To appear in the Proceedings of the 1994 International Parallel Processing Symposium

How to Get Good Performance from the CM-5 Data Network

Eric A. Brewer Bradley C. Kuszmaul
MIT Laboratory for Computer Science*

Programmers of the Connection Machine CM-5 data network can improve the performance of their data movement code more than a factor of three by selectively using global barriers, by limiting the rate at which messages are injected into the network, and by managing the order in which they are injected. Barriers eliminate target-processor congestion, and allow the programmer to schedule communications globally. Injection-reordering improves the statistical independence of the various packets in the network at any given time. Barriers and tuned injection rates provide forms of flow control. Barriers also provide a composition of performance property: if you understand the performance of parallel computations A and B, then you understand the performance of "A; barrier; B". Architectural support for global barriers, injection reordering, and flow control may be worthwhile for achieving good communications performance. Although our evidence comes from the CM-5, we expect these techniques to apply to most parallel machines.

1 Introduction

Suppose you need to perform a sequence of parallel cyclic shifts. In parallel FORTRAN you might see code that looks like this, where the columns of `A` and elements of `B` are distributed among the `P` processors:

```
DIMEN A[P,P], B[P]
A[1] = CSHIFT(B,1)
A[2] = CSHIFT(B,2)
...
A[P] = CSHIFT(B,P)
```

The notation "`A[I] = CSHIFT(B,I)`" means cyclic-shift `B` by `I` and store it into row `I` of `A`.

One natural way to compile data-parallel code is to compile each statement separately with global barriers between the statements: both the source language and the compiled code are a serial sequence of parallel operations. A simple subscript analysis reveals that the barriers are not required to ensure the semantic correctness of this program, since the target rows are all independent.

Question: What happens to the performance when the barriers are removed?

Answer: Surprisingly, the computation gets slower, often by a factor of three.

To understand this problem, we studied a few patterns in a narrow environment:

- We studied the cyclic-shift pattern described above, and the resulting all-pairs communication pattern, in which every processor sends a value to every other processor. The all-pairs pattern appears in sorting and some scientific codes [Ede91]. We also studied random communication patterns.
- We studied the communication patterns in a data-parallel or SPMD environment, in which the real operation being performed by a collection of messages is a bulk data movement. Given this assumption, we examined the problem of sending a large collection of messages as quickly as possible, rather than focusing on the performance of any particular message.
- We used block transfers built on top of 20-byte active messages [vECGS92] on the CM-5 data network [LAD+92].

We found several ways to ensure good performance for bulk data movement. Barriers improve the performance of cyclic shifts by a factor of 2 to 3. The order in which packets are injected into the network is important. If a processor has many packets to send to each of several processors, it is better to interleave the packets to several destinations rather than send large batches of packets to one target. Finally, artificially limiting the injection rate to match the reception rate can improve performance an additional 25% and greatly reduces the variance in bandwidth for large transfers.

Although the scope of this study has been narrowed from the wider problem of obtaining good performance on any interprocessor communications system, we believe that our conclusions apply to a fairly wide range of situations.

An important limitation in any network is the bisection bandwidth. In general, given a data-parallel communication operation, if you divide the processors of a machine into two sets, and then measure the bandwidth, B, in bytes per second, that the network could possibly provide across the corresponding cut, and you measure the amount of data, D,

*E-mail: `{brewer,bradley}@lcs.mit.edu` This work is supported by Project SCOUT, ARPA Contract MDA972-92-J-1032; by the National Science Foundation, grant CCR-8716884; by ARPA, Contract N00014-91-J-1698; by an equipment grant from Digital Equipment Corporation; and by grants from AT&T and IBM. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

in bytes, that must be transferred between the sets, then it will take at least D/B seconds to move the data. The bisection bandwidth of a machine is one such cut. For any given cut, B is a function only of the network, and D is a function only of the communication pattern.

There are 2^P ways to cut P processors into two sets, which makes finding the worst cut a potentially formidable operation. A max-flow min-cut algorithm can be used to solve for the achievable bandwidth directly. Leiserson [Lei85] showed that for fat-trees the problem is much easier: you need only consider the cuts across a single major arm of the fat-tree in order to find the tightest D/B bound (see Figure 1). In a P-node CM-5, which is a uniform 4-ary fat-tree, there are less than $\frac{4}{3}P$ major arms, and the bandwidth of an arm depends only on the height of the arm. The bandwidths for a 64-node CM-5 are shown in Figure 1. We found that the most important cuts for the CM-5 are the links that connect the processors to the network: the bandwidth of these links are determined by software overhead and form the limiting factor for most message patterns.

Sections 2 and 3 provide background on the CM-5, Section 4 examines how barriers can improve communication performance, and Section 5 explores the importance of interleaving packets to multiple destinations. Section 6 examines the effect of matching the injection and reception rates. Section 7 discusses the implications of our results for communications software and hardware, and concludes with a discussion of our results and related theoretical work.

2 CM-5 Background

This section provides some background on the CM-5 data network and examines the fundamental limitations of the machine, including network-processor bandwidth and network capacity.

The CM-5 data network is a 4-ary fat-tree as shown in Figure 1. Each edge is actually two independent links, left and right, but for bulk data movement we always use both simultaneously. Of the various network cuts, at least two matter in practice: the links connecting the processors and the cuts through the root.

Processor overhead limits the bandwidth of the processor-network links, not the network hardware. For these links, the hardware can support up to 40 megabytes per second in each direction. Assuming the 33-megahertz clock found in most CM-5 implementations, and 20-byte packets with 16 bytes of payload (also standard for the CM-5), the sending overhead out of the cache is at least 37 cycles, for a maximum *payload bandwidth* of $\frac{(16\times10^{-6})(33\times10^{6})}{37} = 14.3$ megabytes per second.

The real limit is the cost of receiving packets, which currently requires about 60 cycles for realistic packets with polling and hundreds of cycles using interrupts. Because of the prohibitive cost of interrupts, all of our experiments use polling; we will discuss the implications of this decision in our conclusions. At 60 cycles per 16-byte packet, the payload bandwidth is limited to 8.8 megabytes per second. Kwan, Totty, and Reed [KTR93] measured the actual one-way bandwidth at 8.3 megabytes per second using Thinking Machines' message-passing library.

However, these numbers only cover the case in which a processor is sending or receiving. When a processor is both sending and receiving, the bidirectional bandwidth is somewhere between the two cases. Although the network handles both directions in parallel, the processor can not. The overhead to send and receive a packet is about 90 cycles, saving 7 cycles due to shared code. This translates to an upper bound of 5.9 megabytes per second in each direction for a total of 11.8 megabytes per second. The largest value measured by Kwan *et al.* was 10.4 megabytes per second.

The *capacity* of the network, which is the number of packets that can be injected without the receiver removing any, limits the ability of the processors to work independently. For example, if the network can hold ten packets, then a processor can only inject ten packets before the network backs up, and then it must wait for the receiver to accept packets. For the CM-5, we measured the network capacity for a variety of partition sizes:

Nodes	Total Packets	Packets/Node
8	79.0	9.88
16	158	9.89
32	342	10.7
64	691	10.8
128	1441	11.3

Thus, for any substantial data movement, the senders and receivers must be coordinated. Furthermore, since only the wire time can be hidden and the network capacity is only ten packets, there is little profit in trying to overlap computation and communication on the CM-5.

Nearly all communication on the CM-5 is implemented with active messages. Active messages were developed by von Eicken *et al.* [vECGS92], whose Berkeley CMAM package provided substantially better performance than contemporary versions of CMMD, Thinking Machines Corporation's communication library. CMMD 3.0 incorporated the active-message ideas, and in fact, most of CMMD is now implemented via active messages. Like CMAM, CMMD provides support for barriers and block transfers.[1]

The Strata communications library [BB94], developed at MIT, is an alternative to CMAM and CMMD that provides improved performance, improved support for timing and debugging, precise control over polling, and split-phase control-network operations.[2] Strata incorporates the techniques described in this paper.

[1]The CMMD performance numbers presented in this paper were measured under CMMD 3.1-Final with CMOST 7.2-Final.

[2]Strata is available from `ftp.lcs.mit.edu` via anonymous ftp, directory `/pub/supertech/strata`.

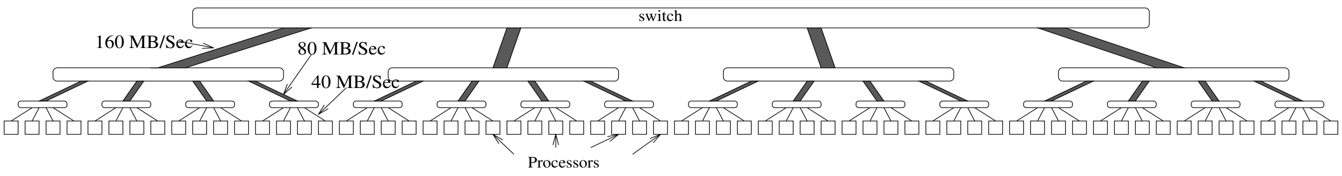

Figure 1: A 64-node CM-5 data-network fat-tree showing all of the major arms and their bandwidths (in each direction). One needs to cut only a single major arm to find the worst bisection for a given message pattern.

3 Timing on the CM-5

We use two forms of timing depending on the expected length of the event. For short events, less than a millisecond or so, we use the 32-bit cycle counter. For longer events, we use the 64-bit timers provided by the operating system.

The cycle counter is extremely accurate: using inline procedures the overhead can be subtracted out to yield timings that are accurate to the cycle. However, the cycle counter counts everything including interrupts and other processes. For example, if our process is time-sliced during an event, then we count all of the cycles that elapse until we are switched back in and the event completes. However, the probability of getting switched out during a one-millisecond event is about 1 in 100, since the time-slice interval is one-tenth of a second. A more common problem is the timer interrupts, which occur every 60^{th} of a second.[3] Thus, to get reliable measurements, we usually perform a timing at least three consecutive times and take the median. Using the median effectively eliminates errors due to time slicing and timer interrupts.

The operating-system timers have their own advantages and disadvantages. The operating-system timers stop running when your process stops running, so they can be used even across time-slice interrupts. However, the operating-system timers are accessed using system calls, which cost hundreds of cycles and thus limit the accuracy. The operating-system timers also experience a time-dilation when the data network is full of messages.

To demonstrate the time-dilation of the operating-system timers, we ran the following experiment. We timed a floating-point loop with the network empty, filled up the network, and timed the same loop with the network full. The only difference between the "empty" time and the "full" time is the presence of undelivered messages sitting in the network. The floating-point code is identical, no messages are sent or received during the timing, there are no loads or stores, and the code is a tight loop to minimize cache effects. Figure 2 shows the results for 18 samples taken across a wide range of overall system loads. Not only does filling up the network increase the measured time to execute the floating-point loop by an average of 21%, but it substantially increases the variation in measured time as well, as shown by the wider 95% confidence intervals.

Network Status	Average	95% CI
Empty	4.56 seconds	± 0.0020
Full	5.52 seconds	± 0.24

Figure 2: The effect of a full network on timings made with the operating-system timers: the timers inflate timings 21% when the network is full.

This implies that timings that occur while the network is full are dilated an average of 21%. The dilation is load dependent, but we were unable to get a reliable correlation between the dilation and the average system load. Fortunately, the timings appear to be consistent given a particular mix of CM-5 jobs, and the inflation appears to change slowly with time. To obtain reliable data, we ran the set of all experiments twelve times, measuring all of the experimental configurations once, and then measuring them all again, and so on, so that slow changes to the environment will tend to affect all of the experiments equally. Algorithms that keep the network full appear to achieve lower performance than algorithms that keep the network empty. We address the time-dilation issue in the context of each experiment.

We believe that the dilation is caused by context-switching. At each time-slice, the operating system empties the network, using the all-fall-down mechanism, so that messages in the network that belong to one process do not affect the next process. When the process is switched back in, its messages are reinjected before the process continues. The cost of context switching appears to depend on the number of packets in the network, and some or all of this cost is charged to the user and thus affects the timings. Our measurements have not been adjusted for time-dilation, since it appears that algorithms that keep the network full really do run slower. Our experiments with bulk data movement indicate that algorithms that keep the network full suffer more from network congestion than from time-dilation.[4]

[3] The timer interrupts take about 250 microseconds to complete, which means that a CM-5 (of any size) spends about $\frac{250}{16666} = 1.5\%$ of its cycles handling timer interrupts.

[4] The use of dedicated mode does not eliminate the dilation, although it does provide a more stable measurement environment.

4 Using Barriers Can Improve Performance

This section shows that adding barriers to a communications operation can actually increase performance, and presents some evidence to explain the benefit. To our knowledge this effect was first noticed by Steve Heller of Thinking Machines Corporation; Culler *et al.* mention the effect in a later paper [CKP+93].

We ran the cyclic-shift experiment as follows. On a 64-node CM-5, each processor sends a total of 1.28 megabytes using block-transfer primitives. Each processor, p, sends a block of data to processor $(p+1)$ mod 64, then sends a block to $(p+2)$ mod 64, and so forth. We vary the block size, B. For example, when B is 0.02 megabytes, each processor sends exactly one block to each processor; and when B is 0.01 megabytes, each processor cycles around twice, on each *round* sending one block to each processor. Figure 3 shows the performance of this cyclic-shift pattern as we vary B for both Strata and CMMD 3.1, with and without barriers.

The versions with barriers use a barrier between cyclic shifts; i.e., each processor sends B bytes, waits for the barrier, and switches to the next destination. The CMMD version with barriers must use Strata's barrier procedure. Unlike Strata's barrier, the CMMD barrier does not poll: combining it with block transfer leads to deadlock. You could use CMMD's barrier if the system was using interrupts instead of polling, but the loss due to interrupt overhead is prohibitively expensive.

Except for very small blocks, the versions with barriers perform much better. At 64-byte blocks (only 4 packets) the difference is small, but by 128 bytes per block, the difference is roughly a factor of two. For larger blocks, which are the common case, the difference is about a factor of 2.5. The substantial drop in bandwidth without the barriers is counterintuitive. Removing the barriers reduces the overhead and provides the data network with more opportunities to route packets. The increased opportunity, however, translates to decreased performance.

Some sort of interference occurs when packets from different batches interact. We were able to measure an interaction that we call *target collisions*. A target collision occurs when two packets arrive at the same processor at nearly the same time. Since packet reception is the bottleneck, target collisions can quickly back up the network. For large batches, target collisions could conceivably slow things down quite a bit; if for some reason, two processors each started sending a batch to the same processor at the same time, then the destination processor would be overloaded, the network would back up, and the performance would drop substantially.

To observe target collisions, we measured, at each instant in time, the number of packets in the network that are destined for a given processor. We did this by recording the time that each packet was injected into the network and the time that the target received the packet. We were able to use the globally synchronous cycle counter to obtain consistent times.

Figure 4 shows evidence of target collisions for one typical case: cyclic shifts with no barriers and a block size of 100 packets (1600 bytes). The plot shows for each processor, at each point in time, the number of messages destined for that processor. There are several interesting patterns in this data. At the far left there are some patterns that appear as a "warped checkerboard" pattern around 200,000 cycles. These patterns reflect the propagation of delays: a single packet was delayed, which caused the destination processor to receive packets from two different senders, which delays the injection of packets and thus exacerbates the problem. In short order, processors alternate between being overloaded (gray) and idle (white). By 400,000 cycles, some processors have as many as 17 packets queued up. The processors above the heavily loaded processors are nearly always idle and thus appear white. These processors are idle because several senders are blocked sending to their predecessor in the shift. The white regions thus change to black in about 100,000 cycles as the group of senders transition together. These black and white regions thus form "lines" that rise at an angle of about 20 degrees from horizontal; we have explicitly marked one of these lines.

Consecutive transfers incur collisions that do not occur when the transfers are isolated with barriers. Hot spots start due to random variation, and then persist systematically, getting worse and worse. The barriers increase the performance by eliminating target collisions.

We have explained, at least partly, the large differences between the experiments with barriers and the experiments without barriers. Now let us examine the other interesting features of Figure 3.

For the barrier-free codes, we expect the performance to drop monotonically as the block size increases, but for the largest block sizes, the performance increases unexpectedly. This is because for large block sizes, we end up doing very few cyclic shifts. For example, for a block size of 40,000 bytes, there are only $\frac{1280K}{40K} = 32$ different cyclic shifts. With so few transitions from one round to the next, the system never gets a chance to get as far out of sync, and the number of target collisions remains low. To demonstrate that large blocks suffer as much as medium-sized blocks, we tried running the Strata version without barriers for 100 shifts instead of 32, transferring about 3 times as much data:

Transfers	**MB/sec**	**95% CI**
32	2.22	± 0.122
100	1.67	± 0.0404

The new data point, 1.67 megabytes per second, fits right on the asymptote implied by the first half of the "Strata without Barriers" curve. Thus, without barriers, as the block size increases, the performance approaches 1.67 megabytes per second for Strata; the asymptote for CMMD is 1.43 megabytes per second.

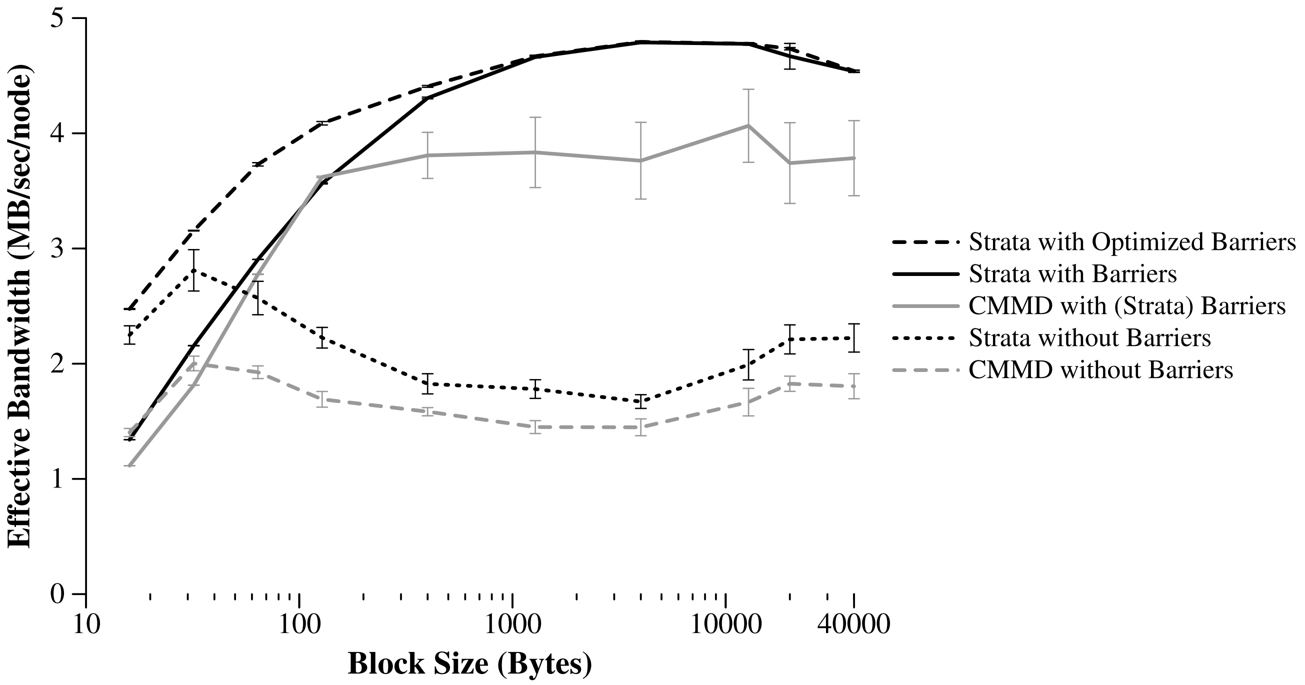

Figure 3: The effect of barriers between block transfers on the cyclic-shift pattern. The lines mark the average bandwidth; the error bars indicate 95% confidence intervals.

The performance of Strata with barriers drops slightly for very large transfers. This is due to cache effects: each sender sends the same block over and over. For all but the largest blocks, the entire block fits in the cache. The performance of CMMD with barriers does not appear to drop with the large blocks; in actuality it does drop but the effects are masked by the high variance of `CMAML_scopy`. The differences in performance and variance between Strata and CMMD are quite substantial; they are due to bandwidth matching and are discussed in Section 6.

The versions with barriers perform worse for small blocks simply because of the overhead of the barrier, which is significant for very small transfers. The "Optimized Barriers" curve shows an optimized version that uses fewer barriers for small transfers. The idea is to use a barrier every n transfers for small blocks, where n times the block size is relatively large, so that the barrier overhead is insignificant. The actual n used for blocks of size B is $n = \lceil \frac{512}{B} \rceil$. The choice of 512 is relatively unimportant; it limits barriers to about 1 for every 512 bytes transferred. Small limits add unneeded overhead, while large limits allow too many transitions between barriers and risk congestion. (At $n = \infty$, there are no barriers at all.)

Another important feature of Figure 3 is that the barrier-free versions have a hump for 16-byte to 400-byte block sizes. The increase from 16 to 32 bytes is due to better amortization of the fixed startup overhead of a block transfer. The real issue is why medium-sized blocks perform better than large blocks. This is because the number of packets sent in a row to one destination is relatively low; the packet count ranges from 2 to 8 for blocks of 32 to 128 bytes. With such low packet counts, the processor switches targets before the network can back up (the network can hold about ten packets per node). Between receiving and switching targets, the time for a transition allows the network to catch up, thus largely preventing the "sender groups" that form with longer block transfers. The next section examines this effect in more detail.

Finally, we have seen some cases in which using barriers more frequently than just between rounds can improve performance. This occurs because barriers act as a form of global flow control, limiting the injection rate of processors that get ahead. Related to this, Section 6 shows that artificially limiting the injection rate can improve performance; we do not yet understand exactly what is the additional benefit of extra barriers over limiting the injection rate directly.

5 Packets Should Be Reordered

After the selective use of synchronization, the most important technique for maximizing bandwidth is to randomize or interleave the packets. The previous section showed that large increases in bandwidth could be achieved by using barriers to prevent interference between adjacent rounds. However, when a round is not a permutation, collisions occur within the round and the benefit of synchronization is

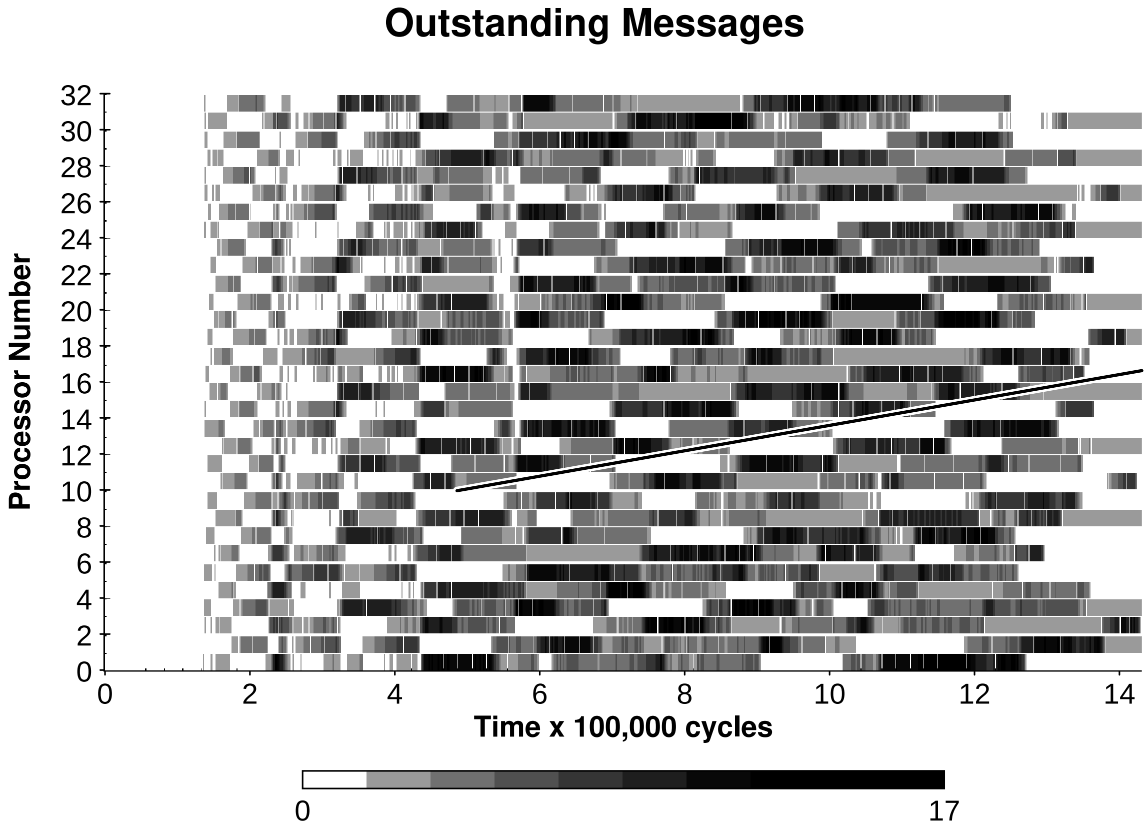

Figure 4: The total number of packets in the network headed to any given processor at any given time. Time is measured in 33-megahertz clock cycles.

minimal. In this section, we show that in the cases where collisions may occur within a round but the distribution of targets is still uniform, reordering packets is the key to performance.

Figure 5 shows the effective bandwidth versus the size of the block sent to each target. The "Strata Block Transfer with Random Targets" version picks each new target randomly from a uniform distribution. Thus, within a round we expect target collisions and barriers do not help. The key is that for small blocks the collisions do not matter because they are short lived. For large blocks the hot spots persist for a long time and thus back up the network, reaching the same asymptote as the cyclic-shift pattern without barriers, which also has a uniform distribution.

The key conclusion from this is that when the distribution is unknown, small batch sizes avoid prolonged hot spots. For example, if a node has ten buffers that require 100 packets each, it is much better to switch buffers on every injection (batch size of one), than to send the buffers in order (batch size of 100).

To explore this hypothesis, we built an asynchronous block-transfer interface. Each call to the asynchronous block-transfer procedure sends a small part of the transfer and queues up the rest for later. After the application has initiated several transfers, it calls a second procedure that sends all of the queued messages. The second procedure sends two packets from each queued transfer, and continues round-robin until all of the transfers are complete. To avoid systematic congestion, the order of the interleaving is a pseudo-random permutation of the pending transfers. Thus, each processor sends in a fixed order, but different processors use different orders.

The performance of this technique appears in Figure 5 as the two "Strata Asynchronous Block Transfer" lines. For random targets, interleaving the packets increases the bandwidth by a factor of about 1.8 for large blocks (more than 400 bytes) and performs about the same for small transfers. When used for the all-pairs cyclic-shift pattern, interleaving the packets increases the performance by a factor of 2.1 for large transfers, and by about 15% for small blocks. The cyclic-shift pattern performs better because the distribution is more uniform than random targets. The version with barriers still performs substantially better, but it requires global scheduling: each round must be a permutation. Thus, packet interleaving should be used when the exact distribution of targets is unknown. In general, the difference between the

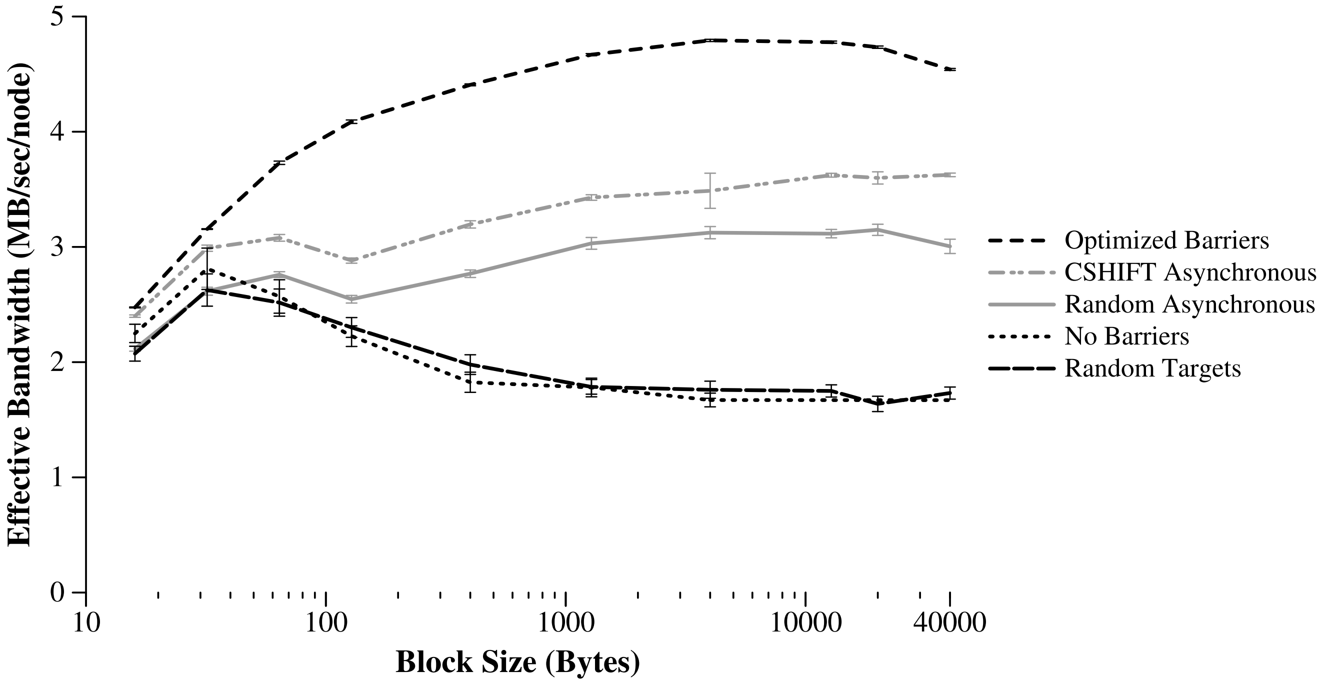

Figure 5: The effect of interleaving. The two asynchronous block transfer versions use packet interleaving to achieve about twice as much bandwidth as the corresponding normal block transfers. The version with barriers still performs much better, but it applies only when the communication can be structured as a sequence of permutations. The asynchronous block-transfer interface avoids this requirement.

asynchronous transfers and the version with barriers is due to the overhead for interleaving.

The dip at 128-byte transfers occurs because there is no congestion for smaller messages, and because the substantial overhead of the interface is amortized for larger messages.

Packet interleaving allows the system to avoid "head-of-line" blocking, which occurs when packets are unnecessarily blocked because the packet at the head of the queue is waiting for resources that those behind it do not need. Karol *et al.* showed that head-of-line blocking can limit throughput severely in ATM networks [Kar87]. Although Strata tries to send two packets from each message at a time, if it fails it simply moves on to the next message. This has no effect on the CM-5, however, because the network interface contains a FIFO queue internally, which allows head-of-line blocking regardless of the injection order. Thus, we expect interleaving to be a bigger win on machines that combine deterministic routing with a topology that has multiple output directions, such as a mesh or torus. In general, all levels of the system should avoid head-of-line blocking.

The benefit of interleaving has important consequences for message-passing libraries. In particular, any interface in which the the library sends one buffer at a time is fundamentally broken. Such an interface prevents interleaving. Unfortunately, the one-buffer-at-a-time interface is standard for message-passing systems. To maximize performance, a library must allow the application to provide many buffers simultaneously. The Strata interface seems quite robust, although it works best with at least four transfers at a time.

6 Bandwidth Matching

Given that the receive overhead limits the effective bandwidth, there is no point is injecting packets any faster than the receive rate. In this section, we show that artificially limiting the injection rate improves throughput and reduces the variance in effective bandwidth of bulk data movement.

Given that every node is both sending and receiving, the ideal situation occurs when every node alternates between injection and reception. Furthermore, we would like the network to contain as few packets as possible, yet still ensure that each node a has a packet ready to be received.

Because we use polling, we can limit reception to at most one per send, unless the send fails in which we must poll to prevent deadlock. Unfortunately, this achieves only about 2 megabytes per second because the network becomes very congested. Thus, CMMD and Strata always choose reception over injection: they poll until the network is empty.

Although this strategy performs much better than limited polling, it is fundamentally unfair: nodes that get a little behind may never catch up until others finish sending. The key problem is that nodes that are sending and have no pending arrivals inject packets faster than the receiver can pull them out. Furthermore, because overloaded targets are not sending, other nodes are likely to have no pending arrivals,

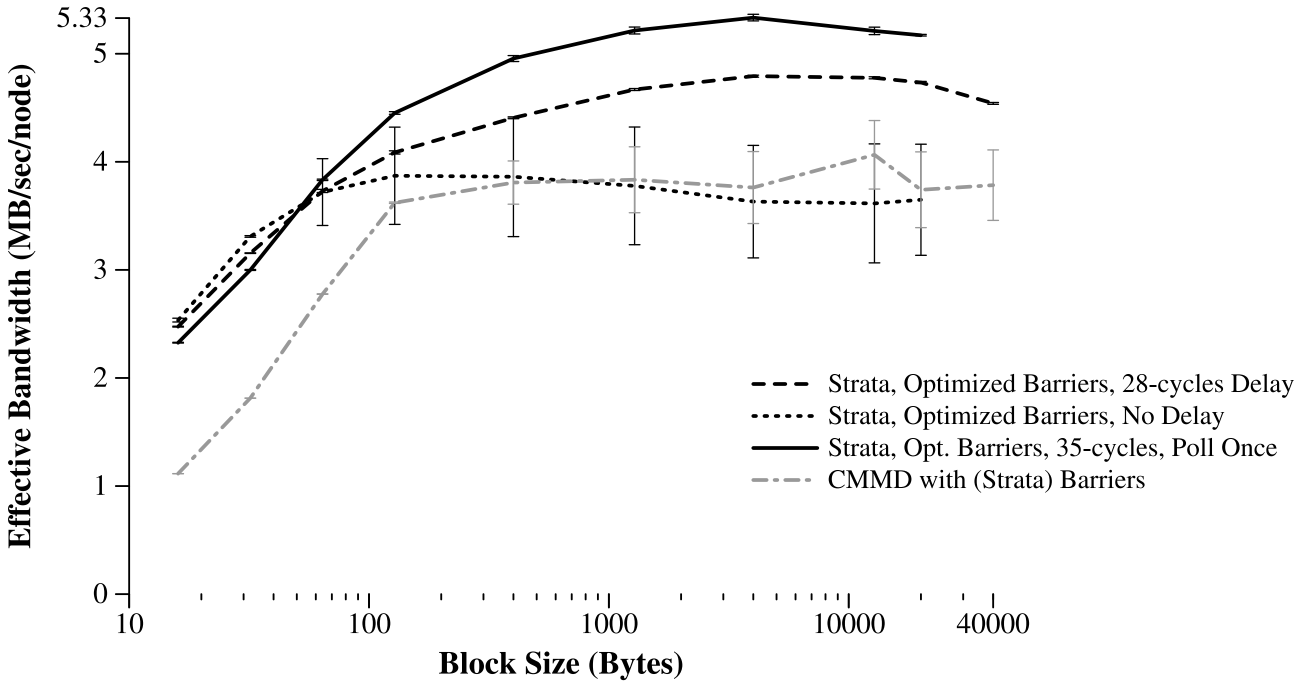

Figure 6: The effect of bandwidth matching on permutations separated by barriers. The error bars show 95% confidence intervals. All of the versions except "Poll Once" poll the network until it is empty. The delay value is how long the sender waits in the case that no packet arrived.

which exacerbates the problem.

Our solution is to delay injection artificially in the case that there are no pending arrivals. This ensures that the receiver pulls out packets faster than they arrive and eventually empties the network. Thus, an overloaded receiver quickly catches up and resumes sending. Because we are artificially limiting the injection rate based on the expected throughput, we call this technique *bandwidth matching*.

Figure 6 shows the impact of bandwidth matching on the cyclic-shift pattern with barriers. Without any delay, Strata actually performs worse than CMMD. This occurs exactly because Strata has lower overhead and thus a correspondingly higher injection rate than `CMAML_scopy`. Increasing the delay to 28 cycles ensures that the injection rate is slightly slower than the reception rate: the sending overhead becomes $37 + 28 = 65$ cycles, while the receiving overhead remains at 62 cycles.

The added delay not only increases the performance by about 25%, it also reduces the standard deviation by about a factor of 50. The drop in variance occurs because the system is self-synchronizing: any node that gets behind quickly catches up and resumes sending.

The poll-once version takes this a step farther. Given that everyone is sending at nearly the same rate, it is now sufficient to pull out only one packet, since it is unlikely that there will be two packets pending. Polling when there is no arrival wastes 7 cycles. This accounts for about a 7% improvement in throughput. The actual improvement is closer to 10%. The additional gain is due to the fact that all nodes run in lock step, which ensures that all nodes finish at nearly the same time. Unlike the case without bandwidth matching, the network remains uncongested even though less polling occurs. Optimum performance requires both bandwidth matching and limited polling. Note that Strata sustains more bandwidth *for all-pairs* than Kwan *et al.* saw for individual messages, 10.66 versus 10.4 megabytes per second [KTR93]. The net improvement over CMMD without barriers is about 390%.

Although limited polling can improve performance, it is not very robust. When other cuts such as the bisection bandwidth become bottlenecks, limited polling causes congestion. We expect that limited polling is appropriate exactly when the variance of the arrival rate is low; if the arrival rate is bursty (due to congestion), the receiver should expect to pull out more than one packet between sends. In practice, this means that patterns such as 2D-stencil that do not stress the bisection bandwidth can exploit limited polling, while random or unknown patterns should poll until there are no pending packets.

Introducing delay for short transfers actually hurts performance, as shown by the superior performance of the "No Delay" version for small transfers. In this case, the startup overhead introduces plenty of delay by itself; any additional delay simply reduces the bandwidth. Thus, future versions of Strata will adjust the delay depending on the length of the transfer. This form of adaptive delay should also remove the performance dip that appeared in the asynchronous block-transfer curves. For the limited polling case, it was beneficial to increase the delay slightly to 35 cycles to ensure that the bisection bandwidth did not affect the arrival rate.

This technique is essentially a static form of flow control. Traditional flow control via end-to-end acknowledgements would be more robust. However, it is very expensive, since each acknowledgement requires overhead at both ends. A relatively cheap solution for many situations is to use barriers as all-pairs end-to-end flow control. In some early experiments we found that frequent barriers improved performance; some of this effect occurred because barriers limit the injection rate, for which bandwidth matching is more effective. We expect that frequent barriers are a more robust form of flow control because they are a closed-loop system. However, despite its lack of feedback, bandwidth matching is quite stable due to its self-synchronizing behavior. Finally, there has also been some theoretical evidence that introducing delays might improve performance [FRU92, GL89].

7 Implications for Hardware and Software

With naive message-passing software and processor-network interfaces, a parallel computer may achieve only a fraction of the communications performance of the underlying network. Our experiments indicate that there are several mechanisms and techniques that can be used at various levels in the system to ensure high performance.

We found three underlying mechanisms that can improve performance. Barriers can be used to quickly determine when all processors are finished sending, or when all are finished receiving. The order in which packets are injected into the network can be managed; we studied interleaving and randomized injection orders. The rate at which packets are injected into the network by the sender can be tuned to match the rate at which the target can receive messages.

We found several reasons why these mechanisms work. They can help avoid target collisions, in which several processosr are sending to one receiver at the same time. They can help to smooth out, over time, the bandwidth demands across various bisection cuts of the network; the mechanisms can help the programmer ensure that the packets in the network at any given time have independent, evenly distributed, destinations. These mechanisms also provide various forms of flow control, which improves the efficiency of the network. For example, barriers act as global all-pairs flow control, guaranteeing that no processor gets too far ahead at the expense of another processor.

The following rules-of-thumb can help programmers decide when and how to use each of these mechanisms.

- If possible, recast the communication operation into a series of permutations. Separate the permutations by barriers, and use a bandwidth-matched transfer routine, such as is provided by Strata, to implement each permutation. We found that this strategy can improve performance by up to 390%.

- If bandwidth matching is impractical, because, for example, the real bottleneck is some internal cut of the network, then using periodic barriers inside each permutation may help. We have seen cases where barriers within a permutation improve performance.

- If you know nothing about the communication pattern, you should try to arrange the communication into a bulk data transfer, and then use an interleaved or randomized injection order, as provided by Strata's asynchronous block-transfer mechanism. Even in this case, periodic barriers within the transfers may improve performance.

- It is important to keep the network empty. It is almost always better to make progress on receiving than on sending. The one exception occurs when the variance of the arrival rate is near zero (due to bandwidth matching), in which case any additional polling wastes cycles.

- If your computation operation consists of two operations each of which has good performance separately, then keep them separate with a barrier. It is difficult to overlap communication and computation on the CM-5 because the processor must manipulate every packet, and the low capacity and message latency of the CM-5 network reduce the potential gain from such overlap. However, large block transfers interact poorly with the cache; we have seen cases where limited interleaving of the communication and computation can improve communications performance by about 5%.

Our results indicate that it may be a good idea to place some of our mechanisms into the network and the processor-network interface. A parallel computer should provide a rich collection of global flow-control mechanisms. Almost any form of flow control is better than none; we do not yet fully understand when to apply each of the various flow-control mechanisms. It may be helpful to have hardware support to determine more about dynamic network congestion, in addition to the CM-5's mechanism that indicates the presence of an arrival.

A parallel computer should support fast predictable barriers. The cost of a barrier should be competitive with the cost of sending a message. The behavior of barriers should be independent of the primary traffic injected into the data network. The CM-5 provides such barriers by using a hardware global-synchronization network; multiple priorities or logical networks could also be used. It may be beneficial for the system to perform a periodic barrier automatically to keep processors synchronized during communications operations.

The receiver must be at least as fast at the sender. Allowing user-level access to the network interface is the most important step in this direction. However, hardware support to speed up the receiving of messages even by a few cycles would help improve the programability of the CM-5.

The network, the processor-network interface, and its software should provide mechanisms to manage the order in which packets are injected into the network. A direct-memory-access (DMA) engine for sending and receiving packets, such as those proposed for MIT's *T [PBGB93] and Stanford's FLASH [KOM+94] machines, can make it easier to overlap communication and computation, but our experiments indicate that such engines may require fairly sophisticated packet-ordering and flow-control policies to achieve good performance. Similarly, very large packets are probably a bad choice because they have the effect of preventing packet reordering. The *entire* system must avoid head-of-line blocking.

We believe that our results apply to a wide range of parallel computers because the effects we observed are fundamental. Bandwidth considerations, scheduling issues, flow control, and composition properties will appear in any high-performance communications network. In particular, the rate at which a receiver can remove messages from the network may be the fundamental limiting issue in any network that has sufficient bandwidth to ensure that internal congestion is not the dominant issue.

Our experiments provide empirical evidence that some of the strategies used by theorists to prove theorems also make sense in real parallel systems. Theorists have argued that slowing things down can speed things up or provide predictable behavior [GL89]; we found that both barriers and bandwidth matching, which at some level slow down the system, actually speed things up. The use of barriers prevents processors that have gotten a little bit ahead from widening their lead. Parallel computing is not a marathon in which the first processor that finishes wins; it is a race against the clock in which we care about the finishing time of the slowest processor. Bandwidth matching is analogous to freeway on-ramp meters, which reduce the variance of the arrival rate to keep traffic flowing smoothly. Other examples of slowing things down to keep them working well include Ethernet's adaptive backoff [MB76], and the telephone system's approach to dealing with busy calls by forcing the caller to redial rather than wait in a queue [Kle76, p.103].

Not only are there sound theoretical arguments for randomized injection order [GL89], but we found significant practical application of reordering, even when we started with what we thought was a reasonable injection order.

Theorists have argued that measuring the load across cuts of a network is a good way to model performance and to design algorithms [LM88]; we have seen a few situations where we could apply such reasoning to make common patterns such as all-pairs run faster.

The MIT Strata library already incorporates these techniques for the CM-5. In the future, this research may lead to the development of mechanisms that routinely provide predictable, high-performance communication.

Acknowledgements

Robert Blumofe implemented most of Strata's active-message layer and block-transfer engine, and is involved in the ongoing work on bandwidth matching. Charles Leiserson and Bill Weihl suggested directions to pursue, and Charles provided the marathon analogy.

[BB94] E. A. Brewer and R. Blumofe. *Strata: A Multi-Layer Communications Library.* Technical Report to appear, MIT Laboratory for Computer Science, January 1994.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of PPoPP '93*, May 1993, 1–12.

[Ede91] A. Edelman. Optimal matrix transposition and bit reversal on hypercubes: all-to-all personalized communication. *Journal of Parallel and Distributed Computing,* **11**, 1991, 328–331.

[FRU92] S. Felperin, P. Raghavan and E. Upfal. A theory of wormhole routing in parallel computers. In *Proceedings of the* 33^{rd} *Symposium on the Foundations of Computer Science*, October 1992, 563–572.

[GL89] R. I. Greenberg and C. E. Leiserson. Randomized routing on fat-trees. *Advances in Computing Research,* **5**, 1989, 345–374.

[KOM+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. To appear in the *Proceedings of ISCA '94*, April 1994.

[Kle76] L. Kleinrock. *Queueing Systems — Volume I: Theory.* John Wiley & Sons, New York, 1976.

[Kar87] M. Karol, M. Hluchyj, and S. Morgan. Input versus ouput queueing on a space-division packet switch. *IEEE Transactions on Communication,* **35** (12), December 1987.

[KTR93] T. T. Kwan, B. K. Totty, and D. A. Reed. Communication and computation performance of the CM-5. In *Proceedings of Supercomputing '93*, November 1993, 192–201.

[Lei85] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers,* **C-34** (10), October 1985, 892–901.

[LAD+92] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. In *Symposium on Parallel and Distributed Algorithms '92,* June 1992, 272–285.

[LM88] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica,* **3**, 1988, 53–77.

[MB76] R. M. Metcalfe and D. R. Boggs. Ethernet: distributed packet switching for local computer networks. *IEEE Transactions on Computers,* **19** (7), July 1976, 395–404.

[PBGB93] G. M. Papadopoulos, G. A. Boughton, R. Greiner, M. J. Beckerle. Integrated building blocks for parallel computers. In *Proceedings of Supercomputing '93*, November 1993, 624–635.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In the *Proceedings of ISCA '92*, Gold Coast, Australia, May 1992.