Hardwired Barriers for Fast Synchronization of Concurrent Processes on Scalable Multiprocessors

Shisheng Shang and Kai Hwang

CENG Technical Report 93-18

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4470

June 1993

Hardwired Barriers for Fast Synchronization of Concurrent Processes on Scalable Multiprocessors [1]

Shisheng Shang and Kai Hwang
Department of Electrical Engineering–Systems
University of Southern California
Los Angeles, CA 90089-2562

Abstract: This paper presents a scalable barrier architecture for fast synchronization of concurrent processes on large-scale, cluster-structured multiprocessor systems. The barrier architecture is distributed and hardwired to minimize network traffic generated and to provide low latency release of waiting processes. The barrier architecture is hierarchically constructed to provide scalability. It scales to support a shared-memory system of hundreds or even thousands of processors with a latency growing only logarithmically. The hardwired barrier is at least 1,000 times faster than a centralized software barrier scheme for a multiprocessor system with more than 500 processors. A set of barrier synchronization primitives is developed to use the proposed barrier architecture effectively. The architecture supports any partially-ordered set of barriers or fuzzy barriers with synchronization points expanded to regions. We demonstrate how to perform fast synchronization in *Doall* or *Doacross* loop execution using the hardwired barriers. Performance analysis shows appreciable speedups of hardwired barriers over memory-based barriers. The versatility, scalability, and low synchronization overhead make the barrier architecture especially attractive for supporting fine-grain parallelism in future massively parallel processors (MPPs).

Contents ..... Page

Index Term: Barrier synchronization, Doacross loops, Doall loops, fuzzy barriers, massively parallel processors, partially-ordered barriers, scalable multiprocessors.

[1] Manuscript submitted to *IEEE Transactions on Parallel and Distributed Systems*, June 1993. A preliminary version of this paper appeared in 1991 International Conference on Parallel Processing. For all future correspondence, contact Email kaihwang@panda.usc.edu, or FAX (213) 740-4449, or mail to EEB 200C, Dept. of EE-Systems, USC, Los Angeles, CA 90089-2562, USA.

1 Introduction

Barriers have been widely used for synchronizing concurrent processes in parallel processing. The concurrent execution of loop iterations is a good example. Barriers can be implemented in shared variables in memory, combining networks, coherent caches, and dedicated hardware. The software approach implements barriers with simple atomic lock/unlock instructions. The linear barrier, tree barrier, and butterfly barrier schemes [2, 3, 9, 25, 28] belong to this approach. The linear scheme uses a centralized barrier, while the others adopt distributed barriers. The difficulty of using a centralized barrier is that it may create a hot spot problem which will degrade system performance. Mellor-Crummey and Scott [16] have evaluated those barrier synchronization algorithms, requiring no special hardware support beyond the use of atomic memory operations.

The combining network approach utilizes special hardware to support simultaneous accesses to the same memory location. Hardware cost may increase sharply in using a combining network. For the coherent cache approach, lock mechanisms are implemented within cache controllers. Goodman, Vernon, and Woest [8] proposed a set of efficient primitives for process synchronization in large-scale shared-memory multiprocessors. Similar lock-based snoopy cache protocol scheme was proposed for bus-based systems [14].

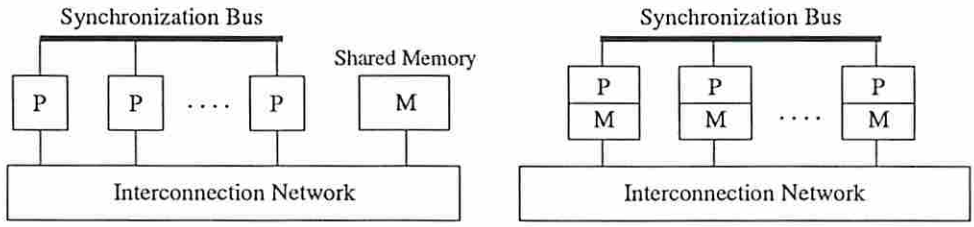
The Flow Model Processor (FMP) [15] was equipped with AND hardware for barrier synchronization. The Sequent Balance system uses special chips to support synchronization [4]. Using these chips, messages are broadcast through an exclusive bus, which may limit scalability. The PAX computer [11] uses a global synchronization hardware providing low-latency barrier synchronization. The method by O'Keefe and Dietz [19, 20] works in situations that barrier patterns are predicted at compile time. The scheme proposed by Beckmann and Polychronopoulos [5] uses a hybrid of distributed and centralized design.

A fast, flexible, inexpensive, and scalable barrier synchronization architecture is very much in demand for multiprogrammed, large-scale, shared-memory multiprocessors. We focus on designing a barrier architecture resulting in a minimum amount of traffic, while processes are waiting, and with low latency release of waiting processes. This paper is organized as follows. Section 2 describes the barrier synchronization architecture and hierarchical expansion schemes. Section 3 assesses the hardwired barrier advantages. Section 4 specifies primitives to use the barrier hardware. Section 5 implements partially-ordered barriers supported by the hardwired scheme. Fuzzy barrier is implemented in Section 6. Sections 7 and 8 present how to use the proposed synchronization architecture for parallel execution of *Doall* and *Doacross* loops. Sections 9 and 10 analyze the times needed to

execute the *Doall* or *Doacross* loops. Finally, we summarize the research contributions and discuss needed further work.

2 Hierarchical Barrier Architecture

A scalable multiprocessor is often constructed from many multiprocessor clusters. The size of a multiprocessor cluster is limited by packaging technologies. The interconnection network used can be a backplane bus, a mesh network, a crossbar, or a multistage network. The shared memory can be either centralized or distributed. The proposed barrier mechanism is architecture-independent, as long as a shared memory paradigm is adopted. This is illustrated in Fig. 1, where the shared memory can be either centralized or distributed. The synchronization bus is hardwired on the backplane bus or using special cables. This extra hardware can be hierarchically constructed with one or more levels.



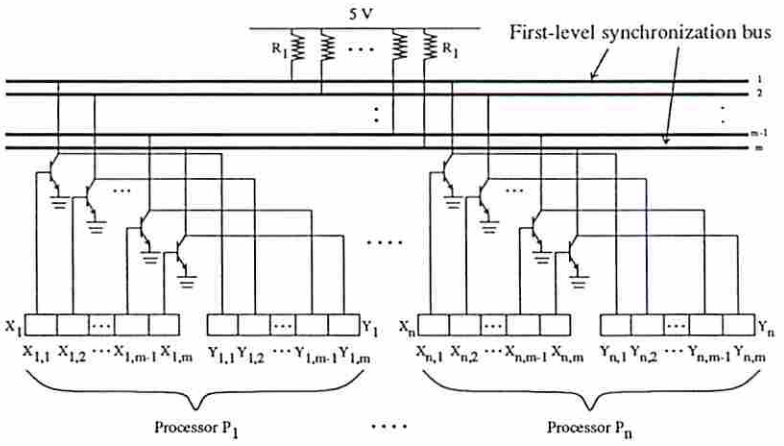
(a) Centralized shared memory (b) Distributed shared memory

Figure 1: Multiprocessor systems using hardwired synchronization buses. (P: processor, M: memory)

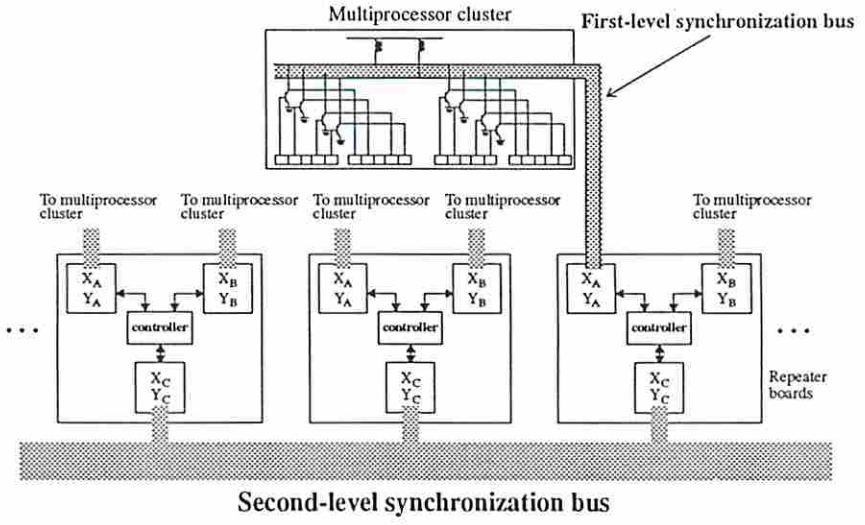
A barrier bus for each multiprocessor cluster is shown in Fig. 2(a). The architecture is extended for multiple multiprocessor clusters in Fig. 2(b). We use m wires across n processors for hardwired barrier synchronization. A wired-NOR, pull-down synchronization bus design was first proposed by the coauthors in [12]. The broadcast capability of this design reduces the hardware complexity. Each processor i, where $1 \leq i \leq n$, uses a *control vector* $X_i = (X_{i,1}, X_{i,2}, \cdots, X_{i,m})$ and a *monitor vector* $Y_i = (Y_{i,1}, Y_{i,2}, \cdots, Y_{i,m})$ for synchronization control. These vectors are mapped into the shared memory. Thus, they are program accessible from each processor. The control vectors can be read and written, but the monitor vectors are read only. Using distributed control, this scheme requires no network transactions, while processes are waiting for synchronization.

Each control vector X_i represents the location of a *set/reset circuit*, which affects voltage levels on the barrier wires. Each $X_{i,j}$ bit is the input to an NPN bipolar transistor attached to wire j. The monitor vector Y_i represents the location of a *probing circuit* for sensoring the voltages on all wires;

i.e. the $Y_{i,j}$ bit checks the output of the transistor tied to the j^{th} wire. Each wire is wired-NOR to n bipolar transistors, which are tied to n probing circuits.



(a) Hardwired barriers for a multiprocessor cluster



(b) Two levels of barrier wires for multiple multiprocessor clusters

Figure 2: The hierarchical barrier architecture for scalable multiprocessors.

Consider a particular wire j. When any $X_{i,j}$ $(1 \leq i \leq n)$ bit is high, the transistor is closed and it pulls down the current. The voltage level of wire j becomes low. When all $X_{i,j}$ bits become low, all transistors connected to barrier wire j become open. Thus, the voltage level of barrier wire j becomes high. It is this pull-down bus that enables fast barrier synchronization. Through a circuit analysis [12], we have shown that the hardware responds in a few processor cycles, depending on how many processors are wired together. In [12], we have proved that a single-level barrier bus can

respond in 46 ns for a 20-processor cluster, or in 540 ns for a 525-processor configuration.

Figure 2(b) shows a two-level barrier bus architecture that connecting the wires from two multiprocessor clusters, say A and B. The hardwired controller on the repeater board monitors the voltage changes of all wires on both clusters through monitor vectors Y_A and Y_B. The repeater board takes appropriate actions through control vectors X_A and X_B to maintain the consistency of voltage levels from both sides. Due to small capacitance in the repeater connects, the scalability of the bus hierarchy is assured. Since the delay on each repeater board is only a few processor cycles, the synchronization overhead grows only logarithmically with respect to the number of levels in the hierarchy. A two-level of barrier buses can synchronize 256 processors in 200 ns. A three-level scheme can support up to 4,096 processors in 300 ns.

3 Advantages of Hardwired Barriers

An analytical timing model is presented below to assess the performance of hardwired barriers. Each process consists of an indefinitely long sequence of subtasks separated by barriers, and each process is executed by a separate processor. Assume that the execution times T_i $(i = 1, 2, \cdots, p)$ of subtasks are *independent and identically distributed* (i.i.d.) random numbers with a mean μ and a variance σ^2. The variation of execution times is caused by memory conflicts, different processor speeds, different cache hit ratios, etc. Consider first the execution time of the p subtasks without barrier overhead. Let T denote the time at which the last processor completes its work. Thus, $T = \text{Max}(T_1, T_2, \cdots, T_p)$. The expectation of T, denoted E(T), was approximated by: $\mu + \sigma\sqrt{2 \log p}$, according to Kruskal and Weiss [13].

Next consider the barrier overhead t_b. Let t_1 and t_p be the execution times of a program by one and p processors, respectively. Clearly the value of t_1 is $p\mu$. The value of t_p is approximated as E(T) $+t_b$, assuming that variation of execution times σ and barrier overhead t_b are mutually independent. The value of t_b can be treated mainly as the release latency of all p processors when the barrier is reached. However, for certain implementations, t_b may decrease as σ increases. For simplicity, this effect is ignored in the timing model.

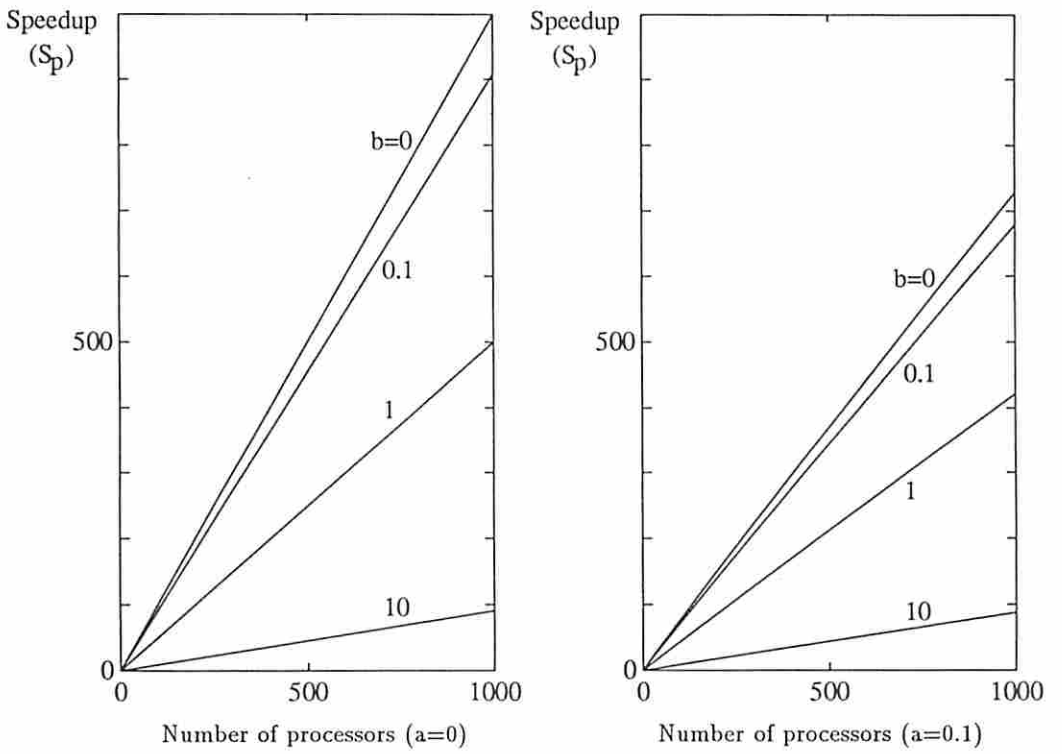
The *speedup* S_p is defined as the ratio of t_1 to t_p.

$$S_p = \frac{t_1}{t_p} \approx \frac{p\mu}{\mu + \sigma\sqrt{2 \log p} + t_b} = \frac{p}{1 + a\sqrt{2 \log p} + b} \tag{1}$$

where $a = \frac{\sigma}{\mu}$ and $b = \frac{t_b}{\mu}$. The parameter a is called coefficient of variation. For positive random variables such as T_i $(1 \leq i \leq p)$, a is likely a constant. The second term $a\sqrt{2 \log p}$ of the denominator

represents performance degradation due to the variation of process execution times. And the last term b represents performance degradation due to the ratio of the barrier overhead to the subtask granularity.

The resulting speedup curves for various machine sizes are shown in Fig. 3 for different values of a and b. When $a = b = 0$, an ideal linear speedup curve is observed. As a equals 0.1, the speedup degrades. That means, it is essential to distribute workload evenly among all processors in SPMD programs in order to keep a small. To maintain at least 90% of the maximum achievable speedups for different values of a, the value of t_b should be kept within 10% the value of μ. For applications with fine-grain parallelisms (μ is quite small), a fast barrier synchronization mechanism is thus strongly demanded.



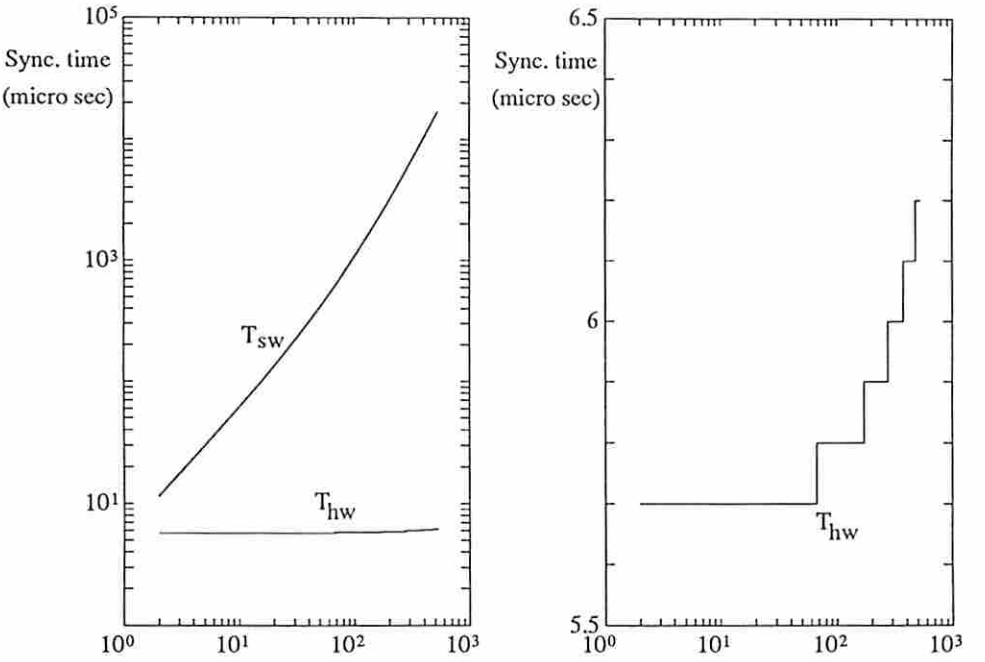
(a) Equal time spans in parallel branches (b) Unequal time spans in parallel branches

Figure 3: The speedup curves of parallel execution of multiple subtasks with different time spans.

The wired-NOR synchronization hardware is compared with the centralized barrier scheme. The time for synchronizing p processors with centralized barriers is estimated first. The *test-and-set* atomic instruction is used, and each processor synchronizes by acquiring a lock, updating a barrier variable, releasing a lock, and polling that variable until the barrier is crossed.

Suppose t_{lock} is the time to acquire and release a lock, and t_{mem} is the access time to read

a barrier counter in shared memory. Assume p processors reach the barrier at the same time. The time to cross the barrier using the centralized barrier scheme is thus estimated as $T_{sw} = pt_{lock} + pt_{mem} + \frac{p(p-1)}{2}t_{mem} = pt_{lock} + \frac{p(p+1)}{2}t_{mem}$. The first term corresponds to the time for p processors to enter the critical section, modify the counter, and leave the critical section; the second term is the time for p processors to read the barrier variable; the last term represents the time in polling the variable by all processors. The values of t_{lock} and t_{mem} for a 20-processor Sequent Symmetry multiprocessor are 5.6 μs and 100 ns [1, 22] respectively. Thus the centralized barrier scheme requires a time of 134 μs.



(a) Timing comparison

(b) Hardwired time (blowup view)

Figure 4: Synchronization time using the wired-NOR barrier compared with the centralized *test-and-set* synchronization through the shared memory.

Using the wired-NOR hardware mechanism for barrier synchronization takes 5.6 μs for a processor to update its local control vector and one memory cycle, i.e. about 100 ns, to read its local monitor vector. As a result, the time T_{hw} for a processor to complete a barrier synchronization is about 5.7 μs. After the control vectors are reset, the hardware mechanism responds within 46 ns.

The time for the atomic lock instruction to update a processor's local control vector may be reduced to a memory access time if the system allows only a processor to update its local control vector. Then, it only takes about 200 ns to complete a barrier because no arbitration is needed. The

above analysis concludes for a 20-processor multiprocessor system the wired-NOR barrier hardware mechanism is at least 30 times faster than that of the centralized barrier scheme using *test-and-set* atomic instructions.

The time difference between the centralized barrier scheme and the wired-NOR barrier mechanism will be even greater as the number of processors increases. Figure 4(a) shows the two plots of the times, T_{sw} and T_{hw}, as the number of the processors, p, ranging from 2 to 525. For a large multiprocessor, say $p > 500$, the hardwired synchronization is at least one thousand times faster than the centralized barrier scheme. Figure 4(b) illustrates a blowup of the hardwired synchronization time T_{hw}. The quantum jumps at $p = 67, 171, 275, 379$, and 483, are due to the additional 100 ns delay caused by capacitance loading effect.

4 Barrier Synchronization Primitives

Consider a multiprocessor using a modified *run-to-completion* static scheduling [29]. Each process once initiated continues execution uninterruptedly until completion. Process migration is disallowed. When more than one process is allocated to a processor, the processor is time-shared in a round-robin fashion. In case of multithreaded processors, the time-sharing of processors by multiple processes (contexts or threads) is realized by hardware.

Table 1 summarizes all the synchronization primitives we have developed to use the hardwired barriers. The first column are the primitive names. The second column list the functions performed. The third column specify the hardware and software supports needed.

Table 1: Summary of Primitives using Hardwired Barriers.

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

(Note: Details of these primitives are specified in pseudo codes in Shang's Ph.D. Thesis [23].)

Specify below in a C-like language[2] the syntax of *fork-join* construct, supported by the synchronization hardware:

```
fork(func[(arg1, arg2,···)], nproc);
   void (*func)();
   unsigned int nproc;

join();
sync();
```

```
init_rendez(bp, func, nproc);
   rendez *bp;
   void (*func)();
   unsigned int nproc;
wait_rendez(bp);
   rendez *bp;
```

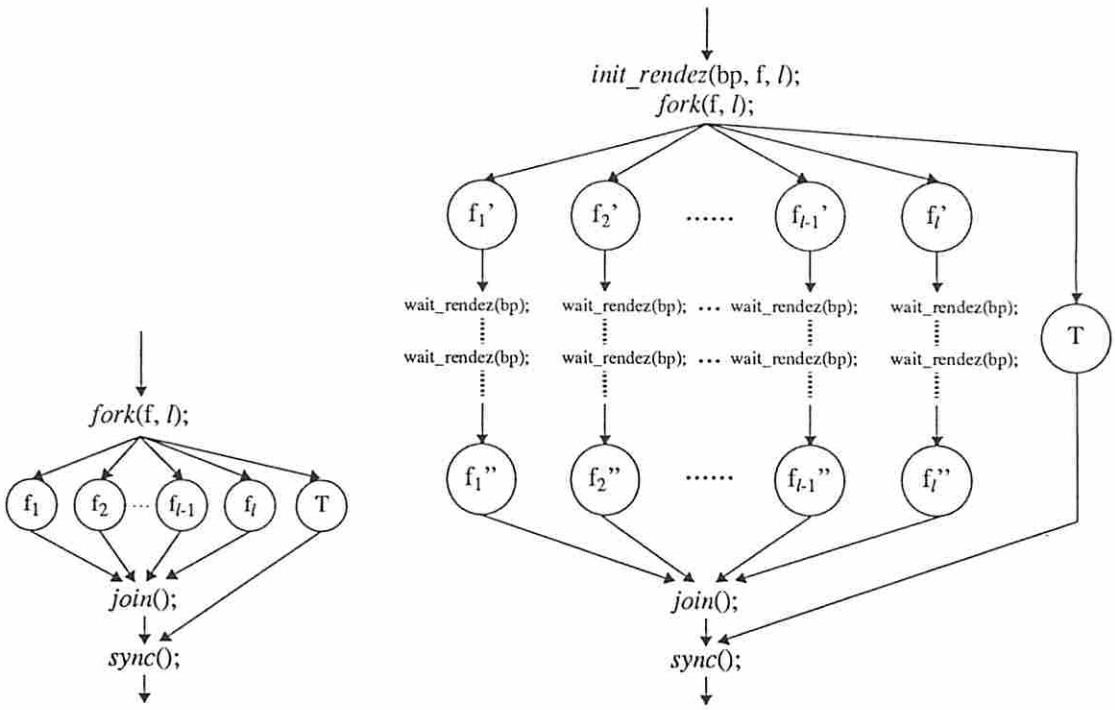
Multiple child processes are created by the *fork* primitive. The number of processes created, *nproc*, is determined either statically at compile time or dynamically at run time. The code needs not be duplicated for each child process; instead it is shared by all child processes. The *join* primitive causes all child processes to spin till all of them have arrived at the barrier. The *sync* primitive causes the parent process to spin till all of its child processes finish their task, and then deallocate the resources used by them. The parent process may perform other computations concurrently with the execution of its child processes.

To support repeated barrier synchronizations within the bodies of the child processes, the *init_rendez* and *wait_rendez* primitives are introduced. These two primitives must be 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 the child processes. The *wait_rendez* primitive put the child processes in a busy-waiting state till all of them have arrived at the rendezvous. Then all the child processes stop spinning and resume execution. A rendezvous can be used many times by forked child processes. Results are not defined if the rendezvous was not previously initialized.

Figure 5(a) shows the program flow graph for the *fork*, *join*, and *sync* primitives when l child processes are forked out concurrently with the parent process. The branch f_i denotes the task executed by child process i ($1 \leq i \leq l$), and T denotes the task executed by the parent process. The child processes are terminated at the joining point specified by the *join* primitive.

A joining point is different from a rendezvous though they are all called barriers. The former allows the child processes to synchronize only once, while the latter allows them to synchronize many times. Figure 5(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.

[2]The type specifier *void* is used to declare functions that does not return values. The operation $*v$ takes the value of v to be a memory location and returns the value stored in this location.



(a) Fork and Join

(b) Repeated barrier synchronizations

Figure 5: Program flow graphs using fork, join, sync, init_rendez, and wait_rendez primitives.

The subtasks executed by child process i are denoted by f_i' and f_i'', the task executed by the parent process is denoted by T. 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 [9]. 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 system 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 then 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 to indicate the barrier completion. When the parent process executes the *sync* primitive, it waits until the barrier completion wire is reset. Then the parent process deallocates all the the synchronization wires used.

To support repeated barrier synchronization, the *init_rendez* primitive allocates additional wires for each rendezvous points, initializes the values in several variables to be used by the forked child processes for synchronization purpose, and sets the corresponding bits in the control vectors. The subtlety in supporting repeated barrier synchronization is that two passes are needed to avoid a racing problem in updating and reinitializing control vectors. Two sets of synchronization wires are allocated and each is used for one pass.

When a child process executes the *wait_rendez* primitive, it first gets 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. However, child processes assigned to the same processor share the same set of wires.

5 Partially-Ordered Barriers

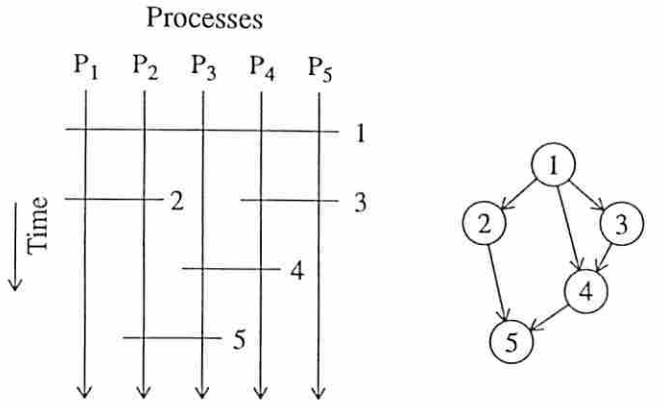
This section shows how to use the wired-NOR synchronization hardware to implement a partially-ordered set of barriers involving different process subsets. The mechanism can also support *dynamic barriers*, as described by O'Keefe and Dietz in [19], where the barrier patterns are predicted at compile time with no process preemption.

Partially-ordered sets are used to represent relations among concurrent barriers. Let a binary relation $<$ on a set of barriers $\mathcal{B}$ be a subset of the Cartesian product $\mathcal{B}^2$. The binary relation $<$ is a partial ordering because it is both irreflexive[3] and transitive[4]. The same notation used by O'Keefe and Dietz [19] is adopted to represent the barrier synchronization of five concurrent processes. As shown in Fig. 6(a), the vertical lines represent the execution time, while the horizontal lines represent the barriers. For example, processes P_1, P_2, P_3, P_4, and P_5 can not proceed barrier 1 till all of them have arrived there; processes P_1 and P_2 can not proceed barrier 2 till both have arrived there; processes P_4 and P_5 can not proceed barrier 3 till both have arrived there, etc.

From Fig. 6(a), the set of barriers $\mathcal{B}$ is $\{1, 2, 3, 4, 5\}$. The following binary relations are observed:

[3] A binary relation $<$ on $\mathcal{B}$ is irreflexive if $x < x$ does not exit for every x in $\mathcal{B}$.

[4] A binary relation $<$ on $\mathcal{B}$ is transitive if $x < y$ and $y < z \Rightarrow x < z$ for all x, y, and z in $\mathcal{B}$.



(a) Barrier coverage patterns (b) Precedence graph

Figure 6: Partial-ordered barriers.

$1 < 2$, $2 < 4$, $1 < 4$, $1 < 3$, and $3 < 5$. The partially-ordered set can be illustrated by a precedence graph (Fig. 6(b)), with the nodes representing the barriers and the arrows representing the ordering relations among them. Described below is an example showing the steps of using the synchronization wires to implement a sequence of five barriers shown in Fig. 6. Five barriers demand to use five synchronization wires, labelled as 1 through 5.

Step 1 initializes the barrier patterns. The corresponding bits in the local control vector for the barriers participated by each process are all set to 1. For instance, for process P_1, bits $X_{1,1}$, $X_{1,2}$, and $X_{1,4}$ are set to 1; for process P_2, bits $X_{2,1}$, $X_{2,2}$, and $X_{2,4}$ are set to 1; for process P_3, bits $X_{3,1}$ and $X_{3,4}$ are set to 1, etc. Processes 1 through 5 are dispatched to processors 1 through 5 for execution, respectively. After the barrier patterns are loaded, the processes start to execute.

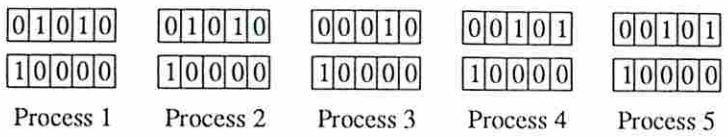
Step 1: Initializing the barrier patterns (use 5 synchronization wires)

	Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
	1	1	1	1	1
X	1 1 0 1 0	1 1 0 1 0	1 0 0 1 0	1 0 1 0 1	1 0 1 0 1
Y	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
	Process 1	Process 2	Process 3	Process 4	Process 5

When a process, say i, arrives at barrier 1, it resets bit $X_{i,1}$ in its local control vector, then starts to read bit $Y_{i,1}$ in the local monitor vector. The process does not stop reading the bit until

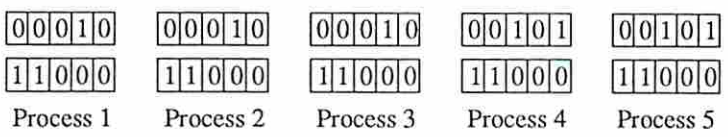
$Y_{i,1}$ becomes 1. When all five processes reach barrier 1, the voltage level on wire 1 becomes high.

Step 2: Synchronization at Barrier 1

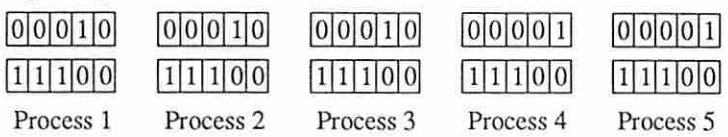


Steps 3 and 4 illustrate the synchronizations at barriers 2 and 3, respectively. We have assumed that Step 3 occurs before Step 4, but the synchronization of processes 1 and 2 is, in fact, independent of the synchronization of processes 4 and 5. The synchronization hardware can support the the synchronization of barriers 2 and 3 simultaneously.

Step 3: Synchronization at Barrier 2

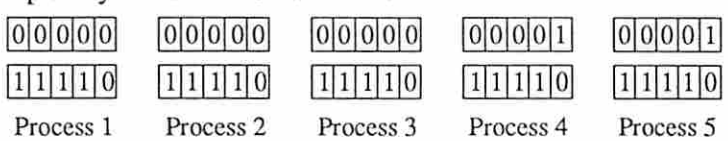


Step 4: Synchronization at Barrier 3

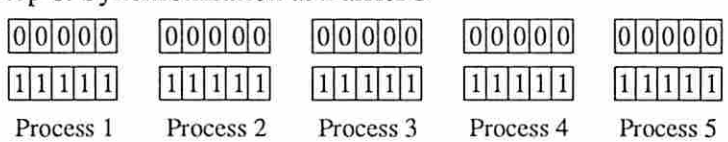


Steps 5 and 6 show the synchronization at barriers 4 and 5, respectively. Again, both steps can occur concurrently.

Step 5: Synchronization at Barrier 4



Step 6: Synchronization at Barrier 5



It may require a lot of wires for an application with hundreds of barriers. The solution to this problem is to reuse the wires. After a barrier is crossed, the allocated wire is released and can be used by another barrier. The allocation and initialization of synchronization wires may be done by a compiler. The compiler inserts suitable system routines in the processes for managing of barrier wires. As proved in [5], at most $n - 1$ wires are sufficient to synchronize n processors.

6 Fuzzy Barrier Implementation

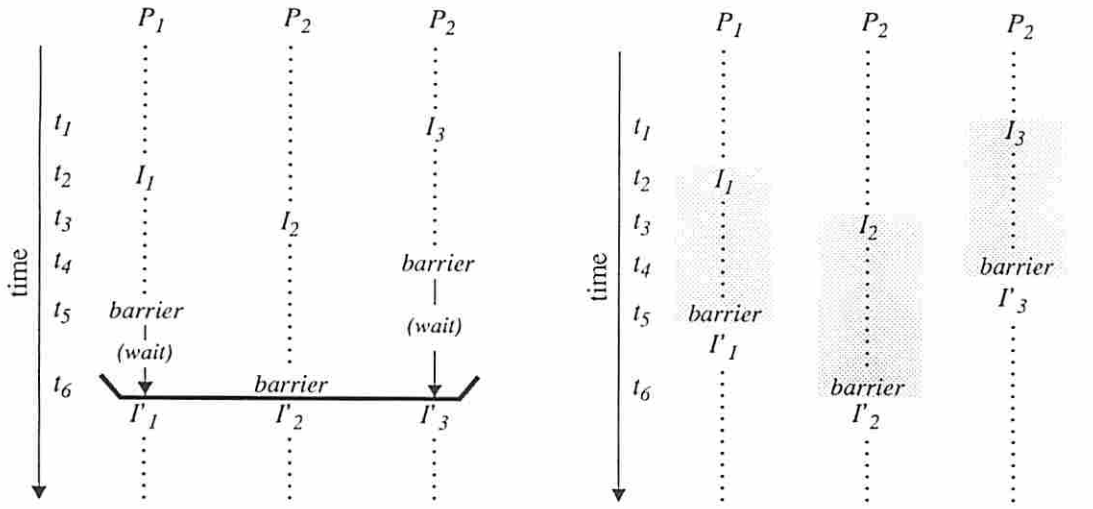
To reduce overhead while processes are waiting for others to reach a barrier, fuzzy barrier was suggested in [10]. Fuzzy barrier extends the barrier concept to include a region of instructions that can be executed by a process while it awaits synchronization. Such a region of instructions is called a barrier region. Barrier regions are constructed by a compiler. In each barrier region, the process is ready to synchronize upon reaching the first instruction and must synchronize before exiting it. When synchronization occurs, the processes may be executing at any point in their respective barrier regions.

The larger the barrier regions are constructed, the more likely it is that none of the processes will be waiting for the barrier. Previous study [10] showed that suitable program transformations 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 speeds. A region of instructions is specified where the synchronization is to take place rather than a specific point at which the processes must synchronize.

To illustrate the concept of a fuzzy barrier as opposed to a regular barrier, a scenario is shown in Fig. 7, where the execution timing profiles of a parallel program with three processes are given. As indicated in Fig. 7(a), where a regular barrier is used, processes P_1, P_2, and P_3 start the barrier synchronization 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* instruction. 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 the instructions I_1', I_2', and I_3' right behind the *barrier* instructions.

The execution timing profile of the same program but using a fuzzy barrier is illustrated in Fig. 7(b). The sizes of these three barrier regions may be different from one another, depending on how compilers identify and arrange the instructions. None of the three processes has to wait for synchronization in this case. Because for any process to exit its barrier region, the other two have already entered their respective barrier regions. Upon exiting their barrier regions, the processes continue executing the instructions behind the barrier without any 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.

Discuss next how to extend the previous defined synchronization primitives to support fuzzy barriers. The strategy to support fuzzy barriers is to divide the functions in the *join* primitive into the *entry* and *exit* primitives: one for indicating the beginning and the other for the end of a



(a) a barrier with two idle processes

(b) a fuzzy barrier without any idle processes

Figure 7: Execution timing profiles for a parallel program with three processes executing on three processors. (The shaded area represents barrier regions, I_k and I'_k are instructions executed by process k, the command *barrier* specifies a synchronization point.)

barrier region. Similarly, the *wait_rendez* primitive is divided into the *rendez_entry* and *rendez_exit* primitives. The *entry/rendez_entry* primitive updates a control vector to indicate the entrance of the barrier region it guards. The *exit/rendez_exit* primitive reads the monitor vector and waits till the barrier takes place.

7 Parallelism in Doall and Doacross Loops

An important source of parallelism in numerical programs is Fortran-like *Do* loops. To speed up an application program on a parallel computer, the sequential *Do* loops must be identified and converted into parallel equivalents [7, 17, 21, 24]. Once parallel loops are generated, suitable synchronization instructions must be inserted to map these loops onto efficient forms for execution on the target parallel machine. The overhead of these synchronization instructions strongly affects the time to finish executing a loop in parallel.

In loop constructs, two types of concurrent loops exist: *Doall* and *Doacross*. *Doall* loops do not have any data dependence across the loop iterations. *Doacross* loops [6] have data dependence across the loop iterations, i.e. an iteration of the loop may have to wait for signal from a previous iteration. Unlike the case of *Doall* loops, processors that execute different iterations of a *Doacross* loop may require frequent synchronization among processors themselves to ensure the correct execution order.

Data dependences [27] define a partial order on the statements of a program. We adopt the

following symbols. A statement instance S_m^i of a statement S_m that is contained in a loop with index I is defined as the *instantiation* of S_m when $I = i$. Suppose a dependence exists from S_x^i to S_y^j, the *dependence distance* is defined $d = j - i$. A dependence graph is a graph whose nodes are statement instances, and whose directed arcs represent the dependence relationship. In Fig. 8(b) and Fig. 9(b) the node at the tail of a directed arc is the *dependence source* (S_x), and the node at the head of the arc is the *dependence sink* (S_y).

This study is restricted to the case where the dependence distance is a constant, and 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 very frequently in numerical programs. Loops with variable rather than constant dependence distances, loops with conditional branches, or loops with multiple nestings are beyond the scope of this paper. Nevertheless, the methods developed here can be extended to solve those complicated cases.

A. Synchronization Instruction Set

In any type of synchronization, at least two capabilities are needed: the ability to signal that the action has occurred, and the ability to wait till an action has occurred. Moreover, a capability of reusing the synchronization hardware is demanded due to the limited hardware resources. As a consequence, a set of three synchronization instructions is proposed to utilize the synchronization hardware.

Defined below are three synchronization instructions: *set*, *wait*, and *reset*. The first capability is provided by the *set* instruction which signals that some event has occurred; the second capability is provided by the *wait* instruction which waits until an event occurs; the third capability is provided by the *reset* instruction which clears the signal of an event. The syntax and semantics of the *set*, *wait*, and *reset* instructions are given below, where j represents an integer parameter, p the processor that executes the instruction, X a control vector, and Y a monitor vector.

set(j):
  $X_{p,j} = 0$;
wait(j):
  while ($Y_{p,j}$ .EQ. 0);
reset(j):
  $X_{p,j} = 1$;

These instructions operate on the control vectors and monitor vectors of the barrier synchronization hardware. When processors use these instructions to access their local synchronization hardware, these accesses are treated simply as local memory accesses. Note no remote accesses via interconnection networks are generated.

To signal an event having taken place in an iteration of a loop, the *set* instruction sets a certain bit in the control vector attached to the processor that executes this iteration. The *wait* instruction in an iteration, waits for an event to occur several iterations before, by waiting for a bit in the monitor vector attached to the processor that executes this iteration. After an event is detected by the *wait* instruction on a bit, the *reset* instruction on that bit immediately follows in order to reuse the synchronization wire for executing another loop iteration.

Then the step of synchronizing iterations among processors becomes simply the finding of suitable integer functions that map iterations to the synchronization vectors. The general forms for these functions will be given later.

B. Synchronization of Doall Loops

The *Doall* loop is considered as a special case of the *Doacross* loop with a zero dependence distance. A commonly used scheduling strategy for *Doall* loops is *self-scheduling* [26], in which processors themselves determine what iterations to execute next. The iterations of a *Doall* loop are usually scheduled through a shared variable. Depending on what particular scheduling policy is used, processors schedule themselves by updating the shared variable to get loop indices of one or more iterations. Nevertheless, after all iterations are scheduled, a barrier synchronization is needed for completion of the *Doall* loop. Described below is a processor self-scheduling algorithm for *Doall* loops.

Algorithm 1: (for synchronizing a Doall loop)

```
   / J: loop index for a Doall loop with M iterations /
   / Initially J = 1 /
Serial Part:
   Request processors and obtain p processors: p_1, p_2, ..., p_p;
   Request a synchronization wire and obtain wire ℓ;
   Activate processors to execute the loop iterations;
Parallel Part:
        reset(ℓ);
   L_1: if (J <= M), get some iteration(s) and update J;
        if no more iterations are left, then goto L_2;
        .
        .
        .     / the original Doall loop body /
        .
        .
        goto L_1;
   / Processors finish their tasks and go back to get more iterations. /
   L_2: set(ℓ);
        wait(ℓ);
```

Processors will be busy-waiting at the barrier until all of the processors complete their tasks

and arrive at the barrier. After that, all processors can pass through. There is a serial step for initializing the synchronization hardware before processors can proceed to execute the loop iterations. The number of processors executing the loop is determined dynamically at run time. Note that this loop synchronization mechanism in some way is similar to the *fork-join* constructs that we discussed previously.

8 Synchronization in Doacross Loops

When synchronizing dependences using the instruction set, the *set* instruction is placed after the source of the dependence to signal that the source has been executed, the *wait* instruction is placed before the sink to ensure that its source has been executed, and the *reset* instruction immediately follows the *wait* instruction and precedes the sink in order to reuse the synchronization wire.

The main advantage of this instruction set is that it allows 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. The loop iterations are pre-scheduled and are folded to p processors. That is, processor i will execute loop iterations i, $i+2p$, $i+3p$, $\cdots$, etc., where $1 \leq i \leq p$.

Described below is an algorithm for synchronizing *Doacross* loops using the synchronization instruction set specified in previous section:

Algorithm 2: (for synchronizing a Doacross loop)

I: the index variable for the Doacross loop;
f, g, h: integer functions;
allocate p processors;
for each dependence in the loop **do**
  allocate $p+d$ barrier wires, where d is the dependence distance;
  if wire allocation fails, stop;
  insert set($f(g(I,d))$) immediately after the source;
  insert wait($f(h(I,d))$) and reset($f(h(I,d))$) immediately before the sink;
end for loop;
Where:
  $g(I,d) = (I+d) \bmod (p+d) + 1$
  $h(I,d) = I \bmod (p+d) + 1$

In Algorithm 2, functions g and h map an integer pair, <*loop index, dependence distance*>, to the logical address of a synchronization wire. Function f then maps the logical address of a synchronization wire to its physical location in the synchronization hardware. The logical address ranges from 1 to $p+d$; the physical address ranges from 1 to m, where m is the total number of the synchronization wires in the target system. Note that functions g and h are periodic so their values do not have to

recompute for each iteration of the loop.

The mappings in function f is determined dynamically at the wire allocation time, but the mappings in functions g and h must be provided by the algorithm. However it is possible to delay the processor allocation from compile time by inserting the value of p dynamically at run time. As a result, processor allocation can be more flexible.

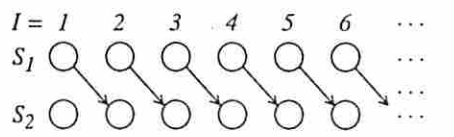
For a *Doacross* loop, the data dependence is either *lexical-forward* or *lexical-backward*. In lexical forward dependences the source of the dependence lexically precedes the sink, whereas in lexical backward dependences the sink of the dependence lexically precedes the source. These two cases are discussed below separately:

C. Doacross Loops with Forward Dependences

Figure 8(a) shows a *do* loop with a lexical-forward dependence from statement S_1 to statement S_2. The dependence distance is one, and the data dependence graph is illustrated in Fig. 8(b). In this

do $I = 1,\ N$
S_1 : $A(I) = B(I) + C(I)$
S_2 : $D(I) = A(I-1) + E(I)$
end do

(a) Original program



(b) Forward dependence

Doacross $I = 1,\ N$
S_1 : $A(I) = B(I) + C(I)$
set($f((I+1) \bmod 5 + 1)$)
wait($f(I \bmod 5 + 1)$)
reset($f(I \bmod 5 + 1)$)
S_2 : $D(I) = A(I-1) + E(I)$
end Doacross

(c) Annotated program

Processor	P_1	P_2	P_3	P_4
Time	S_1^1	S_1^2	S_1^3	S_1^4
	set(f(3))	set(f(4))	set(f(5))	set(f(1))
⇓	wait(f(2))	wait(f(3))	wait(f(4))	wait(f(5))
	reset(f(2))	reset(f(3))	reset(f(4))	reset(f(5))
	S_2^1	S_2^2	S_2^3	S_2^4
	S_1^5	S_1^6	S_1^7	S_1^8
	set(f(2))	set(f(3))	set(f(4))	set(f(5))
	wait(f(1))	wait(f(2))	wait(f(3))	wait(f(4))
	reset(f(1))	reset(f(2))	reset(f(3))	reset(f(4))
	S_2^5	S_2^6	S_2^7	S_2^8
	⋮	⋮	⋮	⋮

(d) Execution profile

Figure 8: Synchronizing a lexical-forward-dependence *Doacross* loop.

example, statement instance S_2^i in iteration i must wait for statement instance S_1^{i-1} in iteration $i-1$ to execute. This waiting between statement instances is enforced by inserting three instructions—*set*, *wait*, and *reset*—between the source and the sink of the dependence. Assume the system allocates four processors to this program. Then five synchronization wires are demanded. Note that function

$g(I,1)$ becomes $(I+1)$ mod $5+1$, and function $h(I,1)$ becomes I mod $5+1$. The resulting annotated program is shown in Fig. 8(c).

Figure 8(d) illustrates the execution profile of the program on four processors. It can be seen that five synchronization wires are shared among different iterations. For example, processor P_1 will periodically use 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 $f(3)$. The indices of function f only need to be computed once.

D. Doacross Loops with Backward Dependences

Figure 9(a) shows a *do* loop with a lexical-backward dependence from statement S_2 to statement S_1. The dependence distance is two, and the data dependence graph is illustrated in Fig. 9(b). In

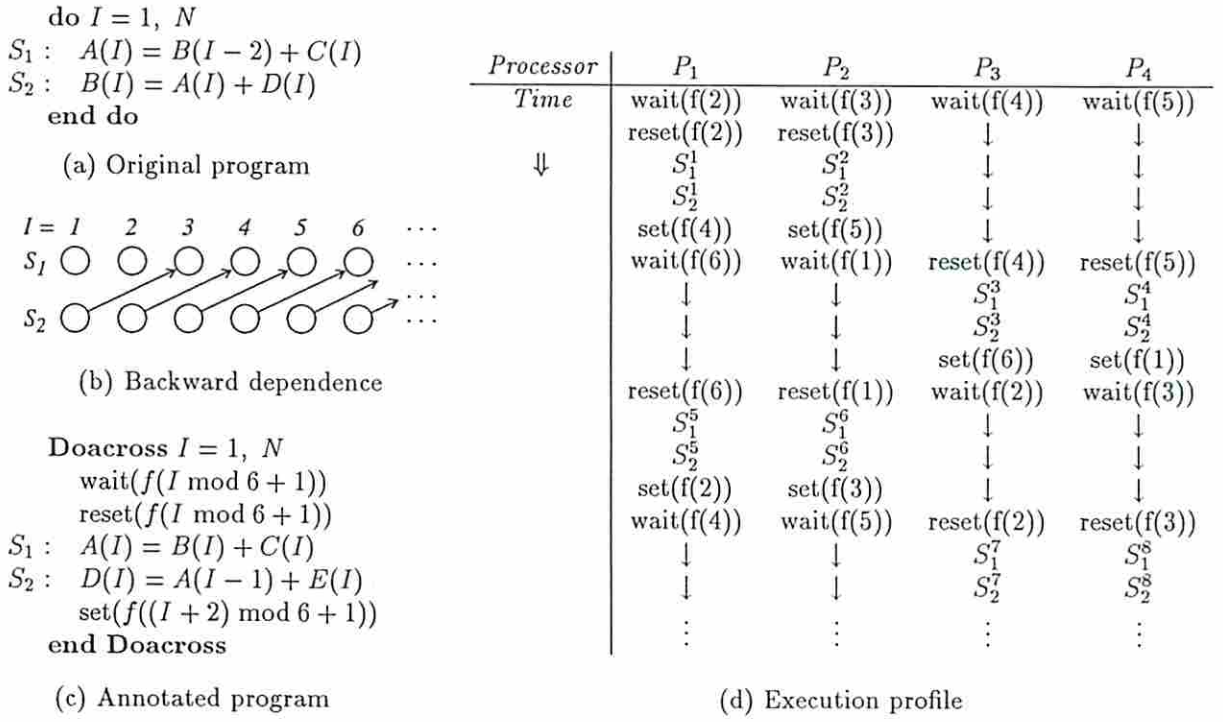


Figure 9: Synchronizing a lexical-backward-dependence *Doacross* loop.

this example, statement instance S_1^i in iteration i must wait for statement instance S_2^{i-2} to execute. Similar to the previous example, this waiting between statement instances is enforced by inserting three synchronization instructions in the loop body. Assume the system also allocates four processors to this program. Then six synchronization wires are demanded. Note that function $g(I,2)$ becomes $(I+2)$ mod $6+1$, and function $h(I,2)$ becomes I mod $6+1$. The resulting annotated program is shown in Fig. 9(c).

Figure 9(d) illustrates the execution profile of the program on four processors. It can be seen

that six synchronization wires are shared among different iterations. Processors P_1 and P_3 share 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 only need to be computed once. Compared to the lexical-forward dependence loop in the previous example, some loop iterations in this example are delayed for execution. This is due to the data dependence constraint within the loop body, not constrained by the underlying hardware.

9 Performance of Doacross Loops

This section analyzes the effect of synchronization overheads to the execution times of *Doacross* loops. *Doacross* loops are the focus because they usually require frequent data synchronization and thus demand more attention than *Doall* loops. It is also important to identify the circumstances under which parallelizing *Doacross* loops is most beneficial.

To simplify the analyses, suppose p processors are executing an M-iteration *Doacross* loop containing only two statements. The data dependence between the statements is either lexical-forward or lexical-backward. Assume the execution times of the two statements are equal and the time for accessing the loop index is equally amortized to the execution times of the two statements. Moreover, the number of the synchronization vectors is assumed sufficient to satisfy synchronization demands, since how fast *Doacross* loops can be executed is the focus. The time for each operation performed by the synchronization instructions is equal to the time for a processor to access its local memory. The *wait* instruction may require multiple operations before it is finished. Summarized in Table 2 are basic parameters and assumptions made.

Table 2: Parameters and Notations for Performance Analysis of Doacross Loops.

Parameter	*Notation*
The number of available processors	p
The number of iterations in the loop	M
Processor cycle	τ
Loop-body statements	S_1, S_2
Type of data dependence	lexical-forward or lexical-backward
Dependence distance	d and $d \leq p$
Granularity of loop bodies S_1, S_2	t/τ
Time for loop index access	amortized over t
Synchronization overhead in using memory-based barriers	$c = k\tau$ for some integer k

Let T_s denote the execution time of a loop being executed sequentially and T_p the execution time of a loop being executed in parallel. The speedup, S_p, is defined to be the ratio of T_s to T_p.

The efficiency, E, of a system is defined as the ratio of S_p to p. Evaluated below separately are the performances of loops with constant, lexical-forward dependence distances and loops with constant, lexical-backward dependence distances.

If the lexical-forward dependence *Doacross* loop is executed sequentially by a processor, an iteration is finished within $2t$ time. Altogether, it takes $2Mt$ to finish executing the loop. On the other hand, if the loop is executed in parallel by p processors, p loop iterations can be finished within $2t + 3\tau$ time, of which $2t$ is spent for executing the two statements and 3τ for executing the three inserted synchronization instructions. It takes totally $\lceil \frac{M}{p} \rceil \cdot (2t + 3\tau)$ time to execute the loop in parallel.

The time to execute the lexical-backward dependence *Doacross* loop sequentially is the same as that to execute the lexical-forward dependence loop. If the loop is executed in parallel, only d loop iterations can be executed at the same time, and it takes $2t + 3\tau$ time to finish an iteration. Therefore, it takes totally $\lceil \frac{M}{d} \rceil \cdot (2t + 3\tau)$ time to execute the loop in parallel. Similarly from T_s and T_p, S_p and E can be derived. The results are summarized as follows.

$$\begin{aligned} T_s &= 2Mt \\ T_p &= \lceil \tfrac{M}{d} \rceil (2t + 3\tau) \\ S_p &= \tfrac{T_s}{T_p} \approx \tfrac{2d}{2+\frac{3\tau}{t}} \\ E &= \tfrac{S_p}{p} \approx \tfrac{\frac{2d}{p}}{2+\frac{3\tau}{t}} \end{aligned} \tag{2}$$

For $S_p > 1$, we have $\frac{2d}{2+\frac{3\tau}{t}} > 1$, or $\tau < \frac{2}{3}(d-1)t$. Since τ must be greater than 0, d must be greater than one. That means it is feasible to execute in parallel a lexical-backward dependence *Doacross* loop only when its dependence distance is more than one. Intuitively no more parallelism can be exploited from a lexical-backward dependence loop with a dependence distance of one because the loop iterations must be executed sequentially. This intuition coincides with our derivation above. Also, we have $\frac{1}{p} < E < \frac{d}{p}$. Clearly it is not feasible to use many more than d processors to speedup the execution in this case.

E. Memory-Based Barriers

The memory-based barrier scheme [17, 18] 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: *test* and *testset* for this purpose. The *test* instruction takes two arguments: a variable to be tested and a dependence distance; it waits until the dependence is resolved. The *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.

Figure 10 shows the execution sequences on four processors for the previous lexical-forward-dependence *Doacross* loop example using the memory-based barriers. Note the exact timings are not specified in the execution sequences.

Processor	P_1	P_2	P_3	P_4
Execution	S_1^1	S_1^2	S_1^3	S_1^4
Sequence	testset(R)	testset(R)	testset(R)	testset(R)
	S_2^1	↓	↓	↓
⇓	S_1^5	S_2^2	↓	↓
	testset(R)	S_1^6	S_2^3	↓
	↓	testset(R)	S_1^7	S_2^4
	S_2^5	↓	testset(R)	S_1^8
	S_1^9	S_2^6	↓	testset(R)
	testset(R)	S_1^{10}	S_2^7	↓
	↓	testset(R)	S_1^{11}	S_2^8
	⋮	⋮	⋮	⋮

Figure 10: Execution sequences for synchronizing a lexical-forward-dependence *Doacross* loop with a dependence distance of one on four processors.

We want to estimate the time to execute a lexical-forward-dependence *Doacross* loop. Let T_p' denote the execution time of a loop being executed in parallel using the memory-based barriers. Assume each operation by the *test* instruction takes τ_1 processor cycles and each operation by the *testset* instruction takes τ_2 processor cycles. Note that the *test* and the *testset* instructions may take multiple τ_1 and multiple τ_2 cycles, respectively. Usually, it takes more time to update a variable than to read a variable, i.e. $\tau_2 \geq \tau_1$.

The execution of the *testset* instruction requires further arbitration if there are multiple updates being performed at the same time. As a result, τ_2 should be one to several orders greater than τ, depending on how large is the system and how the arbitration is implemented. The value of τ_2 increases rapidly, as the number of processors increases.

To estimate T_p' for the loop being executed in parallel, two conditions are considered: (1) the execution of the *testset* instruction fully overlaps with the execution of statements S_1 and S_2, or (2) the execution of the *testset* instruction does not fully overlap. The first condition occurs when one processor starts executing the *testset* instruction and all the other processors finish executing the same instruction. Thus no delay is incurred. It can be seen that t must be greater than or equal to $\frac{p-1}{2}\tau_2$. The second condition occurs when one processor starts executing the *testset* instruction and

some other processors have not finished executing the same instruction. Thus some delay is incurred for updating the variable sequentially. It can be seen that t must be less than $\frac{p-1}{2}\tau_2$.

Based on these two conditions, T'_p is estimated as:

$$T'_p \approx \begin{cases} (2t+\tau_2)\cdot\frac{M}{p}+(p-1)\cdot\tau_2 & \text{if } t \geq \frac{p-1}{2}\tau_2 \\ (M+p-1)\cdot\tau_2 & \text{if } t < \frac{p-1}{2}\tau_2 \end{cases} \quad (3)$$

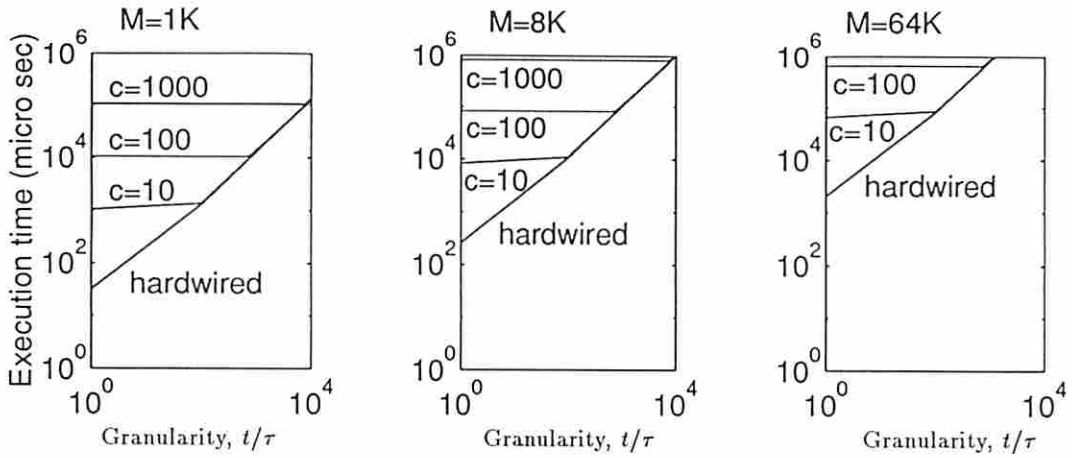
Note my proposed scheme requires a time of $\lceil\frac{M}{p}\rceil(2t+3\tau)$ to execute the loop. If the loop is executed only by a small number of processors, the execution time of using hardwired barriers is comparable to that of using memory-based barriers. On the other hand, if the number of executing processors is large, t will be less than $\frac{p-1}{2}\tau_2$. Compared to the hardwired barrier, the memory-based barrier scheme should require more time to execute the loop in parallel.

10 Comparison with Memory-Based Barriers

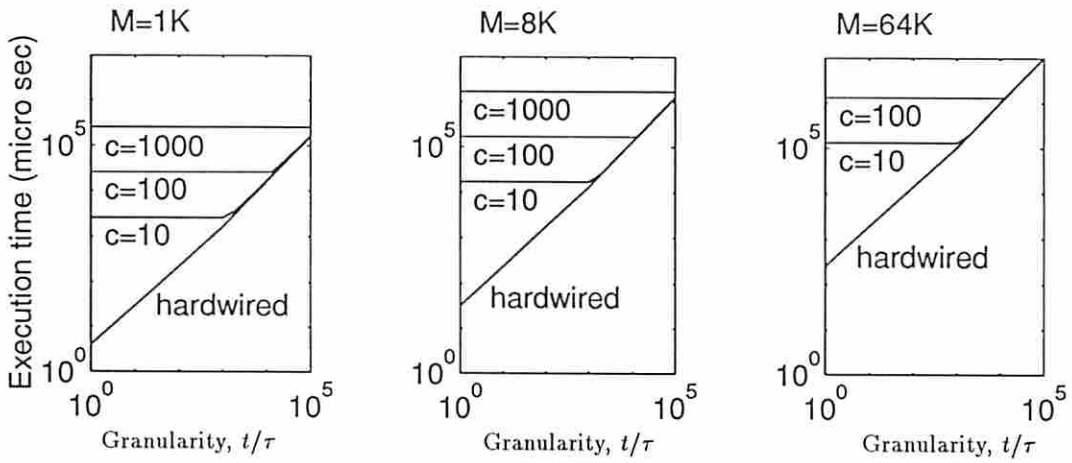
Let $c = \tau_2/\tau$ be the *synchronization overhead* in using memory-based barriers. Typically, this overhead is O(10) to O(10^3) of the processor cycle (τ). Figure 11 illustrates the times to execute lexical-forward-dependence loops using the hardwired barrier scheme or the memory-based barrier scheme. The number of the loop iterations is set to 1K, 8K, or 64K; the number of processors in the system is set to 16, 256, or 4,096. As the *granularity* of the loop body, t/τ, increases, the memory-based barrier scheme performs closer to the hardwired scheme. Unfortunately to be close, the ratio t/τ becomes a very large number and it increases rapidly as the machine size increases. For example, for a 4,096-processor system, the ratio ranges from 10^4 to 10^6, if the overhead c ranges from 10 to 1,000. This means it is not feasible to use the memory-based barrier scheme to explore fine-grain parallelism due to its long latency.

In the worst case when t/τ approaches 1 in the 16-processor system, using the memory-based barrier scheme takes about 30 times more time than using the hardwired scheme for the overhead $c = 10$, 300 times for $c = 100$, or 3000 times for $c = 1,000$. When t/τ approaches 1 in the 256-processor system, using the memory-based barriers takes about 500 times more time than using the hardwired scheme for $c = 10$, 5,000 times for $c = 100$, or 50,000 times for $c = 1,000$. When t/τ approaches 1 in the 4,096-processor system, using the memory-based barrier scheme takes about 10^4 times more time than using the hardwired scheme for $c = 10$, 10^5 times for $c = 100$, or 10^6 times for $c = 1,000$. This clearly proves the superiority of the hardwired barrier scheme over the memory-based barrier scheme.

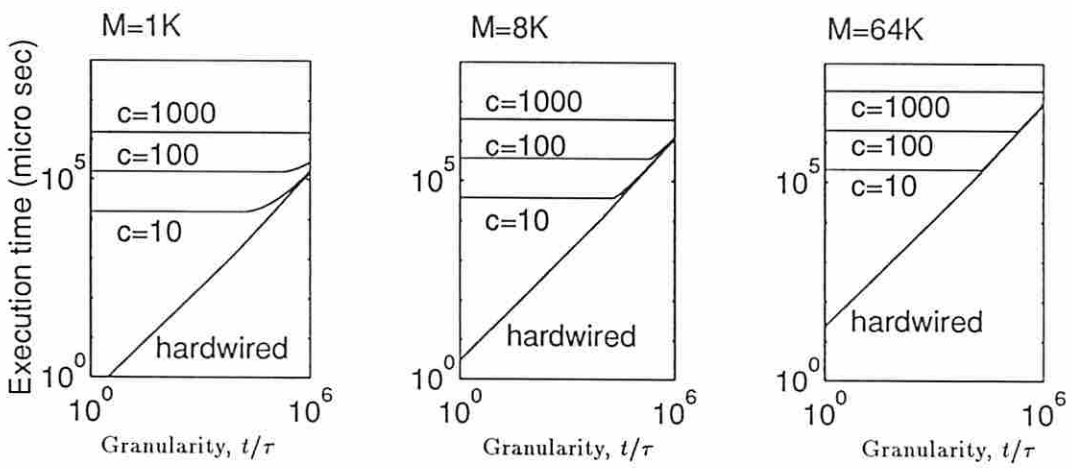
Figure 12 shows the execution sequences on four processors for the previous backward-dependence



(a) 16-processor systems



(b) 256-processor systems



(c) 4,096-processor systems

Figure 11: Timing analysis of Doacross loops with forward dependence using hardwired barriers compared with using memory-based barriers. (c: synchronization overhead)

Doacross loop example using the memory-based barrier scheme. Again, no timing information is given or implied. A substantial amount of delay is incurred with updating the shared variable and waiting

Processor	P_1	P_2	P_3	P_4
Execution	test(R, 2)	test(R, 2)	test(R, 2)	test(R, 2)
Sequence	S_1^1	S_1^2	↓	↓
	S_2^1	S_2^2	↓	↓
⇓	testset(R)	testset(R)	↓	↓
	test(R, 2)	↓	S_1^3	↓
	↓	test(R, 2)	S_2^3	S_1^4
	↓	↓	testset(R)	S_2^4
	S_1^5	↓	test(R, 2)	testset(R)
	S_2^5	S_1^6	↓	test(R, 2)
	testset(R)	S_2^6	↓	↓
	test(R, 2)	testset(R)	S_1^7	↓
	↓	test(R, 2)	S_2^7	S_1^8
	↓	↓	testset(R)	S_2^8
	S_1^9	↓	test(R, 2)	testset(R)
	⋮	⋮	⋮	⋮

Figure 12: Execution sequences for synchronizing a lexical-backward-dependence *Doacross* loop with a dependence distance of two on four processors.

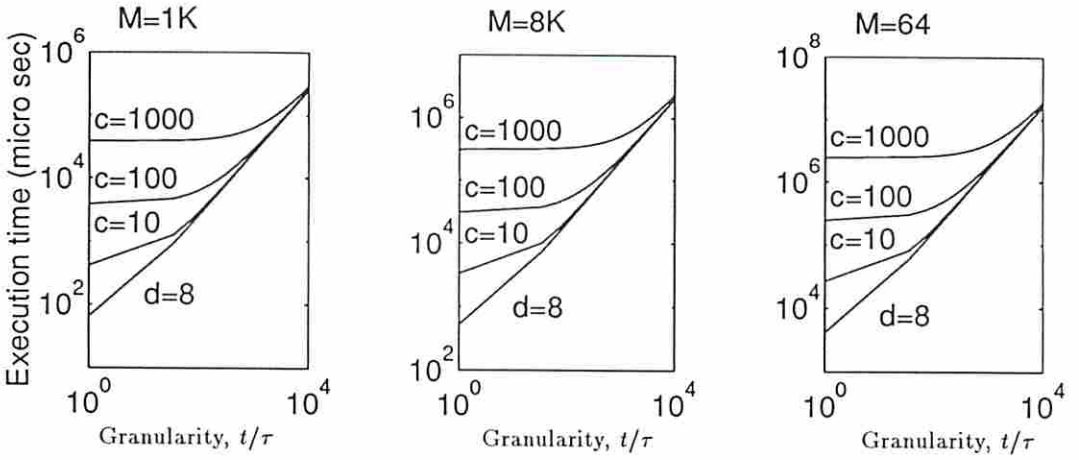
for dependence resolved. The time for executing the loop in parallel is estimated as:

$$T_p' \approx \lceil \frac{M}{d} \rceil \cdot [2t + (d+1)\tau_2] \tag{4}$$

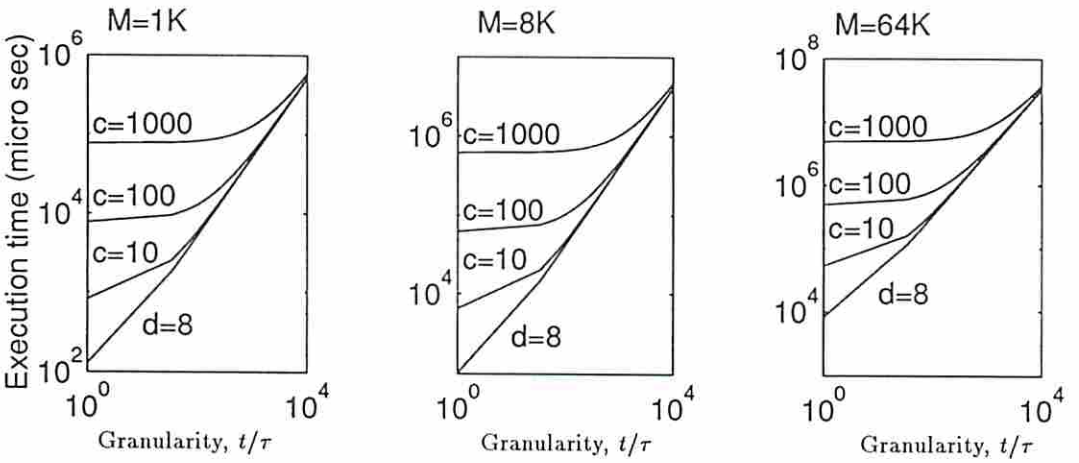
Compared to the proposed scheme which demands $\lceil \frac{M}{d} \rceil (2t + 3\tau)$ time in this case, the memory-based barrier scheme requires more time to finish executing the loop.

Figure 13 shows the times to execute lexical-backward-dependence loops using the hardwired barrier scheme or the memory-based barrier scheme. The number of the loop iterations is set to 1K, 8K, or 64K; the number of processors in the system is set to 16, 256, or 4,096. As the granularity t/τ increases, the memory-based barrier scheme performs closer to the hardwired scheme. In the finest case when t/τ approaches 1, using the memory-based barrier scheme takes 6, 60, and 600 times longer execution time than using the hardwired scheme with respect to memory overhead c = 10, 100, and 1,000 cycles, respectively.

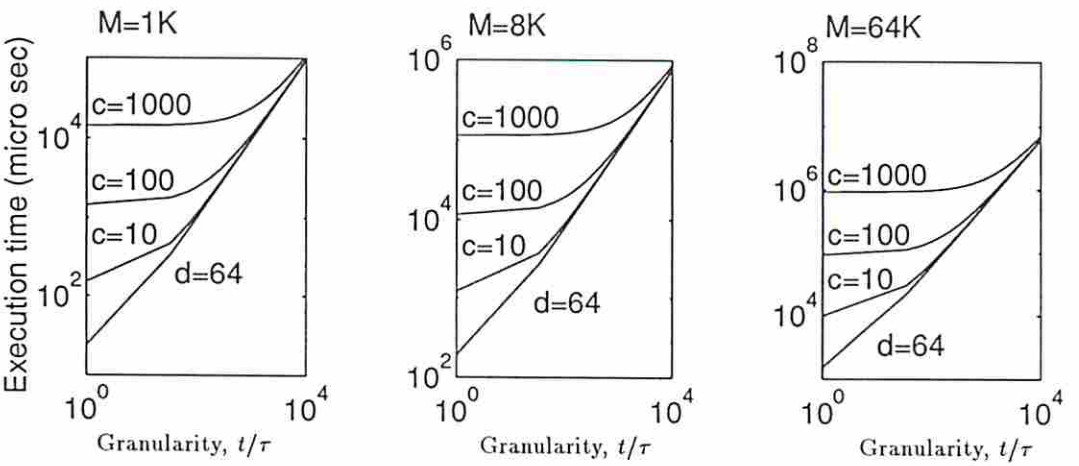
The time difference between the hardwired scheme and memory-based barrier scheme for executing backward-dependence loops is less than that for executing forward-dependence loops, due to limited parallelism in backward-dependence loops. Even so, this still proves the superiority of the hardwired scheme over the memory-based barrier scheme. Table 3 summarizes the estimated speedups of using the hardwired scheme over using the memory-based barrier scheme to execute fine-grain *Doall* and *Doacross* loops, when the granularity t/τ approaches 1.



(a) 16-processor systems



(b) 256-processor systems



(c) 4,096-processor systems

Figure 13: Timing analysis of loops with backward dependence using hardwired barriers compared with using memory-based barriers. (c: synchronization overhead, d: dependence distance)

Table 3: Estimated Speedup of Hardwired Barriers over Memory-Based Barriers.

Loop Type	*Processor No. p*	*Synchronization Overhead, c*		
		10	100	1,000
Doall Loops	16	10^2	10^3	10^4
	256	10^3	10^4	10^5
	4,096	10^4	10^5	10^6
Doacross Loops	16	6	60	600
	256	6	60	600
	4,096	6	60	600

11 Conclusions

Towards scalable multiprocessing, we have proposed a distributed barrier architecture for fast synchronization. The hardwired barriers are demonstrated by parallel execution of *Doall* and *Doacross* loops. The wired-NOR mechanism is shown effective to implement partially-ordered barriers as well as fuzzy barriers. The mechanism requires no network transactions, while processes are waiting for the barrier wires to signal completion.

The proposed hardwired barriers are expanded with hierarchical buses. This introduces only a logarithmic growth of the incurred latency. Based on today's electronic and bus packaging technologies, the hardwired barrier synchronization can be done as fast as in 300 ns for a shared memory system with 4,096 processors. To apply the hardwired barriers, we have developed a complete set of synchronization primitives. Beyond parallel loop execution, the hardwired barriers can be applied for general purpose, MIMD synchronization in large-scale multiprocessors.

The proposed hardwired scheme is especially attractive to implement fine-grain, MIMD parallelism in future massively parallel processors (MPP). In fine-grain MPPs, the synchronization latency must be maintained as low as possible. With a few processor cycles, the hardwired barriers are ideal for building such large-scale systems. The increased cost of adding the hardwired synchronization buses is rather limited, because only limited barrier wires are needed. The lower-level wires can be implemented on backplane buses. The higher-level wires can be implemented with flat or coaxial cables interconnecting the multiprocessor clusters. It would be meaningful to see these claimed advantages be verified through prototyping experiments by MPP manufacturers in industry.

References

[1] T.E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessor. *IEEE Transactions on Parallel and Distributed System*, 1(1):6–16, Jan 1990.

[2] N.S. Arenstorf and H.F. Jordan. Comparing Barrier Algorithms. *Parallel Computing*, 12:157–170, 1989.

[3] T.S. Axelrod. Effects of Synchronization Barriers on Multiprocessor Performance. *Parallel Computing*, 3:129–140, 1986.

[4] B. Beck, B. Kasten, and S. Thakkar. VLSI Assist For a Multiprocessor. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–20, Oct 1987.

[5] C.J. Beckmann and C.D. Polychronopoulos. Fast Barrier Synchronization Hardware. In *Proceedings of IEEE Supercomputing*, pages 180–189, Nov 1990.

[6] R. Cytron. Doacross: Beyond Vectorization for Multiprocessors. In *Proceedings of International Conference on Parallel Processing*, pages 836–844, 1986.

[7] H.G. Dietz. Finding Large-Grain Parallelism in Loops with Serial Control Dependencies. In *Proceedings of International Conference on Parallel Processing*, pages 114–121, 1988.

[8] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Apr 1989.

[9] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *Computer*, 23(6):60–69, Jun 1990.

[10] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, Apr 1989.

[11] T. Hoshino. PAX Computer. in *High-Speed Parallel Processing and Scientific Computing*, edited by H.S. Stone, Addison Wesley, Reading, MA, 1989.

[12] K. Hwang and S. Shang. Wired-NOR Barrier Synchronization for Designing Large Shared-Memory Multiprocessor. In *Proceedings of International Conference on Parallel Processing*, St. Charles, IL, pages I171–I175, Aug 13-15 1991.

[13] C.P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. In *Proceedings of International Conference on Parallel Processing*, pages 236–240, 1984.

[14] J. Lee and U. Ramachandran. Synchronization With Multiprocessor Caches. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 27–37, May 1990.

[15] S.F. Lundstrom. Applications Considerations in the System Design of Highly Concurrent Multiprocessors. *IEEE Transactions on Computers*, C-36(11):1292–1309, Nov 1987.

[16] J.M. Mellor-Crummey and M.L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb 1991.

[17] S. Midkiff and D. Padua. Compiler Generated Synchronization for Do Loops. In *Proceedings of International Conference on Parallel Processing*, pages 544–551, 1986.

[18] S.P. Midkiff and D.A. Padua. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, Dec 1987.

[19] M.T. O'Keefe and H.G. Dietz. Hardware Barrier Synchronization: Dynamic Barrier MIMD (DBM). In *Proceedings of International Conference on Parallel Processing*, pages I43–I46, 1990.

[20] M.T. O'Keefe and H.G. Dietz. Hardware Barrier Synchronization: Static Barrier MIMD (SBM). In *Proceedings of International Conference on Parallel Processing*, pages I35–I42, 1990.

[21] H.B. Ribas. Obtaining Dependence Vectors for Nested-Loop Computations. In *Proceedings of International Conference on Parallel Processing*, pages 212–219, 1990.

[22] Sequent Computer Systems, Inc., Beaverton, Oregon. *Symmetry Technical Summary (Rev. 1.1)*, Dec 1987.

[23] S. Shang. *Fast Barrier Synchronization for Shared-Memory Multiprocessors.* Ph.D. Thesis, University of Southern California, Los Angeles, California, 1993.

[24] W. Shang, M.T. O'Keefe, and J.A.B. Fortes. On Loop Transformations for Generalized Cycle Shrinking. In *Proceedings of International Conference on Parallel Processing*, pages II 132–141, 1991.

[25] H.M. Su and P.C. Yew. On Data Synchronization for Multiprocessors. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 416–423, 1989.

[26] P. Tang and P. Yew. Processor Self-Scheduling for Multiple-Nested Parallel Loops. In *Proceedings of International Conference on Parallel Processing*, pages 528–535, 1986.

[27] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, 1989.

[28] P.C. Yew, N.F. Tzeng, and D.H. Lawrie. Distributing Hot-Spot Addressing in Large-scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, Apr 1987.

[29] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–225, May 1990.