitives, that the barrier mechanism be both flexible and efficient. Software techniques, though flexible, suffer from being inefficient. Hardware techniques, though efficient, are inflexible in implementing distinct barriers to be used simultaneously by several disjoint sets of processors and in allowing processes coexisting on the same processor to reach the same barrier. Hardware barriers, because of underlying electrical and physical properties, cannot be made arbitrarily large.

We propose a new *Distributed Shared Hardware Barrier* (DSHB) mechanism to be used in an optical broadcast ring based distributed shared memory multiprocessor. The DSHB combines the flexibility of software barriers with the efficiency of hardware barriers.

Optical barrier solutions

The architectural environment

The two traditional types of Multiple Instruction Multiple Data (MIMD) multiprocessor architectures have been *distributed memory* and *shared memory* machines. The *Distributed Shared Memory* (DSM) architecture combines the scalability of distributed memory with the flexibility of a global shared memory. The system's global memory is partitioned among the computing nodes, as in a distributed memory architecture. A computing node has direct, local access to its piece of memory, called the *Nearest Shared Memory* (NeSM) and has remote access, via the system's interconnection network, to all the other portions of the memory, the *Remote Shared Memory* (ReSM) modules. Thus, each computing node in a DSM system sees a global, shared memory albeit a node may access one portion (its NeSM) more easily and quickly than the other parts (ReSM). A computing node accesses either its NeSM or the ReSM via

*This work has been funded in part by NSF PYI Award MIP-9058430. A more detailed version is available as a Technical Report [3].

a cache to reduce the latency of memory references. The cache is connected to the NeSM via a standard electrical bus and is connected to the ReSM via an interconnection network. In our DSM architecture the interconnection network is implemented optically.

Because the bus topology is familiar and useful and because optical fiber transmission systems are commonplace, our research has been concerned with designing a DSM multiprocessor employing an optical "bus" topology, which we call the *Optical Broadcast Ring* (OBR). An important characteristic of the OBR is that messages propagate unidirectionally and that it is a broadcast medium (more details may be found in [2]).

Another important property of the OBR is its tremendous transmission capacity (on the order of Terahertz). One way to exploit this capacity is to split the large bandwidth into multiple (relatively) lower-speed channels. Each channel, which acts as an independent logical OBR, is assigned a unique wavelength. Since independent transmission wavelengths may coexist without interference on the same physical optical waveguide, our architecture allows multiple, independent OBRs.

Independent OBRs gives rise to an architectural design principle that we call the *separation of functions*. This design principle says that we may assign different kinds of interconnection network traffic (e.g., reads/writes, synchronization, cache coherency) to independent paths, which are OBRs in our research. Since barrier synchronization is a separate function and needs to be fast, we believe barrier traffic should be assigned to its own OBR.

Pure hardware optical barrier

One OBR channel (one wavelength on the physical waveguide) is dedicated for each barrier. The barrier is initialized by each participating process putting out a continuous optical signal on the channel. When a process reaches the barrier, it discontinues its optical signal. When all processes have reached the barrier, the channel becomes dark. This method is an optical `OR` representation of the barrier. The optical `OR` representation is better than the optical `AND` representation since the latter requires determining the precise optical signal level which represents reaching the barrier. Determining the precise optical signal level is not trivial. The optical `OR` representation only requires being able to detect a dark channel vs. a channel with any optical signal on it.

One potential problem with the optical `OR` barrier is the time for initialization and reuse. Initialization requires that each participating process see some raised signal level on the OBR before dropping its own signal to signify reaching the barrier. Reuse of the same physical barrier channel requires allowing enough time for the participating processes to recognize barrier completion. A more serious problem for pure hardware optical barriers is the limited number of channels available. Each channel requires a transmitter and receiver connection for each processor (or fast tunable transmitter and receiver). Current off-the-shelf optical technology limits the number of these channels to approximately a dozen. Therefore, it is not currently feasible to use pure hardware optical barriers since the dozen or so OBRs available must be used for purposes other than just barriers.

```
structure Barrier_type {

     integer Initializer,
             A_counter,
             B_counter;

     enum    Flag {A,B}
     }
```

Figure 1: The data structure for a barrier.

The Distributed Shared Hardware Barrier

Only one OBR channel, the Barrier Channel, is used for the DSHB, no matter how many logical barriers are used in the program. The Barrier Channel carries all the traffic related to all the barriers. Barriers are differentiated by their software representation, which is an extension of the traditional counter technique. However, instead of there being just one central data structure representing a given barrier, the data structure is replicated by every participating process as a private variable. When the barrier is initialized, every copy of the barrier is initialized to the number of processes participating in the barrier. When a process reaches the barrier, it broadcasts an arrival message to all the other processes stating it has reached that barrier. As a process receives these messages, it updates its private copy of the barrier. Thus, each process independently and asynchronously recognizes barrier completion. Processes may immediately continue past the barrier after recognizing barrier completion regardless of when other processes recognize that fact.

With the data structure shown in Figure 1, the barrier is immediately reusable. Let the barrier be implemented by two counters, designated as the A and B counters, a flag specifying which counter currently represents the barrier, and an initializer. When the barrier is initialized, the initializer is set to the number of participating processes, the flag points to counter A, and both counters A and B are set to zero. When a process reaches the barrier, since its flag points to counter A, it broadcasts a message to the other processes instructing them to increment the counter A. When a process recognizes barrier com-

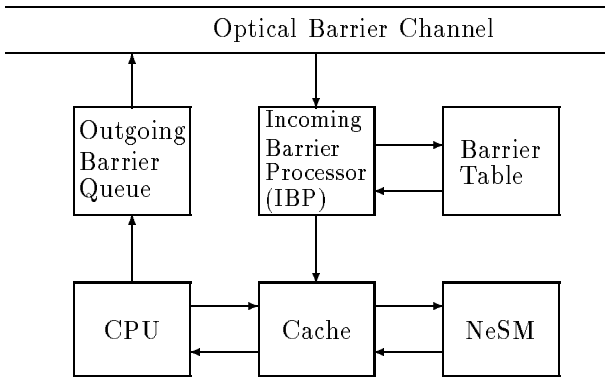

Figure 2: The hardware per node for a barrier.

pletion (counter A's value being equal to the initializer's value in its private copy), it resets its counter A to zero, then changes its flag to point to the other counter B. Since each process has a private copy of the barrier and asynchronously recognizes barrier completion, barrier notification is implicit. Further, depending on the relative speed of processes, barrier reuse may result in a potential race condition wherein a process starts receiving messages to increment the alternate counter B before it has changed its copy of the flag. This race condition is why a process resets its current counter after recognizing barrier completion and before switching the flag to the other counter. Once the process reaches the barrier again, it knows (because of the flag's value) to broadcast a message instructing the other processes to increment counter B. As the barrier is repeatedly used, counters A and B are alternately used, with flag pointing to the current instantiation of the barrier. If 4-byte integers are used in the data structure (allowing over 4 billion processes to rendezvous), it will fit in a typical 16-byte cache line.

Replicating the barrier as a private variable in each process is not the same as having a central shared variable cached at each processor and maintained in a coherent state by the hardware's cache coherency mechanism. Having a central shared variable, even if cached at each processor, still requires that processes must wait to be notified that the barrier has been reached since the cache coherency mechanism is a form of notification.

As shown in Figure 2, the barrier processing hardware is very simple. When a process reaches the barrier, it places a message in the Outgoing Barrier Queue specifying the barrier and the specific counter (A or B) to be incremented. The process either busy waits for barrier completion or is blocked if multitasking is allowed. The Outgoing Barrier Queue places its message on the optical Barrier Channel. Meanwhile, a separate Incoming Barrier Processor (IBP) receives messages on the Barrier Channel from other processes. The IBP places incoming messages into a queue of messages and processes one message at a time. It uses the contents of each message and the information in the Barrier Table to determine the virtual address of the appropriate counter to be incremented. If the IBP cannot find the barrier specified by a message in the Barrier Table, it discards the message since the implication is that no process located on that processor is participating in that barrier. The Barrier Table is structured such that processes using the same barrier can be located on the same processor. Therefore, the Barrier Table is initialized and managed by the operating system when the barrier is first requested. The IBP goes through the cache to reference the appropriate barrier's data structure. Since the IBP's only arithmetic function is to increment an integer, its logic can be optimized for this operation. Barrier completion is recognized by comparing the current counter's value against the initializer's value.

Since we have implicitly assumed that the Barrier Table is located in the NeSM of the processor, an optimization is to locate the Barrier Table in a small, separate, fast memory, possibly associative in nature. This optimization would allow the fast determination of the virtual address of the barrier data structure.

Gupta [6] has proposed a "fuzzy" barrier in which the barrier is a region of statements. The fuzzy barrier allows a process to perform useful work while waiting for other processes to reach the barrier. The DSHB technique can efficiently implement the fuzzy barrier semantics by defining the `hit-barrier` and `wait-barrier` operations. When a process has "reached" the barrier, it issues the `hit-barrier` operation, which translates into a message that the given barrier has been reached being placed into the Outgoing Barrier Queue. After the `hit-barrier` statement is performed, then if the process has instructions that do not depend upon barrier completion even though it has reached the barrier, those instructions are executed. When that supply of instructions is exhausted, the process waits for barrier completion by issuing the `wait-barrier` operation, which translates to the process waiting until the counter's value equals the initializer's value.

Discussion

In analyzing the DSHB scheme, we consider the cost spent in the two different parts of the barrier operation defined earlier, the arrival phase and the notification phase. The arrival phase cost has two parts, the number of messages sent and the time spent counting processes arriving. There are two arrival scenarios: *simultaneous* and *non-simultaneous*. Simultaneous arrival is defined as all processes arriving at the barrier at the same time. Non-simultaneous arrival is defined as $N-1$ processes arriving simulta-

neously, then the last process arriving sometime later. The cost of the notification phase is how many messages are sent out.

The DSHB technique generates N messages in the arrival phase. Since the Barrier Channel is a broadcast medium, all processes hear the N messages. In the simultaneous arrival case, the N messages must be processed serially; hence, each process takes N steps to update its copy of the counter. Since the processes are running in parallel, it takes N steps for all the processes to update their copies of the counters. In the non-simultaneous arrival case, assuming the previous $N-1$ messages have been acted upon, it takes 1 step for each process to act upon the last message (and 1 step overall for all the processes to act upon the last message). The notification phase generates *no* messages (and thus no network traffic) since each process independently determines barrier completion from its private, local copy of the barrier. Once the IBP begins processing arrival messages, it is likely that the barrier will stay in the cache through barrier completion. Hence, checking for barrier completion will be fast.

We believe the simultaneous arrival case is not very probable since it does not seem realistic to expect processes to finish their tasks simultaneously, particularly when the `hit-barrier` and `wait-barrier` commands are used. The unidirectionality of message propagation also makes the non-simultaneous arrival case more likely because processes located at different processors will receive barrier arrival messages at different times. Since the DSHB mechanism generates no messages in the notification phase, it is particularly advantageous for the non-simultaneous arrival case.

Since the DSHB scheme uses a software data structure located in standard memory, rather than a hardware resource such as an `AND` gate or bit-addressable register, it can support an arbitrary number of barriers. Since we provide dedicated hardware support for the transmission of barrier arrival messages and their processing, we overcome the speed limitation of pure software methods.

References

[1] E. D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[2] M. H. Davis, Jr. and U. Ramachandran. Optical bus protocol for a distributed shared memory multiprocessor. In *Optical Enhancements to Computing Technology*, July 1991. SPIE Volume 1563.

[3] M. H. Davis, Jr. and U. Ramachandran. A distributed hardware barrier in an optical bus-based distributed shared memory multiprocessor. Technical Report GIT-CC-92/18, College of Computing, Georgia Institute of Technology, 1992.

[4] E. Freudenthal and A. Gottlieb. Process coordination with fetch-and-increment. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 260–8, Apr. 1991.

[5] K. Ghose and D.-C. Cheng. Efficient synchronization schemes for large-scale shared-memory multiprocessors. In *1991 International Conference on Parallel Processing*, volume I, pages I–153–60, 1991.

[6] R. Gupta and M. Epstein. High speed synchronization of processors using fuzzy barriers. *International Journal of Parallel Programming*, 19(1):53–73, 1990.

[7] R. Gupta and C. R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3):161–80, 1989.

[8] K. Hwang and S. Shang. Wired-`NOR` barrier synchronization for designing large shared-memory multiprocessors. In *1991 International Conference on Parallel Processing*, volume I, pages I–171–5, 1991.

[9] D. N. Jayasimha. Distributed synchronizers. In *1988 International Conference on Parallel Processing*, pages 23–7, 1988.

[10] H. F. Jordan. A special purpose architecture for finite element analysis. In *1978 International Conference on Parallel Processing*, pages 263–6, 1978.

[11] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–78, Apr. 1991.

[12] M. T. O'Keefe and H. G. Dietz. Hardware barrier synchronization: Dynamic barrier MIMD (DBM). In *1990 International Conference on Parallel Processing*, volume I, pages I–43–46, 1990.

[13] C. D. Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, 37(8):991–1004, Aug. 1988.

[14] P. Tang and P.-C. Yew. Software combining algorithms for distributing hot-spot addressing. *Journal of Parallel and Distributed Computing*, 10:130–9, 1990.