Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters

Shisheng Shang, Member, IEEE, and Kai Hwang, Fellow, IEEE

Abstract—Conventional multiprocessors mostly use centralized, memory-based barriers to synchronize concurrent processes created in multiple processors. These centralized barriers often become the bottleneck or hot spots in the shared memory. In this paper, we overcome the difficulty by presenting a distributed and hardwired barrier architecture, that is hierarchically constructed for fast synchronization in cluster-structured multiprocessors. The hierarchical architecture enables the scalability of cluster-structured multiprocessors. A special set of synchronization primitives is developed for explicit use of distributed barriers dynamically. To show the application of the hardwired barriers, we demonstrate how to synchronize Doall and Doacross loops using a limited number of hardwired barriers. Timing analysis shows an $O(10^{5})$ to $O(10^{6})$ reduction in synchronization overhead, compared with the use of software-controlled barriers implemented in a shared memory. The hardwired architecture is effective in implementing any partially ordered set of barriers or fuzzy barriers with extended synchronization regions. The versatility, scalability, programmability, and low overhead make the distributed barrier architecture attractive in constructing fine-grain, massively parallel MIMD systems using multiprocessor clusters with distributed shared memory.

Index Terms—Barrier synchronization, distributed shared memory, Doall loops, Doacross loops, fuzzy barriers, parallel processing, partially ordered barriers, scalable multiprocessors, wired-NOR logic.

I. INTRODUCTION

A MULTIPROCESSOR cluster appears mostly as a bus-connected shared-memory system or as a cluster of tightly coupled workstations. These clusters are being used as building blocks for hierarchical construction of scalable multiprocessors, consisting of hundreds or even thousands of processing nodes. The memory can be physically distributed to each cluster, but logically shared among all multiprocessor clusters. For example, the Stanford Dash multiprocessor [21] is a scalable system using the SGI PowerStation 4D/340 as the building blocks. Each cluster consists of four processors tied to the common bus with a shared memory. Multiple bus-connected clusters are then interconnected by a wormhole-routed mesh network at the higher level. In Cray T3D/MPP system, hardwired barrier is implemented with dynamically partitioned ANDing tree and fanout circuits among distributed processing nodes [10]. The proposed hardwired barrier architecture appeals especially to cluster-structured multiprocessors.

In building a scalable multiprocessor, one must provide a fast synchronization mechanism for concurrent execution of software processes created within each cluster or among multiple clusters. In this paper, we propose a distributed and hardwired barrier architecture, which supports both intraccluster and intercluster synchronizations with low hardware/software overheads. The architecture is hardwired with a minimum logic overhead. The barrier wires are distributed over the local buses in multiple clusters, as well as to the higher level interconnect among clusters.

The major advantage of using distributed barrier wires lies in providing real-time responses with a significant reduction in synchronization overhead. This is especially important in supporting fine-grain MIMD operations in a large-scale multiprocessor. The conventional, memory-based, barrier synchronization is based on a centralized approach using shared variables in the common memory, or using a combining network, distributed caches, or a multicast network [3], [4], [8], [13], [14], [16], [20], [23], [33]. The main problem associated with centralized barriers is due to excessive software overhead or by hot spots experienced in the shared memory [18].

Our work was inspired by previous hardware approaches to high-speed synchronization. Lundstrom [22] suggested to use an ANDing hardware for barrier synchronization. In Cray T3D, barriers and eureka are implemented with hierarchically constructed ANDing trees and fanout circuits [10]. Bechmann and Polychronopoulos [8] used a hybrid approach for fast barrier synchronization. In the PAX computer [17], Hoshino suggested that each processor sets a simple bit to indicate its arrival at the barrier. ANDing/ORing logic is then applied to detect the arrivals of all processors. The hardware barriers suggested by O'Keefe and Dietz in [26], [27] demand the use of an intelligent compiler to schedule the operations. The Sequent computer uses special hardware to broadcast synchronization messages [6].

We use distributed wired-NOR circuits to detect the asynchronous arrivals of different processes at the barrier. The idea was first reported in a 1991 Conference paper [19]. This wired-NOR approach is easy to implement in a backplane bus or in a direct network interconnecting multiprocessor clusters. The scheme demands less hardware overhead, as compared with
the use of a combining network [28] or a multicast network [2] for the same purpose.

The paper is organized as follows: Section II describes the distributed barrier architecture, including the wired-NOR circuitry and a timing analysis of the potential speed gain. A set of synchronization primitives is developed in Section III for efficient use of the barrier wires in a time-sharing or multiprogrammed environment. We demonstrate how to synthesize Doall loops and Doacross loops using the barrier wires in Sections IV and V, respectively. Section VI shows a performance analysis of the proposed hardwired synchronization of parallel loops, compared with loop synchronization using memory-based barriers. Finally, we show how to use the wired-NOR circuits to implement fuzzy barriers or a partially ordered set of barriers. The conclusion section includes a summary of research contributions and comments on further research needed to integrate the proposed barrier architecture into scalable multiprocessor systems.

II. DISTRIBUTION OF HARDWIRED BARRIERS

The distributed barrier architecture is described below. We start with a simple multiprocessor cluster. We show how to extend the architecture into higher levels in a hierarchical manner. The synchronization overhead is estimated to reveal the timing advantages over software-controlled barriers.

A. Multiprocessor Clusters

The concept of a multiprocessor cluster is depicted in Fig. 1. The size of a cluster is limited essentially by the packaging technology. The cluster interconnect can be a backplane bus, a direct network, a crossbar, or a multistage network. The shared memory can be physically centralized or distributed. The proposed barrier mechanism can be applied to both memory organization, as long as a shared memory paradigm is adopted. The synchronization wires are implemented as part of the cluster interconnect.

The synchronization wires are part of the bus system of a multiprocessor cluster. In case of a multiprocessor cluster using a direct network, or a crossbar switch, or a multistage network, the collection of barrier wires forms essentially synchronization wires, that can be implemented on the backplane. Similar to any bus construct, the number of processors, memory modules, or I/O devices that can be tied to a common bus is limited. Based on today's packaging technology, the synchronization wires can be embedded in a backplane bus with 20–30 processor/memory/I/O boards.

Fig. 2 shows the barrier wiring and control circuitry for a typical multiprocessor cluster. We have proposed this wired-NOR architecture in [19]. A brief description of the wired-NOR logic is given below: There are \( m \) synchronization wires, and each wire can be used to implement an independent barrier across any subset of processors. Physically, each barrier wire is connected to all \( n \) processors, where \( n \) is the cluster size. Typically, the cluster size \( n \) lies in the range \((4, 32)\). The number of barrier wires \( m \) is limited by the degree of multiprogramming, and by the packaging or complexity constraints. A typical value of \( m \) is selected from the range \((4, 16)\) based on today's multiprocessor cluster technology.

Each processor \( i, \) where \( 1 \leq i \leq n, \) uses a control vector \( X_i = (X_{i,1}, X_{i,2}, \ldots, X_{i,m}) \) and a monitor vector \( Y_i = (Y_{i,1}, Y_{i,2}, \ldots, Y_{i,m}) \) for synchronization control. These vectors are mapped into the shared memory, or distributed to special registers in each processor board. Thus, they are program accessible from each processor. Each control vector can be dynamically read or written at run time, but each monitor vector is read only by one processor. Using a distributed control, this scheme requires no network transactions where processes are waiting for synchronization.

Each barrier line, labeled as \( j \) for \( 1 \leq j \leq m, \) is connected to \( n \) NPN bipolar transistors, associated with \( n \) processors separately. On the other hand, each processor is equipped with \( m \) transistors tied to \( m \) barrier lines as shown in Fig. 2. The base of each transistor is connected to a control bit \( X_{i,j}. \) The collector of the same transistor is monitored by a monitor bit \( Y_{i,j} \) at processor \( i. \) The control bit \( X_{i,j} \) is set or reset by processor \( i. \) When a barrier line \( j \) is at a high voltage level, all monitor bits \( Y_{i,j} \) for \( i = 1, 2, \ldots, n \) observe the logic level 1. Otherwise, a logic value 0 is monitored by all processors.

When any control bit is set high (a logic 1), the corresponding transistor is closed, which will pull down the voltage level of the barrier wire being monitored. A barrier line will be pulled up (a logic 0), only if all transistors tied to the same line are all reset low. For this reason, the set of transistors performs the wired-NOR logic. In other words, the wired-NOR logic is used to detect the completion of all processes (one from each processor) arriving at the barrier line asynchronously.

In Fig. 3, we illustrate how to use a single barrier wire to synchronize four concurrent processors in a single cluster. Each processor \( P_i \) is associated with one control bit \( X_i \) and one monitor bit \( Y_i \) for \( i = 1, 2, 3, \) and 4. Initially, all control bits are set to 1, pulling down the wired-NOR barrier to a low level. When a software process reaches the barrier, the corresponding control bit is reset to 0. When all four control bits are reset to 0, the barrier wire is pulled up, signaling the completion of the synchronization process. The barrier line is constantly monitored by all processors. The snapshots show the settings and resettings in all synchronization steps involved.

In [19], we have used the TTL circuits to estimate the latency of the synchronization wires. Typical electrical parameters suggest that the hardware responds in a few processor cycles, depending on how many processors are wired together. A single-level barrier wire can respond in 46 ns for a 20-processor cluster, or 5 processor cycles if 100-MHz processors are used.
B. Multilevel Barriers Architecture

The single-level synchronization wires can be expanded hierarchically to multiple levels of barrier wires, as illustrated in Fig. 4. Each multiprocessor cluster is supported by the first-level barrier wires. A repeater board is used to connect a pair of clusters, labeled as A and B, to a set of barrier wires at the second level, via a third port labeled C. Each repeater board has three pairs of control registers, labeled \((X_A, Y_A)\) for cluster A and \((X_B, Y_B)\) for cluster B. The third pair \((X_C, Y_C)\) is tied to the second-level synchronization wires. The repeater board monitors the voltage levels at both clusters through registers \(Y_A\) and \(Y_B\). The control registers \(X_A\) and \(X_B\) are used to maintain the consistency of voltage levels from both clusters being connected.

The voltage level of the second-level barrier lines is monitored constantly by vector \(Y_C\) and adjusted dynamically by vector \(X_C\), in order to propagate the wired-NOR logic operations from the first level to the second level. Assuming a 10 ns (or 100 MHz clock rate) processor cycle, the repeater board only experiences a delay of about 10 processor cycles. For two-level synchronization wires, 256 processors (divided into 16 clusters) can be synchronized in roughly 20 processor cycles, which is around 200 ns.

The time delay of such a hierarchical barrier architecture grows logarithmically with respect to the number of levels used in the hierarchy. For an example, a three-level hierarchy using another level of repeater boards requires only 30 processor cycles to synchronize 4096 processors divided into 256 clusters. This translates to about 300 ns for a 100 MHz clock rate used throughout the system.

C. Hardwired Synchronization Overhead

In memory-based synchronization, the atomic test-&-set instruction is often used. Each processor synchronizes by acquiring a lock, updating a barrier variable, releasing a lock, and polling that variable until the barrier is crossed. The barrier variable is stored in main memory and is not cachable. Let \(t_{lock}\) be the time to acquire and release a lock, and \(t_{mem}\) be
the access time to read a barrier counter in a shared memory. Assume $p$ processors reach the barrier at the same time. The time delay of using this centralized, memory-based test-and-set synchronization is estimated by:

$$T_{sw} = p t_{lock} + p t_{mem} + \frac{p(p-1)}{2} t_{mem}$$

$$= p t_{lock} + \frac{p(p+1)}{2} t_{mem}.$$  \hfill (1)

The first term accounts for the delay of $p$ processors to enter the critical section, to modify the counter, and to leave the critical section. The second term is the time delays of $p$ processors to read the barrier variable. The third term represents the time delay to poll the barrier variable by all processors, assuming a worst condition where each write to the counter is preceded by the reads from each busy-waiting processor. The values of $t_{lock}$ and $t_{mem}$ for a 20-processor Sequent Symmetry multiprocessor [1] are 5.6 $\mu$s and 100 ns, respectively. Thus the memory-based barrier scheme may experience a delay of 134 $\mu$s for $p = 20$ processors.

In a time-sharing or multiprogrammed system, using the wired-NOR barriers, a processor requires about 5.6 $\mu$s to update its local control vector and one memory cycle, say 100 ns to read its local monitor vector. The lock cycle of 5.6 $\mu$s ensures the integrity of data because different programs may update different bits in the same control vector. As a
TABLE I
SYNCHRONIZATION PRIMITIVES FOR USING THE HARDWIRED BARRIERS

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Function</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>create child processes</td>
<td>synchronization wires allocated</td>
</tr>
<tr>
<td>join</td>
<td>synchronize and destroy child processes</td>
<td>used by child processes only</td>
</tr>
<tr>
<td>sync</td>
<td>deallocate wires and rendezvous allocated</td>
<td>used by parent processes only</td>
</tr>
<tr>
<td>init_rendez</td>
<td>initialize a rendezvous</td>
<td>used by parent processes before forking</td>
</tr>
<tr>
<td>wait_rendez</td>
<td>synchronize at a rendezvous and reinitialize it</td>
<td>used by child processes only</td>
</tr>
<tr>
<td>entry</td>
<td>check in a barrier region</td>
<td>update control vectors</td>
</tr>
<tr>
<td>exit</td>
<td>check out a barrier region</td>
<td>read monitor vectors</td>
</tr>
<tr>
<td>rendez_entry</td>
<td>check in for a rendezvous synchronization</td>
<td>update control vectors</td>
</tr>
<tr>
<td>rendez_exit</td>
<td>synchronize at a rendezvous</td>
<td>read monitor vectors</td>
</tr>
</tbody>
</table>

result, the time delay \( T_{hw} \) for a processor to complete a barrier synchronization is less than 5.7 \( \mu s \). After the control vectors are reset, the wired-NOR mechanism can respond in 46 ns.

The time for the atomic lock instruction to update a processor's control vector may be reduced to one memory access time if the system allows a processor to update uninterruptedly its control vector. It takes about 200 ns to cross a wired-NOR barrier because no arbitration is needed. The above analysis indicates that for a 20-processor cluster, the wired-NOR mechanism is at least 30 times faster than that of a centralized, memory-based system for barrier synchronization.

To be more accurate, we plot in Fig. 5 the time delays \( T_{sw} \) and \( T_{hw} \), corresponding to software (memory-based test-and-sets) and hardware (the wired-NOR circuitry) approaches. The time delays are plotted as a function of the machine size, ranging from 2 to 525 processors. The \( T_{sw} \) curve shows a quadratic increase as the machine size increases. A 500-processor system can experience several milliseconds of delay in using a centralized, memory-based, and software-controlled barrier synchronization.

The delay of using hardwired barriers, \( T_{hw} \), is between 5.7 \( \mu s \) to 6.2 \( \mu s \) as shown in the blowup curve in Fig. 5(b). For a 500-processor system, the hardwired synchronization is at least one thousand times faster than the software approach. The quantum jumps at \( p = 67, 171, 275, 379, \) and 483, are due to each additional 100 ns cycle delay caused by the capacitance loading effect, when multiple levels are used.

III. BARRIER SYNCHRONIZATION PRIMITIVES

Synchronization primitives are developed to use the hardwired barriers as summarized in Table I. The first column lists the primitive names. The second column describes the functions performed. The third column specifies the hardware and software support needed. The last four primitives support fuzzy barrier synchronization to be described in Section VII.

These synchronization primitives are devised to operate in a time-sharing or multiprogramming environment, in which multiple programs can operate in an interleaved fashion using disjoint subsets of barrier lines. Therefore, multiple synchronizations can be carried out simultaneously across different program mixes on a dynamic basis.

These primitives are specially developed for locking or unlocking the control vectors and for requesting or releasing the synchronization wires. We assumed that the multiprocessor system adopts a modified run-to-completion static scheduling [34]. That is, each process once initiated continues executing uninterruptedly until completion. Thus process migration is disallowed under this assumption. When more than one process is allocated to a processor, the processor is time-shared in a round-robin fashion.

A. Syntax of Synchronization Primitives

We use a C-like syntax\(^1\) to specify the synchronization primitives. The synchronization wires are allocated upon the execution of a fork primitive, when child processes are created. The join primitive is used by each child process upon joining other child processes. The sync primitive is used to deallocate synchronization wires by the parent processes. The entry, exit, rendez_entry and rendez_exit are used to update the control vectors or to read the monitor vector by individual processors. Redevous can be initiated or reinitiated by using the init_rendez and wait_rendez primitives, respectively as exemplified at the bottom of the next page. Multiple child processes can be created by a single fork primitive. The number of processes created, nproc, is determined either statically at compile time or dynamically at run time. The join primitive causes all child processes to spin until all have arrived at the barrier. The sync primitive causes the parent process to spin until all child processes finish their tasks, and then deallocate the wire resources committed. The parent process may perform other computations, concurrently with the execution of its child processes.

To support repeated barrier synchronizations within the bodies of child processes, the init_rendez and wait_rendez primitives are used along with the fork, join, and sync primitives. The type specifier rendez, is used to declare rendezvous points. The init_rendez primitive initializes a rendezvous for child processes. The wait_rendez primitive put child processes in a busy-waiting state until all have arrived at the rendezvous. Then all child processes stop spinning and resume the execution. A rendezvous can be used many times by forked child processes. Results are not defined if the rendezvous was not previously initialized.

\(^1\) The type specifier void is used to declare functions that does not return values. The indirect operation '*' uses the value of '*' to determine a memory location where the value stored in that location can be retrieved.
B. Use of Synchronization Primitives

Fig. 6(a) shows a program flow graph for the fork, join, and sync primitives when \( l \) child processes are forked out concurrently from the parent process. The branch \( f_i \) denotes a task being executed by a child process \( i (1 \leq i \leq l) \), and \( T \) denotes the task being executed by a parent process. The child processes are terminated at the joining point specified by a join primitive.

A joining point is different from a rendezvous although they both are called barriers. The former allows the child processes to synchronize only once, while the latter allows them to synchronize many times. Fig. 6(b) illustrates a program flow graph using the init.rendez and wait.rendez along with the fork, join, and sync primitives when \( l \) child processes are created by a parent process. The subtasks executed by a child process \( i \) are denoted by \( f_i' \) and \( f_i'' \). The rendezvous \( bp \) can be used many times by the forked child processes. The fork and sync primitives support data partitioning by distributing different data segments to forked child processes. Passing different argument values to child processes will enable them to execute different code segments, the fork and sync primitives support function partitioning as well.

To ensure the correct accesses of shared variables required in the primitives, atomic instructions are needed. One possible atomic implementation is that they can be embedded in the underlying cache coherence protocol [14]. A write operation in cache obtaining the ownership of a particular cache block is equivalent to having exclusive access of the data block. Another possible implementation uses hardware locks.

After forking out the child processes, the fork primitive allocates processors and synchronization wires, initializes the data to be used by each child process for synchronization purpose, and sets the corresponding bits in the control vectors. When a child process finishes its tasks and starts to execute the join primitive, it resets a bit in the control vector to 0 and starts to probe that bit via the monitor vector. When all child processes reach the barrier, they read a value of 1 from the bit, stop probing the bit, and reset a bit in the control vector.

```c
fork(func[(arg1, arg2, ...)], nproc); init.rendez(bp, func, nproc);
void (*func)();
rendez *bp;
void (*func)();
unsigned int nproc;
unsigned int nproc;
wait.rendez(bp);
rendez *bp;
rendez.entry(bp);
rendez *bp;
rendez.exit(bp);
rendez *bp;
```
to indicate the barrier completion. When the parent process executes the *sync* primitive, it waits until resetting the barrier completion wire. The parent process then deallocates all the the synchronization wires used.

C. Repeated Barrier Synchronization

To support repeated synchronizations, the *init.rendez* primitive allocates additional barrier wires for each rendezvous points, initializes the values in those variables to be used by forked child processes, and sets the corresponding bits in the control vectors. Two passes are needed to avoid a racing problem in updating and reinitializing control vectors. Two subsets of synchronization wires are allocated and each is used for a different pass.

When a child process executes the *wait.rendez* primitive, the process first obtains the pass number, reinitializes the control vector for the next rendezvous, resets a bit in the control vector to 0, and starts to probe that bit via the monitor vector. When all child processes reach the rendezvous, they stop probing and update the pass number for the next rendezvous. When the number of processes is much greater than the number of available processors, additional wires are required for each active joining point. Child processes share the same set of wires when they are assigned to the same processor. Details of these primitives are specified in pseudo codes in Shang’s Thesis [30].

IV. SYNCHRONIZING ITERATIONS IN A DOALL LOOP

In Fortran-like Do loops, the loop iterations can be independent or dependent. A *Doall* loop contains iterations, which are all independent, and thus can execute in parallel by different processors. By independence, we mean the loop iterations are free from data dependence, control dependence, or resource dependence [32]. A *Doacross* [11] loop may contain iterations that are dependent of each other in terms of those dependence relationships. To speed up an application program on a parallel computer, the sequential Do loops must be converted into *Doall* or *Doacross* loops [12, 25, 29]. Once parallel loops are generated, synchronization instructions must be inserted to map these loops on the target parallel machine. The overhead of these synchronization instructions strongly affects the time to finish executing a loop in parallel. In this section, we concentrate on how to synchronize *Doall* loop using the proposed hardwired barriers.

An instance $S_m^i$ of a statement $S_m$ in a loop iteration identified by an index $I$ is defined as the instantiation of $S_m$ when $I = i$. Suppose $S_m^i$ depends on $S_m^j$, the dependence distance is defined as $d = j - i$. A dependence graph consists of nodes corresponding to statement instances, and directed arcs showing the dependence relationship among the nodes. In Figs. 7(b) and 8(b) the tail node of an arc is the dependence source ($S_2$), and the head node of an arc is the dependence sink ($S_4$). We consider only the case of constant dependence distance. The loops are normalized and singly-nested and do not possess conditional branches. The reason to focus on loops with constant distances is that they occur more frequently in numerical programs. Loops with variable dependence distances, or conditional branches, or loops with multiple nestings are beyond the scope of this paper.

A. Synchronization Instructions

In a synchronization process, one needs to signal that an action has occurred and to wait until an action has completed. Moreover, being able to reuse the synchronization wires is required with limited hardware resources. As a consequence, three synchronization instructions are specified below for this purpose: A *set* instruction signals that an event has occurred. A *wait* instruction allows waiting until an event gets completed; A *reset* instruction clears the signal of an event occurrence.
do I = 1, N
  
  S1: A(I) = B(I-2) + C(I)
  S2: B(I) = A(I) + D(I)
end do

(b)

(a)

Doacross I = 1, N
wait(f(I mod 6 + 1))
reset(f(I mod 6 + 1))

S1: A(I) = B(I-2) + C(I)
S2: B(I) = A(I) + D(I)

set(f((I + 2) mod 6 + 1))
end Doacross

(b)

(c)

(d)

Fig. 8. Synchronizing a lexical-backward-dependence Doacross loop. (a) Original program. (b) Backward dependence. (c) Annotated program. (d) Execution profile.

The syntax and semantics of the set, wait, and reset instructions are specified below, where j represents an integer, p is the processor executing an instruction, X is a control vector, and Y is a monitor vector associated with the barrier wires allocated.

\[
\text{set(j)}:
X_{p,j} = 0;
\]

\[
\text{wait(j)}:
\text{while} (Y_{p,j} \text{ .EQ. 0});
\]

\[
\text{reset(j)}:
X_{p,j} = 1;
\]

To signal an event’s occurrence in a loop iteration, the set instruction sets a control bit associated with the processor assigned to execute this iteration. The wait instruction waits for a monitor bit associated with the processor assigned to execute this iteration. After an event is detected by a wait instruction, a reset instruction resets the bit in order to assign the synchronization wire for executing other loop iterations.

B. Synchronization of a Doall Loop

The Doall loop is a special case of the Doacross loop with a zero dependence distance among the iterations. Doall loops can be self-scheduled [31], in which processors themselves determine what iterations to execute next. The iterations of a Doall loop are usually synchronized through a shared variable. Depending on the scheduling policy, processors schedule themselves by updating the shared variable to obtain a workload of one or more loop iterations. A barrier synchronization is needed to detect the completion of the Doall loops. Described below is a trivial self-scheduling algorithm for executing and synchronizing Doall loops.

Algorithm 1: (Synchronizing a Doall loop)

// J: loop index for a Doall loop with M iterations
// Initially J = 1
Request processors and obtain p processors: p1,p2,...,pp;
Request a synchronization wire f;
Activate processors to execute the assigned loop iterations; reset (f);

L1: if (J <= M) get some iteration(s) and update J;
if no more iterations are left, then goto L2;

 goto L1;

/ the original Doall loop body /

goto L1;

L2: set (f);
wait (f);

Processors will be busy waiting at the barrier until all complete their tasks and arrive at the barrier. After that, all processors can cross the barrier. There is an overhead to initialize the synchronization hardware before processors can execute the assigned loop iterations. The number of processors executing the loop is determined dynamically at run time. Note this loop synchronization mechanism is similar to the fork-join constructs discussed in Section III. The synchronization of Doall loops is rather simple. But the problem becomes much more involved with Doacross loops.

V. SYNCHRONIZATION IN DOACROSS LOOPS

To resolve data dependences, a set instruction is placed after dependence source, a wait instruction is placed before a dependence sink, and a reset instruction follows the wait instruction in order to reuse the synchronization wire once it is released. The main advantage of doing so is to allow more parallelism in loops to be exploited. Because the execution of the set instruction in an iteration does not need to wait, it is possible to execute the loop entirely in parallel if there is no constraint on hardware resources. We describe below a basic algorithm for synchronizing both types of Doacross loops, where loop iterations are folded to p processors, i.e., processor i will execute loop iterations i,i+2p,i+3p, etc., for 1 ≤ i ≤ p.
A. Doacross Loop Synchronization

In Algorithm 2 (shown at the bottom of this page), functions $g$ and $h$ map an integer pair, $\langle$loop index, dependence distance$\rangle$, to the logical address of a synchronization wire. These mappings are defined by $g(I, d) = (I + d) \mod (p + d) + 1$ and $h(I, d) = I \mod (p + d) + 1$. Function $f$ maps the logical address of a barrier wire to its physical location. The physical address ranges from 1 to $p + d$; the logical address ranges from 1 to $m$, where $m$ is the total number of the synchronization wires in the system. Note that functions $g$ and $h$ are both periodic. Their values do not have to be computed for each iteration of the loop. The mapping in function $f$ is determined dynamically at wire allocation time. The mappings in functions $g$ and $h$ are computed at compile time. It is possible to delay the processor allocation at compile time by inserting the value of $p$ dynamically at run time. As a result, the processor allocation can be made flexible. In using Algorithm 2, $p + d$ synchronization wires are needed, where $d$ is the dependence distance in the loop body. These barrier wires are sufficient to synchronize a Doacross loop with lexical-forward dependence. For backward dependence, the number can be further reduced to $d$ wires.

For a Doacross loop, the data dependence is either lexical-forward or lexical-backward [25]. In lexical-forward dependence the source of the dependence lexically precedes the sink, whereas in lexical backward dependence the sink of the dependence lexically precedes the source. Algorithm 2 is applicable to synchronize any Doacross loops, especially for a lexical-forward dependence loop. For lexical-backward dependence loops, only minor modifications are needed, which will be discussed later in this section, may further improve the capability of the algorithm. These two types of Doacross loops are discussed below separately. To simplify the analysis, we assumed a loop body containing only one dependence type, not mixed with both types.

B. Doacross Loops with Forward Dependences

Fig. 7(a) shows an example of a do loop with a lexical-forward dependence from statement $S_1$ to statement $S_2$. The dependence distance is one, as illustrated in the dependence graph in Fig. 7(b). In this example, statement instance $S_2$ in iteration $i$ must wait for statement instance $S_1^{i-1}$ in iteration $i - 1$ to complete. This waiting between statement instances is enforced by inserting three instructions—set, wait, and reset—between the source and the sink of the dependence relation. Assume that the system allocates four processors to execute this loop. Five synchronization wires are needed. We define $g(I, 1) = (I + 1) \mod 5 + 1$ and $h(I, 1) = I \mod 5 + 1$. The annotated program is shown in Fig. 7(c). Fig. 7(d) illustrates the execution profile of the loop iterations on four processors. Five synchronization wires are shared among different iterations. For example, processor $P_1$ uses wires $f(3)$ and $f(2)$, wires $f(2)$ and $f(1)$, wires $f(1)$ and $f(5)$, wires $f(5)$ and $f(4)$, and wires $f(4)$ and $f(3)$ periodically. The indices used in function $f$ needs to be computed only once.

C. Doacross Loops with Backward Dependences

Fig. 8(a) shows an example of a do loop with a lexical-backward dependence from statement $S_2$ to statement $S_1$. The dependence distance is two, as shown in the dependence graph in Fig. 8(b). In this example, statement instance $S_1^i$ in iteration $i$ must wait for statement instance $S_2^{i-2}$ to complete. This waiting between statement instances is enforced by inserting the three synchronization instructions in the loop body. Assume again four processors execute the loop iterations. Six synchronization wires are assigned. We define the functions $g(I, 2) = (I + 2) \mod 6 + 1$ and $h(I, 2) = I \mod 6 + 1$. The annotated program is shown in Fig. 8(c).

Fig. 8(d) illustrates the execution profile of the loop iterations on four processors. Six synchronization wires are shared among the iterations. Processors $P_1$ and $P_3$ share the wires $f(2), f(4)$, and $f(6)$. Processors $P_2$ and $P_4$ share wires $f(1), f(3)$, and $f(5)$. The indices of function $f$ need to be computed only once. Compared to the lexical-forward dependence loop in the previous example, some loop iterations are delayed for execution. Actually, two processors are sufficient to execute the loop in parallel. This is due to the dependence constraint within the loop body.

In general, if the number of processors allocated for executing a lexical-backward dependence loop is not greater than its dependence distance $d$, then $d$ synchronization wires are sufficient. The algorithm can be modified to deal with this case. As a result, the barrier hardware resource requirements for parallelizing a lexical-backward dependence loop can be determined at compile time, not necessarily at run time.

---

**Algorithm 2: (Synchronizing a Doacross loop)**

- $I$: index variable for a Doacross loop;
- $f, g, h$: integer functions;
- Allocate $p$ processors;
- for each dependence in the loop do
  - Allocate $p + d$ barrier wires.
  - If wire allocation fails, stop;
  - insert `set(f(g(I, d)))` immediately after the source;
  - insert `wait(f(h(I, d)))` and `reset(f(h(I, d)))` before the sink;
- end for loop;
VI. PERFORMANCE ANALYSIS AND COMPARISONS

In synchronizing parallel loop execution, the relative performance in using the hardwired barriers versus memory-based barriers is analyzed below. To simplify the analysis, we assume a single-level basic block structure in the loop body with a single entry point at the beginning and an exit point at the end of the block. The loop iteration carries no branch instructions. Loop-carried dependence may exist in the Doacross loop, either lexical-forward or backward but not both in the loop body. The dependence cycle has a distance \( d \) between a pair of instructions, labeled as \( S_1 \) and \( S_2 \).

The above assumption eliminates complicated dependence relations which prevent parallelization. The time to calculate or to access the loop index is amortized within the loop iteration. The following timing analysis extends from the general analysis performed in Section II as reported in Fig. 5. Again, we assume \( p \) processors, with a base processor cycle time \( \tau \). The loop body has a granularity of \( t \) cycles.

A. Use of Hardwired Barriers

Let \( T_p \) be the parallel execution time of a Doacross loop by \( p \) processors. The loop body will be executed \( M \) times. We can assume \( M \gg p \) in large-scale scientific applications. Ideally, \( p \) iterations are executed in parallel by \( p \) processors. Using hardwired synchronization, three special instructions: \textit{set}, \textit{wait}, \textit{reset} are used as described in Sections IV and V. This introduces \( 3\tau \) execution overhead in the loop body. For a Doacross loop with a forward dependence distance \( d \), the execution time is expressed by:

\[
T_p^{(f)} = \left[ \frac{M}{p} \right] \cdot (t + 3)\tau. \tag{2}
\]

For a Doacross loop with a backward dependence distance \( d \leq p \), only \( d \) loop iterations can be executed in parallel as a block. Thus we have the execution time:

\[
T_p^{(b)} = \left[ \frac{M}{d} \right] \cdot (t + 3)\tau. \tag{3}
\]

From the above time expressions, the loop iterations can be executed with a potential speedup factor of \( p \) or \( d \) over the sequential loop execution in \( M\tau \) time.

B. Use of Memory-Based Barriers

The memory-based barrier scheme [24], [25] assigns one synchronization variable in the memory for each statement that is a source of data dependence. The variable is shared among all instances of that statement. Two synchronization instructions are used: \textit{test} and \textit{testset} for this purpose. The \textit{test} instruction takes two arguments: a variable to be tested and a dependence distance; it waits until the dependence is resolved. The \textit{testset} instruction takes a variable as its only argument; it tests the variable to see if the previous iteration has completed; it then signals the event by incrementing the variable by 1. The scheme partially serializes the loop, because it updates the synchronization variables sequentially.

Let \( c \) be the total execution time of an atomic memory operation \textit{testset} in a memory-based synchronization process.

<table>
<thead>
<tr>
<th>Processor</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
<th>( P_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution</td>
<td>( S_1 )</td>
<td>( S_2 )</td>
<td>( S_3 )</td>
<td>( S_4 )</td>
</tr>
<tr>
<td>Sequence</td>
<td>( \text{test}(R) )</td>
<td>( \text{test}(R) )</td>
<td>( \text{test}(R) )</td>
<td>( \text{test}(R) )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( S_3 )</td>
<td>( S_4 )</td>
<td>( S_5 )</td>
<td>( S_6 )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \text{test}(R) )</td>
<td>( S_3 )</td>
<td>( S_4 )</td>
<td>( S_5 )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \text{test}(R) )</td>
<td>( S_3 )</td>
<td>( S_4 )</td>
<td>( S_5 )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \text{test}(R) )</td>
<td>( S_3 )</td>
<td>( S_4 )</td>
<td>( S_5 )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \text{test}(R) )</td>
<td>( S_3 )</td>
<td>( S_4 )</td>
<td>( S_5 )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \text{test}(R) )</td>
<td>( S_3 )</td>
<td>( S_4 )</td>
<td>( S_5 )</td>
</tr>
</tbody>
</table>

Fig. 9. The use of \textit{testset} to synchronize four concurrent software processes based on updating the same barrier variable in shared memory.

Usually, \( c = k\tau \) for some integer \( k \), which can be very large or very small depending on a number of machine and program factors, such as memory atomicity and machine size, etc. Thus \( c \) has been called a memory overhead.

Fig. 9 illustrates an example, where the synchronization of four concurrent loop is implemented with multiple \textit{testset} instructions updating the common barrier variable \( R \) in the shared memory. Four software processes update the shared variable asynchronously. Four processes \( p = 4 \), are available for the parallel execution, one loop iteration per processor.

When the loop body is small with a granularity \( \tau r < (p-1) c \), the original loop execution is hidden within the overhead of the \textit{testset} instructions. In other words, the execution of the loop body \( (S_1 \& S_2 \text{ instructions, etc.}) \) overlaps with the execution of the \textit{testset} instructions. This will occur, when the machine size \( p \) is large; the memory overhead \( c \) is high; and the loop granularity \( t \) is small. Thus, we obtain the following execution time:

\[
T_p^{''} = (M + p - 1) k\tau. \tag{4}
\]

When the loop body is sufficient large with \( \tau r \geq (p-1) c \), the executions of regular instructions and the \textit{testset} instructions do not overlap. Therefore, the total execution time increases linearly with respect to both \( t \) and \( c = k\tau \). In the following time expression, we attribute \( [M/p](\tau r + k\tau) \) cycles to the block iterations and \( (p-1)k\tau \) cycles to synchronization overhead. Thus, the total execution time is:

\[
T_p^{'''} = \left[ \frac{M}{p} \right] (\tau r + k\tau) + (p-1)k\tau. \tag{5}
\]

The case of a Doacross loop with a backward dependence distance of \( d \), Fig. 10 shows the execution sequences on four processors of a backward-dependence Doacross loop using the memory-based barrier scheme. A substantial amount of delay is incurred with updating the shared variable and waiting for dependence to be resolved. The execution time is similarly derived as follows:

\[
T_p^{''''} = \left[ \frac{M}{d} \right] \cdot [\tau r + (d + 1)k\tau]. \tag{6}
\]
C. Performance Comparison

In the above time expressions, $p$ and $\tau$ are two machine parameters. The remaining four parameters, $t$, $d$, $M$, and $c = k\tau$ depend on the program behavior. Among the six parameters, the performance is mostly sensitive to two parameters: the loop granularity $t$ and the barrier overhead $c$ associated with memory-based synchronization. In the comparison study, we consider a typical system with $p = 256$ processors driven by a 100 MHz clock with a $\tau = 10$ ns cycle time. The Doacross loop consists of $M = 8192$ iterations with a dependence distance of $d = 8$.

Fig. 11 plots the parallel execution times of the Doacross loop using hardwired barriers versus memory-based barriers. Part(a) corresponds to a loop with a forward dependence, while Part(b) for a backward dependence loop. In both loop types, the hardwired barrier performance increases linearly with respect to loop granularity $\tau$, using (2) and (3) respectively. The lower slope line corresponds to the monotonically increasing function. For a fixed loop size, the hardwired synchronization time is always lower than that of memory-based barriers.

For forward dependence, (4) is used to plot the execution time using memory-based barriers. This is shown by three flat curves in Fig. 11(a), corresponding to three memory overheads $c = 10\tau$, $c = 100\tau$, and $c = 1,000\tau$ respectively. For sufficiently large $\tau$, these curves coincide with the monotonically increasing curve (5) associated with hardwired barriers.

Using (6), the memory-based barrier performance is plotted in Fig. 11(b) for backward dependence with a distance $d = 8$.

The performance curves are lower than the corresponding ones in Fig. 11(a), but the slopes are steeper for finer granularity (smaller $\tau$). The memory-based barriers perform equally as well as the hardwired barriers for coarse grain loop body (very large $\tau$). The difference in performance gets widened for finer granularity. The larger is the memory overhead ($c\tau$), the worse will be the performance of memory-based barriers.

In Table II, we have estimated the potential speedup factors in using hardwired barriers as compared with the use of memory-based barriers. These factors are obtained from synchronizing the extreme case of fine-grain Doacross loop iterations, when the granularity approaches $\tau$.

Described below are the steps of using 5 wires to implement the five barriers in Fig. 12. Step 1 (shown on the bottom of this page) initializes all 5 barrier control vectors. All control bits are set to be 1. For instance, process $P_1$ sets bits $X_{1,1}$, $X_{1,2}$, and $X_{1,4}$. Process $P_2$ sets bits $X_{2,1}$, $X_{2,2}$, and $X_{2,4}$. Process $P_3$ sets bits $X_{3,1}$ and $X_{3,4}$, etc. Processes 1 through 5 are dispatched to processors 1 through 5 for execution, respectively. After the control vectors are loaded, the processes start to execute following the precedence ordering.
### Table II

Projected Speedup Factors of Using Hardware Barriers to Synchronize Doacross Loops, Compared with Memory-Based Barriers

<table>
<thead>
<tr>
<th>Loop Type</th>
<th>Processor</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. p</td>
<td>c = 2τ</td>
<td>c = 20τ</td>
</tr>
<tr>
<td>Lextra</td>
<td>p = 16</td>
<td>50</td>
</tr>
<tr>
<td>Forward</td>
<td>p = 256</td>
<td>500</td>
</tr>
<tr>
<td>Dependence</td>
<td>p = 4,096</td>
<td>8,000</td>
</tr>
</tbody>
</table>

### Processes

- **Step 1:** Initializing the barrier patterns (use 5 synchronization wires).
- When a process, say i, arrives at barrier 1, it resets control bit \(X_{i,1}\) and then constantly reads the monitor bit \(Y_{i,1}\). The process continues until \(Y_{i,1}\) becomes 1. When all five processes reach barrier 1, the voltage level on wire 1 is raised to high, as depicted in Step 2.

- **Step 2:** Synchronization at Barrier 1 (shown on the bottom of this page).
- Steps 3 and 4 illustrate two synchronizations at barriers 2 and 3, respectively. These two steps can be carried out in parallel, since they are independent of each other.

- **Step 3:** Synchronization at Barrier 2 (shown at the bottom of this page).

- **Step 4:** Synchronization at Barrier 3 (shown at the bottom of this page).
- Steps 5 and 6 show two synchronizations at barriers 4 and 5, respectively. Again, the two steps can be carried out simultaneously.

- **Step 5:** Synchronization at Barrier 4 (shown at the bottom of this page).
- **Step 6:** Synchronization at Barrier 5 (shown at the top of the next page).

It may require a lot of wires for an application with hundreds of barriers. The solution to this growth problem is to reuse the synchronization wires. After a barrier is crossed, the allocated

---

(a) Fig. 12. A set of 5 partially ordered barriers for 5 synchronizations. (a) Barrier coverage patterns. (b) Precedence graph.
wire is released and can be used by another barrier. The allocation and initialization of synchronization wires can be done by an intelligent compiler. The compiler inserts suitable routines in the software processes for this purpose. In case of hierarchically nested tasks with terminating barriers (a special case of partially ordered barriers), at most \( n - 1 \) wires are sufficient to synchronize \( n \) processes as proved by Beckmann and Polychronopoulos in [7].

B. Implementation of Fuzzy Barriers

To reduce overhead while processes are waiting for others to reach a barrier, the concept of fuzzy barriers was suggested in [9], [15]. A fuzzy barrier extends the conventional barrier concept to include a region of instructions that can be executed by a process while it awaits synchronization. Such a segment of instructions is called a barrier region. Barrier regions are constructed by a compiler. During the synchronization process, the software processes can execute at any point in the barrier region.

The larger the barrier regions are constructed, the more likely to eliminate unnecessary waits. Gupta [15] have proved that through program transformations, one can significantly increase the sizes of barrier regions. Fuzzy barriers are used for reducing penalties of spin-waiting. The fuzzy barrier mechanism provides tolerance to variations of process execution times. A region of instructions is specified rather than a specific point at which the processes must synchronize.

To illustrate the scenario of fuzzy barriers as opposed to regular barriers, Fig. 13 shows the execution profiles of a parallel program consisting of three processes. Fig. 13(a), shows the case of using a regular barrier. Processes \( P_1, P_2, \) and \( P_3 \) are synchronized at times \( t_5, t_6, \) and \( t_4 \), respectively. Processes \( P_1 \) and \( P_3 \) have to wait, until time \( t_6 \) when process \( P_2 \) reaches the barrier. That is, process \( P_1 \) sits idle from time \( t_5 \) to time \( t_6 \), and process \( P_3 \) sits idle from time \( t_4 \) to time \( t_6 \). Only after time \( t_6 \), the processes are allowed to continue executing new instructions \( I_{1}', I_{2}', \) and \( I_{3}' \).

The execution timing profile of the same program is illustrated in Fig. 13(b) using a fuzzy barrier. The sizes of the three barrier regions may differ, depending on how the compiler rearranges the instructions. None of the three processes has to wait in this case. For any process to exit the barrier region, the other two have already entered their barrier regions. Upon exiting their barrier regions, these processes continue executing the instructions behind the barrier without spinning. For example, while completing the barrier instruction, process \( P_1 \) continues executing instruction \( I_{1}' \), process \( P_2 \) continues executing instruction \( I_{2}' \), etc.

The strategy to support fuzzy barriers is to divide the functions in the join primitive into the entry and exit primitives indicating the beginning and the ending of a barrier region. Similarly, the wait.rendez primitive is divided into the rendez.entry and rendez.exit primitives. The entry/rendez.entry primitives update a control vector to indicate the entrance of a barrier region it guards. The exit/rendez.exit primitives read the monitor vector and wait until the barrier gets crossed over.
VII. CONCLUSIONS

The single-level wired-NOR barriers are shown effective to synchronize concurrent processes in a small-scale (say \( p = 16 \)), bus-connected, multiprocessor cluster. The speed gain in hardware barrier synchronization is shown between 10 to 10000 in small multiprocessor clusters. When multiple levels of hardwired barriers are used to construct large-scale multiprocessors, the latency of the hierarchical barrier architecture grows only logarithmically with respect to the number of levels in the hierarchy.

A 256-processor system can be built with a two-level of multiprocessor clusters with 16 processors in each cluster. The potential improvement in synchronization speed lies anywhere between 500 and 50000 in these medium-size multiprocessors. A 4096-processor system may require three levels of clustering. The potential synchronization speed gain lies between 8000 and 80000 in these large-scale systems.

The proposed hardwired synchronization is especially attractive to fine-grain, MIMD multiprocessor systems, where the synchronization overhead must be reduced to a manageable level. Although we have only used parallel loop synchronization to illustrate the idea, the distributed barrier synchronization can be applied to execute any program partitions, containing a large number of iterative computations. The sharing of the hardware barrier resources show a particular advantage when used in a multiprogramming environment, where the synchronization of different program segments can use different subsets of barrier lines.

The dynamic use of the shared barrier wires is still a wide-open research problem. We encourage further research be carried out in this area. The proposed hardware resources need to be better utilized with the help of self-scheduling compilers. More sophisticated runtime support system is also needed for dynamic use of the shared barrier hardware. Both compiler and OS directives could be used to help programmers, or compilers, or operating systems to enhance the performance.

For an example, \( \text{loop distribution} \) [5] can be used to split a Doacross loop into two Doall loops, which will create a large amount of parallelism. Another interesting extension of this work to explore the use of multicast networks for fast synchronization [2], [8]. The relative merits of proposed hardwired barriers and certain broadcast or multicast networks built with point-to-point direct networks should be revealed in extended studies.

In summary, the proposed hardwired barrier mechanism is shown effective to implement fixed barriers for Doall loops, fixed barriers for Doacross loops, fuzzy barriers, and multiple partially ordered barriers. It would be meaningful to see the claimed speed advantage be verified with prototype experiments in follow-up industrial settings. Physical constraints such as wire length and capacitance loading effects will be revealed from prototyping experiments.

ACKNOWLEDGMENT

The anonymous referees have made valuable suggestions to improve the quality and presentation of this paper. The authors are grateful to their efforts and appreciate Dr. David Padua’s suggestions in the revision process.

REFERENCES


Shisheng Shang (S’88–M’94) received the B.S. degree from the Taiwan University, Taipei, Republic of China in 1985, and the Ph.D. degree from the University of Southern California, Los Angeles in 1993, all in electrical engineering.

He is currently an Associate Professor in the Department of Computer and Information Engineering at the Kaohsiung Polytechnic Institute, Kaohsiung, Taiwan, Republic of China. His research interests include parallel system performance analysis and parallel computer architectures.

Dr. Shang is a member of the IEEE Computer Society and Phi Kappa Phi. In 1993, he received an academic achievement award from the Office for International Students and Scholars at the University of Southern California.

Kai Hwang (F’86) received the Ph.D. degree from the University of California at Berkeley.

He is a Professor of Electrical Engineering and Computer Science at the University of Southern California. He has engaged in computer research and higher education for 25 years. He is the author or co-author of 150 scientific papers, and five books in the areas of computer architecture, digital arithmetic, and parallel processing. His latest book, Advanced Computer Architecture: Parallelism, Scalability, Programmability (New York: McGraw-Hill, 1993), has been adopted by more than 40 Universities in the United States. He is the founding Co-Editor-in-Chief of the Journal of Parallel and Distributed Computing. His present research focuses on scalable multiprocessors using distributed shared memory. Presently, he collaborates with M.I.T. Lincoln Laboratory on benchmark evaluation of the SP2, T3D, and Paragon MPPs for real-time applications.