Computer Sciences Department

Synchronization Hardware for Networks of Workstations: Performance vs. Cost

Rahmat Hyder
David Wood

Technical Report #1289

October 1995

UNIVERSITY OF
WISCONSIN
MADISON

1289

Synchronization Hardware for Networks of Workstations: Performance vs. Cost

Rahmat S. Hyder and David A. Wood

Computer Sciences Department
University of Wisconsin—Madison
1210 West Dayton Street
Madison, WI 53706
wwt@cs.wisc.edu

Abstract

Networks of workstations (NOWs) are gaining popularity as lower-cost alternatives to massively-parallel processors (MPPs) because of their ability to leverage high-performance commodity workstations and data networks. However, fast data networks will not suffice if applications require frequent global synchronization, e.g., barriers, reductions, and broadcasts. Many MPPs provide hardware support specifically to accelerate these operations. Separate synchronization networks have also been proposed for NOWs, but such add-on hardware only makes sense if the performance improvement is commensurate with its cost. In this study, we examine the cost/performance tradeoff of add-on synchronization hardware for an emulated 32-node NOW, running an aggregate workload of nine shared-memory, message-passing, and data-parallel applications. For low-latency messaging (e.g., ~10 μs), add-on hardware is cost-effective only if its per-node cost is less than 8% of the base workstation cost. For higher-latency messages (e.g., ~100 μs), add-on hardware is cost-effective if it costs less than 23% of the base cost. Of course, individual applications behave differently: four of the nine applications show no benefit from the extra hardware, while one application improves by over a factor of three with higher-latency messages.

Keywords: synchronization, networks of workstations, massively-parallel processors, cost/performance.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF PYI Award CCR-9157366, NSF Grant MIP-9225097, and donations from A.T.&T. Bell Laboratories, Digital Equipment Corporation, Sun Microsystems, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

1 Introduction

Networks of workstations (NOWs) are gaining popularity as lower-cost alternatives to the current generation of massively-parallel processors (MPPs). NOWs exploit commodity *systems*—entire workstations—rather than individual components. In addition, NOWs leverage commodity network technology to further reduce engineering cost. While current local-area network performance is poor by MPP standards (latencies in the 100s to 1000s of microseconds and bandwidths in the 10s to 100s of Mbits/second), emerging networks promise better performance. For example, repackaged multicomputer interconnects, such as Myrinet [4] and Shrimp [3], may yield up to two orders-of-magnitude performance improvement over previous-generation local-area networks.

However, high-performance CPUs and data networks will not be sufficient for NOWs to achieve good speedups for all existing parallel applications. Some applications require frequent synchronization to coordinate computation (e.g. barriers), to compute global results (e.g. reductions), or update common data structures (e.g. broadcasts). To address the requirements of these applications, many MPPs—the Cray T3D [13], the Fujitsu VPP500 [29], and the Thinking Machines CM-5 [14]—provide explicit hardware support for global synchronization.

NOWs can also employ synchronization hardware in the form of a separate add-on synchronization network. For example, Dietz, et al., have developed barrier hardware which connects to the standard Centronics parallel port of an IBM-compatible PC [9]. Hall and Driscoll have proposed a synchronization network for Sun workstation clusters that suppors barriers, 64-bit reductions, and broadcasts [11]. Shang and Hwang have proposed barrier hardware for cluster-based multiprocessors, including workstation clusters [25].

However, whether or not such add-on synchronization hardware is cost-effective depends upon its cost, the cost of the base NOW, and the performance improvement which the synchronization hardware provides. For example, if the base NOW costs $20,000 per node (including the network) and the add-on synchronization hardware costs $2000 per node, then performance must improve by at least 10% for the synchronization hardware to be cost-effective [32]. While individual applications may improve this much, or more, the add-on hardware is only cost-effective if the aggregate performance of the NOW's entire workload improves by 10%.

In this paper, we examine the cost/performance tradeoff of add-on synchronization hardware for a NOW. We focus specifically on global synchronization (e.g., barriers), not pairwise synchronization (e.g., locks). We study a range of benchmarks to understand which existing applications will benefit—and by how much—from explicit hardware synchronization support. We examine the synchronization requirements for three important classes of applications: shared-memory, message-passing, and data-parallel. We consider two alternative synchronization networks: a low-cost version that supports only simple barriers and single-bit AND operations, called *HW-1* ("Hardware-One"); and a higher-cost version which additionally supports integer reductions and broadcasts, called *HW-All* ("Hardware-All"). For these applications, we calculate the break-even cost—the price at which synchronization hardware becomes cost-effective.

We use a Thinking Machines CM-5 to model a NOW both with and without hardware synchronization support and use the measured performance improvement to calculate the cost/performance break-even point. We study the effects of two different network latencies: a "fast data network", modeled by native CM-5 messages (~10 μs latency), and a "slow data network", modeled by CM-5 messages delayed by 100 μs using a relay-node technique described in Section 2.3. For our emulated 32-node NOW, we find the following results.

- For our aggregate workload—a weighted average of nine shared-memory, message-passing, and data-parallel applications—add-on synchronization hardware for a NOW with a fast data network is only cost-effective if it costs less than 8% of the base system cost, or $1700 per node for a base per-node cost of $20,000. For a NOW with a slow data network, add-on hardware is cost-effective if it costs less than 23% of the base system cost, or $4500 per node.
- Individual applications may benefit much more from synchronization hardware. Among our applications, the HW-1 hardware improves performance by up to 54% on a NOW with the fast data network and up to a factor of 3 with the slow data network. The HW-All hardware yields further improvement, increasing performance by factors of up to 2.5 and 3.6, for the two network latencies. Some applications can be restructured to avoid using global synchronization operations, decreasing the performance benefit. However, for the one code we examined in detail, *water* [26], the barrier version is 58% faster than the (original) locking version even *without* hardware barrier support.
- Other applications cannot be restructured as easily to avoid global synchronization, such as the *Wisconsin Wind Tunnel (WWT)* [21]. While most applications synchronize to ensure that messages have been delivered, *WWT* frequently synchronizes to ensure that *no* messages are in flight. For *WWT*, the HW-1 network improves performance up to 54% on a NOW with the fast data network and up to a factor of 3 with the slow data network; the HW-All network improves performance by up to 60% and a factor of 3.56.

Our results show that global synchronization hardware can be a cost-effective addition to a NOW for some workloads. The benefit is relatively greater for slower data networks, since for each synchronization operation the hardware eliminates $O(\log N)$ messages latencies from the critical path. On the other hand, nearly half our applications received no benefit from synchronization support; additional hardware is not justified to support workloads dominated by these applications. A key advantage of add-on synchronization hardware—as opposed to the integrated synchronization hardware in most MPPs—is that only those people that can benefit from it need to buy it.

The remainder of the paper is organized as follows. Section 2 presents the implementation and performance of the synchronization operations which our applications require. Section 3 presents our benchmark suite and the benchmarks' synchronization requirements. Section 4 presents our experimental performance results. Section 5 presents our model for determining the cost/performance break-even point, estimations of system cost, and the cost/performance break-even points of our applications. Section 6 discusses related work, and Section 7 summarizes our results and conclusions.

2 Implementation of synchronization operations

The applications in this study require a variety of global synchronization operations of differing complexities. The simplest operations include simple barriers and single-bit AND reductions. The more complex operations involve many-to-one ADD reductions which deliver the result to a single node, ADD and MAX reductions which deliver the result to all nodes, and broadcasts. In this section, we discuss the implementation and performance of these synchronization operations, both with and without hardware support. Section 2.1 discusses software implementations, Section 2.2 discusses hardware implementations, and Section 2.3 examines their performance. We

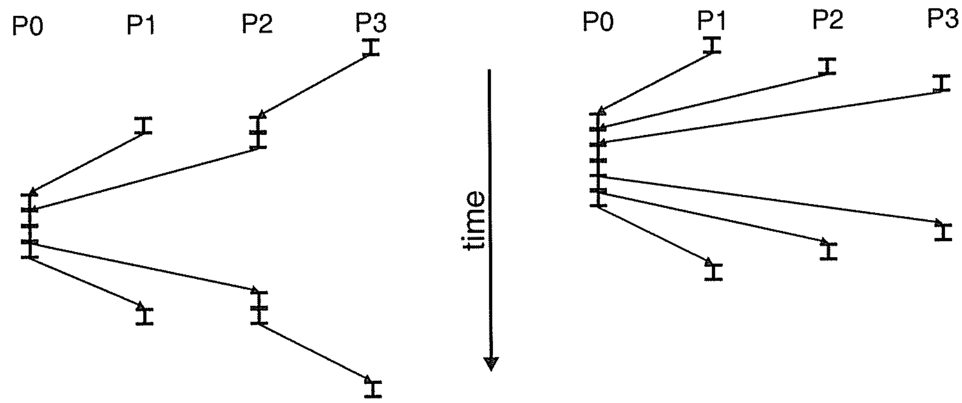

FIGURE 1. Tournament barriers of radix-2 (left) and radix-4 (right) for 4 nodes. The I-bars represent send/receive overhead.

defer the discussion of how our applications use these operations until Section 3.

2.1 Software synchronization

Synchronization operations can be implemented in software by using explicit messages. For example, N processors can perform a barrier synchronization simply by designating one node as a master node, and then indicate barrier *arrival* by sending messages to that master node. Once the master receives arrival messages from all processors, it sends *wakeup* messages to all processors, informing each that all have reached the barrier. Intuitively, this barrier is very inefficient because of the contention at the master. To quantify its speed, let L be the message latency, let S be the overhead of sending a message, and let R be the overhead of receiving a message. Then the barrier latency is $2L + (N+1)R + (N+1)S$. Assuming sufficient network bandwidth, the message latencies can overlap, but the overheads which the master incurs must serialize.

The overheads cannot be eliminated, but their impact can be lessened by distributing them among the processors. Tournament barriers [12, 15] distribute these overheads by having processors perform the arrival phase in pairs ("radix-2" combining), forming a tree. With sufficient network bandwidth, the wakeup phase can also be performed in a tree fashion [15]. Since for every message sent there is exactly one send and one receive overhead latency incurred, the latency of this barrier is then $O(2 \times \lceil log_2 N \rceil \times (L + S + R))$, where the factor of 2 comes from the two separate trees. However, in the presence of significant messaging overhead, radix-2 combining does not hide enough overhead; higher-radix combining is needed. Consider Figure 1, which depicts radix-2 and radix-4 tournament barriers for 4 processors, with significant send and receive overhead. The radix-4 solution hides much more of the messaging overhead, resulting in lower latency: $O(2 \times \lceil log_4 N \rceil \times (L + S + R))$.

Butterfly barriers [7] eliminate the factor of 2 by effectively performing multiple tournament-arrival binary trees in parallel, with each processor at the root of a different arrival tree. Figure 2 illustrates the messaging pattern for radix-2 and radix-4 butterfly barriers for 4 nodes. Once again, the radix-4 barrier allows more send/receive overhead to be hidden than in the radix-2 barrier. The butterfly barrier has the potential for better latency than the tournament barrier, but it sends $O(N\ log_k N)$ messages for a fixed radix k, while the tournament barrier sends only $O(2\ log_k N)$ messages. The butterfly barrier will then outperform the tournament barrier only if there is sufficient network bandwidth. Our emulated NOW satisfies this assumption, so we will concentrate on only the butterfly barrier in the remainder of this paper.

Reductions and broadcasts can also be implemented via data messages as follows. Reductions which require delivering the result to all nodes are essentially simple barriers which also send data, and hence we implement them with a butterfly-style combining pattern. Reductions which deliver the result to a single node and broadcasts are implemented via unbalanced trees [8] in which the fanout of nodes is set according to the latency and overhead of messages.

2.2 Hardware synchronization

Add-on synchronization hardware for a NOW consists of two separate components: the workstation interface and the synchronization network itself. Modern workstations provide a range of possible interfaces, each with different latency, bandwidth, and cost considerations. At the low-end, most workstations provide a parallel port that can be used for low-bandwidth operations such as simple barriers. Parallel ports typically require system calls for user-level access, but in some systems can be memory-mapped directly into user space [9]. Higher performance, at higher cost, can be obtained by interfacing to the workstation's I/

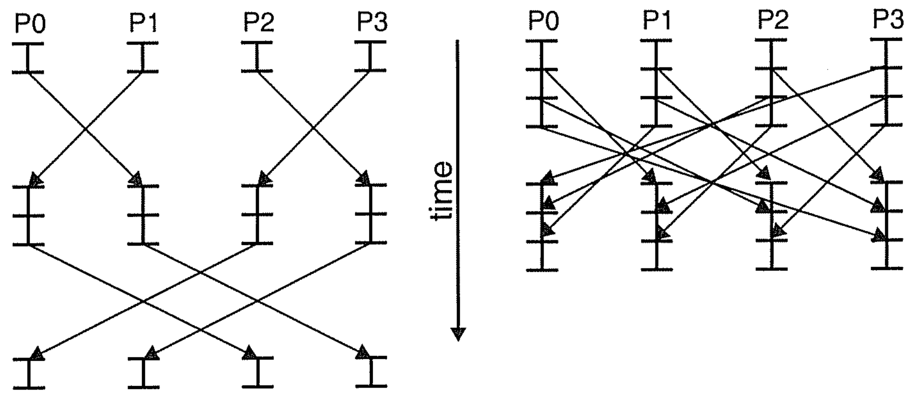

FIGURE 2. Butterfly barriers of radix-2 (left) and radix-4 (right) for 4 nodes. The I-bars represent send/receive overhead.

O bus. This is a better choice for more complex operations, such as broadcasts, that require lower latency and/or higher bandwidth. At the high-end, the synchronization hardware can interface directly to the memory bus in some workstations; however, the increase in performance is unlikely to outweigh the increase in cost. Regardless of which interface location is chosen, the latencies should be relatively low: from a few tens of cycles to a few hundreds of cycles.

The actual combining operation in the switch should be even faster. For example, since a barrier operation is simply a logical AND, it could be implemented as a large AND tree, with delays measured in nanoseconds. More complex operations such as integer reductions require more complex logic, but the basic combining operation will be at most a few processor cycles. Control logic, e.g. for partitioning or virtualizing the network, will introduce additional delays. Nonetheless, overheads will be minimal within the synchronization network.

We consider two possible hardware synchronization networks, each with different complexity and cost. The first is lower-complexity and lower-cost, supporting simple barriers and single-bit AND reductions; we call this network *HW-1* ("Hardware-One"). We include the single-bit AND because some applications make heavy use of this reduction operation, and it requires hardware comparable in complexity to the simple barrier. The second is higher-complexity and higher-cost, supporting the HW-1 features plus the reductions described earlier and broadcasts; we call this network *HW-All* ("Hardware-All"). We assume that the networks connect to the workstation's memory bus, which provides a low-latency, high-bandwidth interface.

The importance of hardware synchronization is magnified by longer network latencies, because software synchronization requires many (slower) messages. To illustrate this point, we express parallel-program runtime as the sum of computation time (T_{work}), communication time (T_{comm}), and synchronization time (T_{global}):

$$T = T_{work} + T_{comm} + T_{global}$$

In the absence of synchronization hardware, T_{comm} and T_{global} increase linearly with message latency, while T_{work} remains constant. As a result, synchronization time becomes a larger portion of the total runtime. Conversely, with synchronization hardware, T_{global} remains constant and becomes a smaller portion of the total runtime. We expect synchronization hardware to be more important for NOWs with long message latencies, especially for fine-grain codes which perform little work and send few messages between synchronization operations.

2.3 Synchronization performance

We now measure the performance of hardware and software synchronization operations on the emulated NOW. We use a Thinking Machines CM-5 to emulate 32-node NOWs with and without hardware synchronization support. To approximate the higher-latency messages of a NOW, we divide a 64-node CM-5 partition into 32 compute nodes and 32 *relay* nodes, using the relay nodes to slow down messages as follows: a message destined for node N is first sent to node $N+32$—a relay node—where it is delayed for the appropriate latency and then sent to node N. Figure 3 illustrates this process for an emulated 4-node NOW with 4 relay nodes. We do not use the compute nodes as relay nodes because doing so would perturb the computation on those nodes. For hardware synchronization, the CM-5 conveniently provides hardware support for all the synchronization operations our benchmarks need.

In addition to the basic 32-node NOW, some of our applications require the existence of a separate *host* node which connects to the 32 processing nodes. The broadcasts

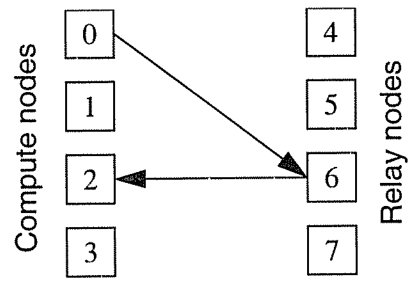

FIGURE 3. 4-node NOW with long-latency messages emulated on an 8-node host machine. Node 0 sends a message to node 2 by sending the message to node 2's relay, which is node 6. The message is delayed at node 6 and then sent to node 2.

Synchronization type	Hardware	Software	
		Message delay	
		0	100 μs
Simple barrier	2.6 μs	60 μs	307 μs
1-bit AND	2.6 μs	67 μs	326 μs
32-bit ADD	2.7 μs	67 μs	326 μs
32-bit ADD (to one node)	3.2 μs	118 μs	283 μs
64-bit MAX	3.5 μs	78 μs	339 μs
Broadcast	2.5 μs	43 μs	230 μs

TABLE 1. Latency of hardware and software synchronization operations. Message delay is the amount by which CM-5 messages (typically 3-5 μs) are delayed. ADD reductions to a single node and broadcasts involve data transmission between a host workstation and a 32-node NOW; all other operations involve only a 32-node NOW. Note that ADD reductions to a single node are significantly slower than broadcasts because sending a message from a node to the host on the CM-5 involves a system call. However, these reductions are very infrequent in our benchmarks, so this is not a concern.

and many-to-one reductions are used in this "extended" NOW: the host broadcasts data to all 32 nodes and receives reduction results from all 32 nodes. The CM-5 directly implements this model with a separate workstation serving as the host.

Table 1 presents the latency, in microseconds, of each type of synchronization operation. These measurements were taken in the context of the software runtime environments for our benchmarks, which are discussed in Section 3. We consider two network latencies: CM-5 base message latency (approximately 3-5 μs), or "zero-delayed messages", and CM-5 messages delayed by 100 μs, or "100 μs-delayed messages". All software operations were optimized with respect to messaging latency and overhead; for example, the simple barriers for 100 μs messages yield the lowest latency when the butterfly pattern uses two phases, combining the nodes in groups of eight nodes in the first phase and then in groups of four nodes in the second phase.

The individual hardware operations are roughly comparable in speed. This result is expected because the bulk of the latency is spent in the network-interface hardware, rather than in the actual combining hardware. The software operations range from one to two orders of magnitude slower than the hardware implementation, depending on the message latency.

3 Synchronization requirements of benchmarks

We consider three classes of applications: shared-memory applications (SM), message-passing applications (MP), and data-parallel applications (DP). Table 2 lists the benchmarks we study in each of the three classes, along

Benchmark	Class	Input	Synchronization frequency (events/sec) Barriers	Integer reductions	Broadcasts
barnes	SM	16,384 mols	0.28	n/a	n/a
dsmc	SM	48,600 mols, 200 iter.	33	n/a	n/a
moldyn	SM	2048 mols, 30 iter.	107	n/a	n/a
water	SM	512 mols, reg. lattice	2324	n/a	n/a
applu	MP	24x24x24, 30 iter	1.01	n/a	n/a
dycore	MP	64x45 grid, 50 iter.	1319	n/a	n/a
WWT	MP	Appbt (8x8x8, 30 iter)	2834	2834	n/a
		Tomcatv (128x128)	2404	2405	n/a
		Sparse (128x128 dense)	2856	2891	n/a
		Water (128 mols, 10 iter)	10221	10227	n/a
metspin	DP	128x128 grid, 250 iter	467	101	2565
nbody	DP	4096 bodies	958	839	10009

TABLE 2. Benchmark suite. Synchronization frequencies are as measured on the CM-5, i.e. when run with zero-delayed messages and hardware support for all synchronization operations.

with their input data sets and frequencies of synchronization on the CM-5.

3.1 Shared-memory benchmarks

The shared-memory benchmarks are *barnes*, *DSMC*, *moldyn*, and *water*. *Barnes* uses the Barnes-Hut algorithm to calculate N-body interactions [26]. To obtain better speedup from *barnes*, we used a modified version which allocates free cells in a processor's local portion of shared memory, obviating the original global free cell pool[1]. *Barnes* invokes barriers after creation and traveral of octrees, and after updates are made to global values. *DSMC* is a rarefied gas simulation, computing interactions between molecules in a 3D box [16]. *DSMC* invokes barriers after new molecules enter the box, after simulating the collision of molecules, and after moving molecules into new space cells. *Moldyn* calculates the motion of atomic particles based on forces acting on each particle from particles within a certain radius [16]. Barriers are invoked after calculating the forces on and velocities of molecules, and after updating the coordinates of molecules. *Water* is a molecular dynamics simulation, computing the interactions among water molecules [26]. For *water*, we use a modified version which is restructured to use barriers instead of locks; this version provides better performance on our platform. Barriers are invoked after updates to molecules.

All of these benchmarks were parallelized by hand using a locally-modified version of the PARMACS macro package and were run on Blizzard, a software system that provides fine-grain distributed shared memory on the Thinking Machines CM-5 [24]. Blizzard uses the user-level Stache protocol—a COMA-like invalidation protocol—to maintain sequentially consistent shared memory [22]. *Barnes* and *water* use Stache to maintain coherence on 128-byte blocks, and *DSMC* and *moldyn* maintain coherence on 1024-byte blocks.

3.2 Message-passing benchmarks

The message-passing benchmarks are *applu*, *dycore*, and the *Wisconsin Wind Tunnel* (*WWT*). *Applu* is a computational fluid dynamics code which solves five coupled parabolic/elliptic partial differential equations [1]. The computation consists of successive over-relaxation iterations with a barrier after each iteration. *Dycore* computes the equations of motion for a grid-point atmospheric global climate model [28]. Barriers are invoked between phases of computation and near-neighbor communication. We ran *applu* and *dycore* on Blizzard, using Blizzard-provided mechanisms to implement message-passing functions for these applications.

Unlike the above applications, *WWT* is a native CM-5 program. *WWT* is a parallel discrete-event simulator which simulates cache-coherent distributed-shared-memory multiprocessors [21]. *WWT* has much more heavy and diverse

1. This modification is similar to the newly-released SPLASH-2 version [30].

Benchmark	Message delay			
	0		100 μs	
	No HW	**HW-1**	**No HW**	**HW-1**
barnes	12.09	11.99	9.65	9.65
DSMC	13.06	13.07	6.92	7.09
moldyn	8.91	9.13	3.50	3.60
water	5.81	6.66	2.87	4.06

TABLE 3. Speedups for shared-memory benchmarks. "No HW" indicates synchronization was done in software. *Barnes's* insignificant slowdown with fast messages results from random system perturbations.

synchronization requirements: up to three types of synchronization may occur after every phase. First, after every *quantum* (100 cycles) of target-program execution, *WWT* must ensure that all messages sent during the current quantum have been received prior to the start of the next quantum. This *Network-Done* operation involves halting further network activity until all messages currently in-flight have been delivered. We can perform Network-Done by either: (i) waiting until the number of messages received by all nodes equals the number sent by all nodes; or (ii) sending an explicit acknowledgement (ACK) for each message and waiting at the end of the quantum for all ACKs to be collected. Solution (i) requires repeated ADD reductions[1], while solution (ii) requires a simple barrier. Second, after every quantum, *WWT* must determine if all nodes encountered a barrier in the target program in the last quantum. This can be accomplished with a single-bit AND reduction. In fact, if option (ii) above is used, the single-bit AND reduction can double as the barrier. Third, if all nodes indicate that they have arrived at a target-program barrier, each target node must set its local clock to the maximum time of all target nodes' local clocks, where the clocks are 64-bit values; this requirement necessitates 64-bit MAX reductions.

3.3 Data-parallel benchmarks

The data-parallel benchmarks are *metspin* and *nbody*, written in CM-Fortran and linked with a modified version of the CM-Fortran communication library. CM-Fortran employs a *host-node* model of computation, where a front-end host machine coordinates computation on a set of parallel nodes by broadcasting parallel functions and data to the nodes, and by receiving results from the nodes. In studying these applications, we assume that the 32-node NOW is connected to an additional workstation which functions as the host.

Metspin uses the Metropolis Monte Carlo algorithm to simulate an Ising spin model of a ferromagnet and calculate the energy and magnetization at a particular temperature [18]. The basic computation is successive over-relaxation, with red-black iterations performed on a 2-D grid. *Nbody* calculates the force between N bodies interacting via long-range forces [10]. Both *metspin* and *nbody* use barriers after cyclic shifts of arrays, broadcasts to distribute both code pointers and data to nodes, and a Network-Done operation to determine the completion of cyclic shifts of arrays. *Metspin* also invokes barriers after phases of near-neighbor computation, and performs many-to-one ADD reductions to the host to track the sum of all cells in the grid whose values have stabilized.

1. The CM-5 network interface provides hardware support to automatically repeat the ADD reduction. However, *WWT* does not employ this feature.

4 Performance results

This section presents the performance for our shared-memory benchmarks (Section 4.1), our message-passing benchmarks (Section 4.2), and our data-parallel benchmarks (Section 4.3) as we vary the synchronization methods and the network latency.

4.1 Shared-memory benchmarks

We ran each of our shared-memory benchmarks on the emulated 32-node NOW with zero-delayed and 100 μs-delayed messages, comparing a system with the HW-1 synchronization network against a system with no synchronization hardware (these benchmarks do not need the added features of HW-All). Table 3 presents the resultant speedups: execution time on a single node divided by the execution time on 32 nodes.

Three of the four benchmarks—*barnes*, *DSMC*, and *moldyn*—perform little global synchronization[2], thus using the HW-1 network does not significantly improve

2. Barnes makes extensive use of locks, but not barriers.

Benchmark	Message delay			
	0		100 μs	
	No HW	HW-1	No HW	HW-1
applu	15.84	15.90	12.98	13.05
dycore	8.94	9.96	5.85	7.93

TABLE 4. Speedups for message-passing benchmarks *applu* and *dycore*

performance regardless of the message latency. On the other hand, *water* synchronizes much more frequently, and thus hardware barrier support improves performance substantially. Performance improves 15% with fast messages and 41% with 100 μs-delayed messages.

In this study we used a restructured version of *water* [26] that uses barriers instead of locks as the primary synchronization operation. Surprisingly, this version runs consistently faster than the (original) locking version, even without hardware barrier support. Specifically, the barrier version is 58% faster than the locking version with software barriers and fast messages, and 77% faster with software barriers and slow messages. With hardware support, the barrier version is 81% faster than the locking version with zero-delayed messages, and 2.5 times faster than the locking version with 100 μs-delayed messages. This result illustrates that restructuring applications to avoid global synchronization may not always be easy.

4.2 Message-passing benchmarks

Table 4 presents the speedups for the first two message-passing benchmarks, *applu* and *dycore*, comparing a system with the HW-1 synchronization network against a system with no synchronization hardware. (These two benchmarks also do not require the added features of HW-All.) For *applu*, its low synchronization frequency implies minimal potential for improvement, regardless of the message latency. For *dycore*, with its higher degree of synchronization, hardware synchronization support yields a 11% improvement with zero-delayed messages and a 36% improvement with 100 μs-delayed messages.

In contrast to the above two applications, *WWT* can utilize either the HW-1 or HW-All synchronization networks. We compare the performance improvement on systems using these networks to a system without synchronization hardware. In the system without synchronization hardware, the Network-Done operation is performed using ACKs[1], and the quantum barrier and maximum-target-barrier-time operations are performed using software reductions. In the system with the HW-1 network, the quantum barrier uses the hardware 1-bit AND reduction, Network-Done is performed using ACKs, and the maximum-target-barrier-time operation is performed with a software reduction. Finally, in the system with the HW-All network, the Network-Done operation uses the 32-bit ADD reduction, the maximum-time operation uses the 64-bit MAX reduction, and the quantum barrier uses the 1-bit AND reduction.

Table 5 presents the speedups for *WWT* for the three systems. The experiments involve *WWT* simulating a 32-node shared-memory multiprocessor, running four shared-memory programs with the *Dir1SW+* protocol [31]. We use four different programs in order to observe how different communication patterns affect *WWT's* performance.

We first examine the HW-1 system. Looking at the performance improvements, we see that with zero-delayed messages, three of the four inputs exhibit modest improvement (no more than 17%), while Water, the fourth, exhibits a 54% improvement. Water has substantially less communication than the other four inputs, so the bulk of the time spent in simulating Water is in the quantum barrier, in contrast to the other three inputs. As a result, performing the quantum barrier in hardware for Water has a greater overall impact than for the other three inputs. With 100 μs-delayed messages, the HW-1 system delivers significantly greater performance: three of the four inputs improve by nearly 50%, and Water improves by a factor of three. The performance of the software reduction used to implement the single-bit AND in the system without synchronization hardware degrades significantly with the higher latency network, and hence the HW-1 system speeds up more by comparison.

We next examine the HW-All system. We first notice that with zero-delayed messages for all inputs except Sparse, the HW-All system is actually slower than the HW-1 system. This slowdown is tied to the method used for performing Network-Done. At a quantum boundary, the HW-All system repeatedly sums the number of messages sent and received across all of the nodes, waiting for this value to reach zero. After that point, it encounters the quantum barrier, at which the target-barrier flags on all nodes are AND'ed. However, the HW-1 system skips the reduction phase, and instead each node waits for all its ACKs to be received before proceeding to the quantum barrier, resulting in only one synchronization operation

1. We found that performing Network-Done with ACKs was faster than performing it with a software reduction.

Message delay	Input	No HW	HW-1	HW-All
0	Appbt	8.55	10.04	9.76
	Tomcatv	10.82	12.60	12.39
	Sparse	7.04	8.07	8.62
	Water	6.96	10.72	10.32
100 μs	Appbt	4.20	6.27	8.12
	Tomcatv	6.00	8.89	10.83
	Sparse	3.81	5.74	7.24
	Water	2.48	7.45	8.83

TABLE 5. Speedups for message-passing benchmark *WWT*

instead of two. The HW-1 system should then outperform the HW-All system if the target application has "low enough" communication. For all inputs except Sparse, the communication is in fact low enough; Appbt and Tomcatv are stencil computations with small data sets, and Water communicates in only 15% of the quanta. Sparse, on the other hand, requires multiple broadcasts and a reduction each iteration. With 100 μs-delayed messages, the HW-All system consistently outperforms the HW-1 system. This is because the cost of sending ACKs increases drastically, and therefore the HW-All system's repeated hardware ADD reductions to determine network quiescence is a better solution.

4.3 Data-parallel benchmarks

Table 6 presents the speedups for the data-parallel applications, comparing systems with the HW-1 and HW-All synchronization networks against a system without synchronization hardware. In the system without synchronization hardware, the barriers, reductions, and broadcasts are implemented using messages, and the Network-Done operation is implemented with ACKs and a message-based barrier, similar to the implementation in *WWT*[1]. In the HW-1-based system, the barriers are performed in hardware, with the Network-Done operation implemented via ACKs and a hardware barrier; all other operations are implemented as in HW-1. Finally, in the HW-All-based system, the barriers, reductions, and broadcasts are all implemented in dedicated hardware, with the Network-Done operation using a hardware reduction.

For the HW-1 system and the system without synchronization hardware, we additionally "throttle" the software broadcasts by forcing the host to delay for some amount of time (200-500 μs, depending on the benchmark and the network latency) after a number of broadcasts, instead of letting the host broadcast data to the nodes *ad infinitum*. Brewer and Kuszmaul found that inserting barriers between successive communication operations, such as cyclic shifts, actually improved performance by up to a factor of three, because a message destined for a processor *A* still working on the first operation could not be delayed by a message sent to *A* from a processor *B* working on the second operation [5]. We encounter similar conflicts between sets of messages in our benchmarks, and throttling yields substantial benefit. We determine the delaying parameters experimentally.

We first examine the HW-1 system. With zero-delayed messages, the hardware synchronization provides only modest benefit: 16% for *metspin* and 4% for *nbody*. This result is expected given the low barrier frequencies of these applications. *Nbody* benefits less than *metspin* because *nbody's* increased volume of broadcasts dominates communication and hence its runtime, decreasing the room for improvement for hardware barriers. With 100 μs-delayed messages, HW-1 yields a 28% improvement for *metspin* and no noticeable improvement for *nbody*. The increased benefit of hardware barriers in *metspin* with 100 μs-delayed messages occurs simply from having slower software barriers.

We now examine the HW-All system. This system yields much more performance improvement— 62% for *metspin*, and 68% for *nbody*. For both applications, the bulk of the improvement relative to HW-1 comes from the hardware broadcasts: when hardware reductions are used as well as hardware broadcasts, performance improves by no more than 1%. Since *nbody* broadcasts more often than *metspin*, we would expect that *nbody* would benefit much more from broadcast hardware than *metspin*. However, *metspin* benefits a great deal from the broadcast hardware because its computation-to-communication ratio is lower, and hence the separate broadcast network prevents broadcast traffic from interfering with data traffic. As evidence of *metspin's* higher degree of data communication, we

1. Again, the ACK-based method was found to be faster than performing software reductions.

Msg delay	Benchmark	No H/W	HW-1	HW-All
0	*Metspin*	4.01	4.67	6.53
	Nbody	6.39	6.68	10.75
100 μs	*Metspin*	3.28	4.23	6.12
	Nbody	4.26	4.26	10.76

TABLE 6. Data-parallel speedups

found that when we throttled broadcasts in the system without synchronization hardware, *metspin's* performance improved by 44%, while *nbody's* performance improved by only 4%.

With 100 μs-delayed messages, HW-All yields an 87% improvement for *metspin* and a factor of 2.5 improvement for *nbody.* These improvements are the opposite of those achieved with zero-delayed messages. In *metspin,* we see much less improvement than we saw with zero-delayed messages; the high degree of data communication limits the potential speedup. Conversely, for *nbody* we see much more improvement than we saw with zero-delayed messages; broadcasts form the bulk of the communication, since data communication is minimal, and thus the speedup improves greatly.

5 Cost/performance

In this section, we examine the cost/performance tradeoffs for global synchronization hardware. Section 5.1 presents a break-even model for cost/performance. Section 2.2 examines the expected cost of a system with synchronization hardware. Section 5.3 presents the resulting cost/performance break-even points for our applications and for the entire workload.

5.1 A cost/performance model

Intuitively, a performance enhancement to a computer system is cost-effective only if the increase in performance exceeds the increase in cost. Wood and Hill [32] recently formalized this intuition by showing that parallel computing is more cost-effective than uniprocessor computing whenever the following inequality holds:

$$speedup(p) > costup(p)$$

where *speedup(p)* is the runtime on one processor divided by the runtime on p processors, and *costup(p)* is the cost of a p-processor system divided by the cost of a 1-processor system. In particular, they show that when memory is a significant fraction of uniprocessor cost, parallel computing can be cost-effective even with very low speedups.

This same intuition also applies to add-on synchronization hardware for a parallel computer. Synchronization hardware will be cost-effective if the performance improvement is greater than the increase in cost:

$$speedup(synchronization\ hardware) > costup(synchronization\ hardware)$$

To make this result concrete, let the cost of a base NOW (without hardware synchronization support) be C_{base} and let the cost of the additional synchronization hardware be C_{synch}; then the cost of a NOW with hardware synchronization support is $C_{base} + C_{synch}$. Let the runtime of a workload W on the base NOW be $T_{base}(W)$ and on the NOW with hardware synchronization support be $T_{synch}(W)$. Then synchronization hardware is more cost-effective whenever:

$$\frac{T_{base}(W)}{T_{synch}(W)} > \frac{C_{base} + C_{synch}}{C_{base}} \tag{1}$$

$T_{base}(W)$ and $T_{synch}(W)$ assume a fixed workload W that could either be a single application or the weighted mean of the runtimes of many different jobs.

To make our results independent of any particular hardware implementation, we calculate the break-even cost [19] for add-on synchronization hardware by making Equation 1 an equality and solving for C_{synch}:

$$C^{*}_{synch} = C_{base}\left(\frac{T_{base}(W)}{T_{synch}(W)} - 1\right) \tag{2}$$

Thus synchronization hardware is cost-effective (for a particular workload W) if its actual cost is less than the break-even cost C^{*}_{synch}.

5.2 System cost

Without loss of generality, we make C^{*}_{synch} more concrete by choosing reasonable estimates of the base system cost, C_{base}, and synchronization hardware, C_{synch}. In this section and throughout the rest of the paper, we treat C_{base}, C_{synch}, and C^{*}_{synch} as per-node costs, with the cost of shared resources (e.g. network routers) amortized over all nodes.

C_{base} depends heavily on the particular choice of workstation node. List prices can range from a few thou-

Application	0 delay		100 μs delay	
	HW-1	HW-All	HW-1	HW-All
barnes	< $100	< $100	< $100	< $100
DSMC	< $100	< $100	$482	$482
moldyn	$484	$484	$613	$613
water	$2908	$2908	> $5000	> $5000
applu	< $100	< $100	$101	$101
dycore	$2287	$2287	> $5000	> $5000
WWT	> $5000	> $5000	> $5000	> $5000
metspin	$3279	> $5000	> $5000	> $5000
nbody	$899	> $5000	< $100	> $5000
AGGREGATE	$1694	$4104	$4515	> $5000

TABLE 7. Break-even cost of synchronization hardware for all applications and for the aggregate workload, for zero-delayed messages and 100 μs-delayed messages

sand dollars for an IBM-compatible PC to many tens of thousands of dollars per node for a high-end workstation, depending on the processor speed, memory capacity, and I/O configuration. Similarly, fast networks range from approximately $2000 per node for Myrinet (adapter plus switches) to over $5000 per node for Fore's ATM network. For the purposes of this paper, we assume C_{base} is $20,000, which is roughly the per-node cost of the Wisconsin COW, our local network of dual-processor SPARCstation 20s.

C_{synch} depends on the cost of the synchronization-network switch and workstation interface. For hardware that only supports simple barriers, the switch cost is a minor component, since it can be implemented using a few standard PALs. For example, Dietz, et al., estimate that the PAPERS add-on hardware—which interfaces to the workstation's parallel port and implements simple barriers and binary reductions—has a parts cost of less than $50 [9]. We assume a commercial implementation of PAPERS would cost at least $100 per node. More complex functionality—for example, integer or floating-point reductions—requires more expensive hardware such as FPGAs or ASICs. Interfacing to the workstation's I/O or memory bus can also be expensive, particularly for low-volume parts like synchronization hardware. Nonetheless, it seems reasonable that the total per-node cost for synchronization hardware should not exceed the cost of high-end network hardware. Thus we assume that synchronization hardware will always cost less than $5000 per node.

Synchronization hardware is cost-effective when the actual cost C_{synch} is less than the break-even cost C^*_{synch}. Thus given our estimated costs, add-on synchronization hardware is always cost-effective when C^*_{synch} is greater than $5000 and is never cost-effective when C^*_{synch} is less than $100.

5.3 Results

From the performance results from Section 4, we calculate the weighted means of T_{base} and T_{synch} for our entire combined workload, weighting each application by its fraction of the cumulative runtime of all applications in the workload. For the WWT application, we used the average runtime of all four inputs.

Table 7 presents the values of C^*_{synch} for the individual applications and the aggregate workload. For clarity, values less than $100 are indicated by <$100 (never cost-effective); values greater than $5000 are indicated by >$5000 (always cost-effective). For our aggregate workload on a system with zero-delayed messages, we find that synchronization hardware is cost-effective when the hardware costs less than 8% of the system cost, or $1700 given our cost assumptions. On a system with 100 μs-delayed messages, synchronization is cost-effective when the hardware costs less than 23% of the system cost, or $4500 given our cost assumptions.

These results indicate that simple synchronization hardware, i.e., HW-1, is likely to be justifiable for a system with a fast data network, since such hardware can easily be built for much less than $1700. More complex hardware like HW-All is less clearly justifiable, due to its higher parts cost and greater design requirements. The results for 100 μs-delayed messages accentuate the importance of synchronization hardware for systems with slow data networks: for the aggregate workload, synchronization hardware is nearly always cost-effective with 100 μs-delayed messages.

Focusing on individual benchmarks, we see that of the shared-memory benchmarks, only *water* strongly motivates purchasing synchronization hardware; the cost-effectiveness of synchronization hardware for the other three is doubtful. However, shared-memory overheads in Blizzard are quite high; reducing these overheads, e.g., with hardware support for shared memory, will increase the relative benefit of synchronization hardware.

For our message-passing benchmarks, only *dycore* and *WWT* motivate purchasing synchronization hardware. For *WWT*, synchronization hardware makes sense regardless of message latency. *WWT* is unique compared to the rest of our workload in that it must synchronize to ensure that no messages are in flight (Network-Done).

For our data-parallel benchmarks, the HW-All network is cost-effective and economically feasible regardless of message latency, and is arguably necessary in order to garner acceptable speedups. The parallel function invocations from the host processor necessitate the HW-All network; a decentralized SPMD-style computational model would lessen this necessity.

6 Related work

Numerous proposals for add-on synchronization hardware have recently appeared. PAPERS [9] is a low-cost synchronization network which supports fine-grain execution on workstation clusters, specifically operations on data aggregates as in a data-parallel program and VLIW-style execution. Among other operations, PAPERS supports a simple barrier and a single-bit AND operation, both with latencies of 2 μs. The design interfaces to a network of PCs running Linux, with connections through each PC's parallel port. Hall and Driscoll's COP network [11] provides synchronization support equivalent to the HW-All network; they claim that its cost is 2-3% of overall system cost, which for our workload is clearly feasible. Shang and Hwang have designed add-on synchronization hardware for cluster-based multiprocessors, allowing synchronization to be performed both within and between clusters [25]. The ALLNODE barrier synchronization hardware uses the broadcast facility of the Allnode switch to perform barrier operations; an arbitrary number of nodes can synchronize in a few microseconds, and the mechanism consumes less than 5% of network bandwidth [17].

Other machines besides the T3D, VPP500, and CM-5 have provided synchronization hardware separate from the data network. PASM, a hybrid SIMD/MIMD machine, uses its SIMD synchronous instruction-fetch mechanism as a barrier when in MIMD mode [6]. A MIMD version of a FFT benchmark with the hardware barrier support was 39% faster than a MIMD version without the hardware barrier.

Fast barrier synchronization has been found to speed up specific patterns of communication and computation. Brewer and Kuszmaul found that using the hardware barrier on the CM-5 to limit the rate of message injection and limit congestion improved performance by more than a factor of three [5]. The direct deposit message-passing library of Stricker et al. [27] uses hardware barriers, rather than employing buffering or handshaking, to ensure that messages have been delivered to their destination. For a 2-D FFT code, their system runs approximately 2.8 times faster than an optimized request-response message-passing library. Ramakrishnan et al. [20] present two methods for efficiently supporting deep control nesting in data-parallel programs by using synchronization hardware. The first solution employs a pair of single-bit OR and AND reductions and code transformations, and the second solution requires a MAX reduction but no code transformations.

Additional work has looked at synchronization hardware specifically for parallel discrete-event simulators. Reynolds proposed a separate synchronization network to compute minimum next-event times, minimum timestamps of unacknowledged messages, and to compute Network-Done [23]. Beaumont et al. propose to dynamically synthesize application-specific hardware synchronization for a desired simulator by using FPGAs [2]. For synchronous parallel discrete event simulators, they suggest synthesizing a barrier and a MIN reduction to determine the minimum of all next-event times.

7 Summary and conclusions

This paper examined the cost/performance tradeoffs of adding a separate synchronization network to a network of workstations. We studied the synchronization requirements of three important classes of applications, shared-memory, message-passing, and data-parallel, experimentally measured the performance benefit obtained from hardware synchronization support. We combined these experimental results with cost estimates to calculate the cost/performance break-even point where hardware synchronization support becomes cost-effective.

- For our aggregate workload, add-on synchronization hardware is cost-effective if it costs less than 8% of the base system cost, for a fast data network, and less than 23% of the base system cost, for a slow data network.
- Individual applications may benefit much more from synchronization hardware. Among our applications, the HW-1 hardware improves performance by up to 54% on a NOW with the fast data network and up to a factor of 3 with the slow data network. The HW-All

hardware yields further improvement, increasing performance by factors of up to 2.5 and 3.6, for the two network latencies.

Our results show that global synchronization hardware can be a cost-effective addition to a NOW for some workloads. On the other hand, nearly half our applications received little benefit from synchronization support. A key advantage of add-on synchronization hardware—as opposed to the integrated synchronization hardware in most MPPs—is that only those people that can benefit from it need to buy it.

References

[1] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

[2] C. Beaumont, P. Boronat, J. Champeau, J.-M. Filloque, and B. Pottier. Reconfigurable technology: An innovative solution for parallel discrete event simulation support. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, pages 160–163, July 1994.

[3] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathon Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[5] Eric A. Brewer and Bradley C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *Proceedings of the Eighth International Parallel Processing Symposium (IPPS)*, pages 858–867, Cancun, Mexico, June 1994.

[6] Edward C. Bronson, Thomas L. Casavant, and Leah H. Jamieson. Experimental Application-Driven Architecture Analysis of an SIMD/MIMD Parallel Processing System. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):202–215, April 1990.

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

[8] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Toward a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–12, May 1993.

[9] H. G. Dietz, W. E. Cohen, T. Muhammad, and T. I. Mattox. Compiler Techniques For Fine-Grain Execution On Workstation Clusters Using PAPERS. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[10] G. Fox et al. *Solving Problems on Concurrent Processors*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[11] Douglas V. Hall and Michael A. Driscoll. Hardware for Fast Global Operations on Multicomputers. In *Proceedings of the Ninth International Parallel Processing Symposium (IPPS)*, pages 673–679, Santa Barbara, CA, April 1995.

[12] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[13] R. E. Kessler and J. L. Schwarzmeier. CRAY T3D: A New Dimension for Cray Research. In *Proceedings of COMPCON 93*, pages 176–182, San Francisco, California, Spring 1993.

[14] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1992.

[15] B. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II–175–II–179, August 1989.

[16] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, 1995. To appear.

[17] Howard T. Olnowich. ALLNODE Barrier Synchronization Network. In *Proceedings of the Ninth International Parallel Processing Symposium (IPPS)*, pages 265–269, Santa Barbara, CA, April 1995.

[18] G. Parisi. *Statistical Field Theory*. Addison-Wesley, Reading, MA, 1988.

[19] Steven Przybylski, Mark Horowitz, and John Hennessy. Performance Tradeoffs in Cache Design. In *15th Annual International Symposium on Computer Architecture*, pages 290–298, June 1988.

[20] Vara Ramakrishnan, Isaac D. Scherson, and Raghu Subramanian. Efficient Techniques for Fast Nested Barrier Synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1995.

[21] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM*

Sigmetrics Conference on Measurement and Modeling of Computer Systems, pages 48–60, May 1993.

[22] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[23] Paul F. Reynolds, Jr. An Efficient Framework for Parallel Simulations. In *Proceedings of the SCS multi-conference on Advances in Parallel and Distributed Simulation*, pages 167–174, Anaheim, CA, January 1991.

[24] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[25] Shisheng Shang and Kai Hwang. Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 6(6):591–605, June 1995.

[26] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[27] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers. In *Proceedings of the Ninth International Conference on Supercomputing (ICS)*, Barcelona, Spain, July 1995.

[28] Max J. Suarez and Lawrence L. Takacs. Documentation of the ARIES/GEOS Dynamical Core: Version 2. Technical Report 104606, Vol. 5, NASA Goddard Space Flight Center, March 1995.

[29] Teruo Utsumi, Masayuki Ikeda, and Moriyuki Takamura. Architecture of the VPP500 Parallel Supercomputer. In *Proceedings of Supercomputing '94*, pages 478–487, Washington, D.C., November 1994.

[30] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[31] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993. Also appeared in *CMG Transactions*, Spring 1994.

[32] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.