# "Hot Spot" Contention and Combining in Multistage Interconnection Networks

GREGORY F. PFISTER, SENIOR MEMBER, IEEE, AND V. ALAN NORTON

***Abstract*—The combining of messages within a multistage switching network has been proposed [1], [11], [14] to reduce memory contention in highly parallel shared-memory multiprocessors, especially for shared lock and synchronization data. This paper reports on a quantitative investigation of the performance impact of such contention, performed as part of the RP3 project [7]–[9] and the effectiveness of combining in reducing this impact. We investigated the effect of a nonuniform traffic pattern consisting of a single *hot spot* of higher access rate superimposed on a background of uniform traffic. The potential degradation due to even moderate hot spot traffic was found to be very significant, severely degrading all memory access, not just access to shared lock locations, due to an effect we call *tree saturation*. The technique of message combining was found to be an effective means of eliminating this problem if it arises due to lock or synchronization contention.**

***Index Terms*—Concurrent computation, highly parallel systems, hot spots, message combining, multiprocessors, multistage interconnection networks, parallel processing.**

## I. INTRODUCTION

IN proposed highly parallel multiprocessor systems, e.g., systems with 100 or more processors, contention for memory access is a potential bottleneck. At the same time, it is often proposed that access to a shared memory in such systems be provided by means of a message- or packet-switched multistage switching network, with topologies such as a tree [12], Omega network or variant [1], [9], binary $N$-cube [11], etc. Two projects in this area, the NYU Ultracomputer [1] and the Columbia CHoPP (or GEM) [11], [14], have proposed the technique of *message combining* within the switch to use its multistage nature to help alleviate potential memory access bottlenecks. This technique (described in more detail below) merges similar references into composite *combined* references at each stage of the network.

However, the proposers addressed neither the detailed hardware cost nor the benefit to be derived from this technique in a quantitative fashion. This paper addresses these issues.

The hardware needed to support this feature is quite costly. As part of the RP3 project [7]–[9], detailed cost and size estimates were made based on state-of-the-art silicon and packaging data. These indicated that message combining increases the switch size and/or cost by a factor of between 6 and 32. The wide range is due to the variability of factors like circuit technology, packaging technology, and network topology.

To determine if this very significant added cost is worth the benefit derived, we performed a series of simulation experiments whose results are reported and interpreted in this paper. The following was determined.

1) A type of network traffic nonuniformity, a "hot spot," typically but not uniquely produced by global shared locks, can produce effects that severely degrade *all* network traffic, not just the traffic to shared locks. This effect, which we call *tree saturation,* has not previously been reported.

2) This effect is quite general. It is independent of network topology, switching mode (packet or circuit), or whether the network is used for memory access or message passing. It requires only a multistage network with distributed routing, and a network traffic pattern which, for any reason, exhibits "hot spot" nonuniformity.

3) Message combining, originally proposed to solve a different set of problems, is an effective technique for dealing with this problem when it arises due to global shared locks.

The technique of message combining is described below, along with the simulation experiments we performed and interpretation of the results.

We interpret these results as implying the need for message combining in any highly parallel multipurpose machine, such as RP3. The need for combining to avoid the effect of hot spots can be quantified; we in fact shall show that systems which do not employ message combining are limited by tree saturation in the degree of parallelism obtainable. How the results reported here are being applied in RP3 is also described.

It should be noted that message combining was proposed to solve problems different from those reported here. In [1], its justification is the elimination of serial bottlenecks in programs. In [11] and [14], its justification is broadcasting read-only data and reducing the latency of memory references in general memory traffic. While these claims may be true, we do not present evidence here to support them. In particular, we have, and presently know of, no quantitative evidence to support or deny the value of combining in general (i.e., not locking or synchronizing) memory traffic. Such information is difficult to obtain because it relies on the dynamic properties of large-scale parallelism, properties which are not observable without impractically detailed system-wide simulation. RP3 is itself intended to provide a

Manuscript received February 1, 1985; revised May 30, 1985. This paper appeared in the IEEE 1985 International Conference on Parallel Processing, St. Charles, IL, Aug. 1985.

The authors are with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

0018-9340/85/1000-0943$01.00 © 1985 IEEE

test environment to determine whether such combining is beneficial.

## II. Message Combining

Message combining works by detecting the occurrence of memory request messages directed at identical memory locations as they pass through each switch node. Such messages are combined, at the switch node, into a single message. The fact that combining took place is recorded in a *wait buffer* in each switch node. When the reply to a combined message reaches a node where it was combined, multiple replies are generated to satisfy the multiple individual requests. Since in successive switch stages combined messages can themselves be combined, the generation of multiple replies produces the effect of a dynamically generated broadcast of data to multiple processors.

The form of message combining described above is that of the NYU Ultracomputer. The Columbia ChoPP/GEM scheme of "repetition filter memories" (RFM's) operates somewhat differently, acting more like a cache at each network node, and may catch more combinable references. However, it appears to be an even more complex design, and since it is usable only for read-only data, it cannot, as will be seen, address the problem we later present.

## III. Simulated Switch

The specific method of combining investigated here uses a switch node that is a slight variation on the NYU Ultracomputer's. Its data flow is shown in Fig. 1. It is a two-way switch, and actually contains two separate switching nodes: one in the forward direction, which compares message addresses and performs combining, and one in the reply direction, which performs the required broadcasting.

The forward direction subnode is a standard 2 × 2 crossbar with output queues, with the following characteristics.

1) The output queues are used only when a succeeding stage indicates that it cannot accept a message. When not used, only a single stage of pipelining is seen by the message.

2) Comparisons are performed only between queued messages. Thus, no combining occurs if traffic is low enough that no queueing occurs.

3) The output queues can accept two messages simultaneously. This feature is used if two messages destined for the same output port arrive simultaneously under conditions where they must both be enqueued.

4) An additional buffer able to hold one complete message is associated with each input. It is used to hold a message in the event that the destined output queue is full. Without it, each node would have to signal to both its predecessors that it cannot accept input if either queue had less than two message slots free. With it, the signal that a message cannot be accepted on a given input is identical to that input's buffer being full. The buffer therefore allows greater output queue utilization; and since combining is done only in the output queues, this greater utilization implies that more opportunities for combining are available. These additional buffers are not present in the NYU design.

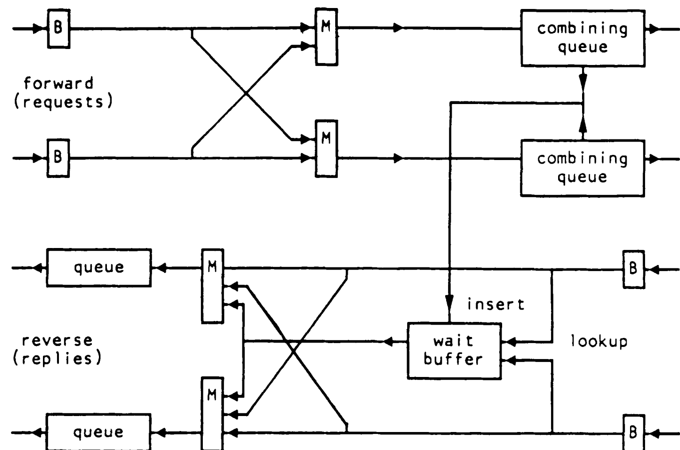



Fig. 1. Block diagram of the switch node used in simulation.

5) A message can combine with only one other message in a given node. A combined message can combine again in a later node.

6) One "packet" of a message travels from one switch node to the other in a single switch clock cycle (e.g., with an 8 bit data path a 64 bit message requires 8 clock cycles to go from one node's queue to the next).

7) The entire operation is completely pipelined, so that when all arriving messages are combinable, the arrival of two messages can be overlapped with the departure of a third message formed as the combination of two prior messages.

Saved information about combinations made in both input queues is held in a single wait buffer. Replies arriving from either reverse-direction port are decombined using information in that wait buffer. The output queues in the reverse direction are assumed to be able to accept up to four inputs simultaneously: two messages from the two reverse-direction inputs and two "decombinations."

More detail about how the comparison and combining takes place is available in [1].

## IV. The Experiments

Our network was configured as an omega network [13]. There were $N$ processors, $N$ memory modules, and the total switch contained $\log(N)$ ranks of $N/2$ switches with ranks connected by a shuffle-exchange connection. Our "processors" were simply generators of memory request messages. Our memories turned requests into replies in a single network cycle; this is unreasonably fast, but as will be seen it makes our results conservative.

Initially, we simulated a variety of network sizes ($4 \leq N \leq 64$) using the usual, analytically tractable, assumption that each source's memory references were independent and uniformly distributed across the entire address space. This was done to establish agreement with analytical models of switch performance (used in [8]), and to determine the queue length for which adequate performance was obtained (e.g., a length of four messages). Under those circumstances, virtually no combining occurred since the

probability that references to exactly equal addresses are queued in the same switch node at the same time is negligible.

Independent uniformly distributed references are not, however, an adequate model in the presence of global locks, even if all nonlock references are uniformly distributed. Locking operations do not work unless directed at identical memory locations.

We therefore altered the address distribution to be a "flat" (uniform) distribution with a single "spike" or *hot spot,* i.e., a single location to which a specified fraction of the total memory references was directed. That fraction was varied from 0.5 to 32 percent.

The simulation results with combining disabled are shown in Fig. 2. With combining enabled, the same experiments produced the results of Fig. 3. These figures show the steady-state average response time for a memory request as a function of the total switch traffic. The response time is in units of network cycles, and the switch traffic is in units of packets per network cycle per input. The lowest dashed line shows the analytically predicted response time with a uniform address distribution and infinite queue sizes. The other lines indicate the response time with various hot spot percentages.

It is important to note that the response times of Fig. 2 and Fig. 3 average response time of *all* memory requests, *not* just requests to the single lock location. If the delays for the hot spot traffic and background traffic are plotted separately, they are found to exhibit essentially identical behavior (the hot spot traffic does show slightly more delay). By comparison, if hot spot and background traffic are plotted separately when combining is being used, additional overhead appears only for references to the hot spot.

Combining clearly has a very substantial effect in reducing the average memory response time. However, two questions can be asked.

1) Without combining, why do all memory requests, not just those to lock locations, exhibit increased latency as the percentage of lock references rises?

2) Is the situation modeled realistic, i.e., do any practical situations correspond to the events modeled here?

These questions are discussed below.

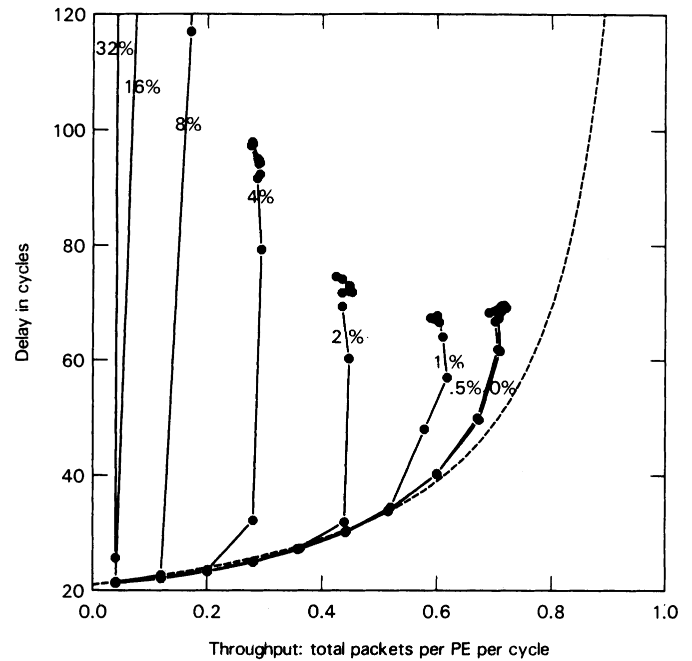



Fig. 2. Average memory latency in the presence of a hot spot, without combining, versus total network throughput for various percentages of network traffic referencing the hot spot. The switch nodes used had a queue size of four messages and a wait buffer size of six messages. The dashed line shows analytical estimates based on infinite queues.

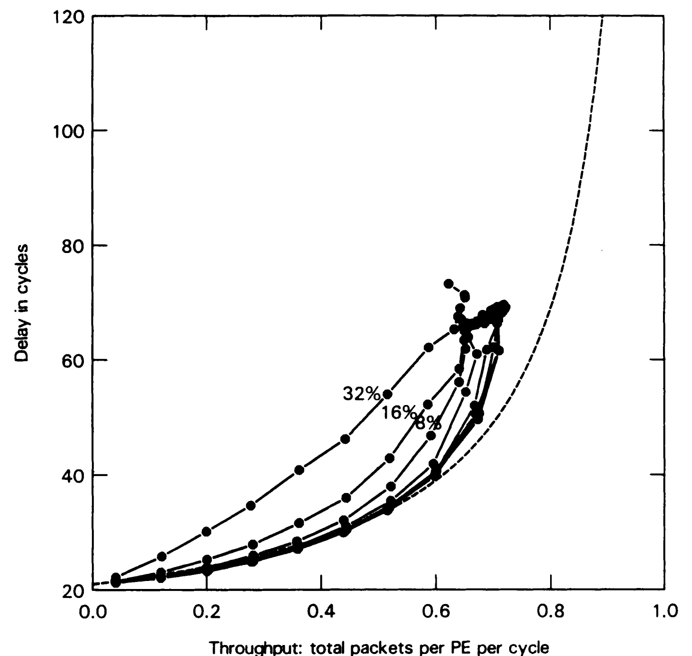



Fig. 3. The same experiments and conditions as shown in Fig. 2., but with combining used.

## V. Modeling of Noncombining Hot Spot Traffic

This section addresses the reasons why nonlock memory requests are delayed. A cause—*tree saturation*—is described, and its effect with regard to system scaling is discussed.

### A. Tree Saturation

Examining Fig. 2, one can notice that with a hot spot the latency climbs to an asymptote at the point where the total traffic and hot spot percentage combine to *saturate the weakest link* in the round trip from processor to memory and back.

More specifically, given that

- $p$ is the number of processors, and there are an equal number of memories
- $r$ is the number of network packets emitted per processor per switch cycle ($0 \leq r \leq 1$)
- $h$ is the fraction of memory references directed at the hot spot, i.e., each processor emits packets directed to the hot spot at a total rate of $rh$,

then the effective data rate into the "hot" memory module is $r(1 - h) + rhp$, i.e., the system attempts to send that many packets to the "hot" memory module every network cycle. The asymptote occurs when this value is equal to the

capacity of the weakest link in the round trip between a processor and the "hot" memory. In general, since loads dominate over stores and the response to a load is generally larger than the load request itself, the weakest link will actually be the interface between the memory and the return-trip network. In our simulations, however, all requests and replies were the same size, and the memories cycled in one switch cycle. As a result, the links into and out of the hot memory should saturate at the same point, namely, when the above formula equals unity. This is in close agreement with the simulated results.

Saturating the capacity of the link into the hot memory causes the queues in the switch closest to that memory to fill; then the same happens to the two switches in the prior rank that feed that one; then the same happens to the four in the next prior rank; etc. Thus, a tree of switches rooted at the hot memory and extending to all the processors is saturated, i.e., all the queues in that tree are full.

Since the tree has a leaf at every processor, all memory references from any processor to any memory module must begin within it, and therefore, all memory references, whether involved with the lock or not, are delayed. The fact that some of the references cross only very few levels of the tree is counterbalanced by the fact that they cross levels further from the memory, whose queues are emptying most slowly. Since the memory is filling requests serially, any fair routing scheme will cause the rate of queue service to decrease exponentially with distance from the memories.

### *B. Effect of System Size on Tree Saturation*

The effect of tree saturation as the number of processors is increased can be illustrated by noting that the asymptotically maximum network throughput is obtained when the expression derived above for the probability of reference to a hot memory module, $r(1 - h) + rhp$, equals 1. Thus, the asymptotically maximum value of the network throughput per processor, $R$, is defined by

$$R = \frac{1}{1 + h(p - 1)}.$$

More revealing is the expression

$$B = pR = \frac{p}{1 + h(p - 1)},$$

which gives the asymptotic limit of the total communication bandwidth available as a function of the number of processors and the hot spot percentage. This is plotted in Fig. 4 as a function of $p$ for various values of $h$. The amount of computation a system can do is very strongly related to the available communication bandwidth for a fixed processor architecture (neglecting input and output). The graph of Fig. 4 therefore indicates how hot spot contention, in the absence of combining, limits the speedup achievable with a given number of processors. The limitation from this effect alone is quite significant for large systems: with 1000 processors, only 0.125 percent hot spot traffic limits the potential speedup to 500, i.e., 50 percent efficiency.

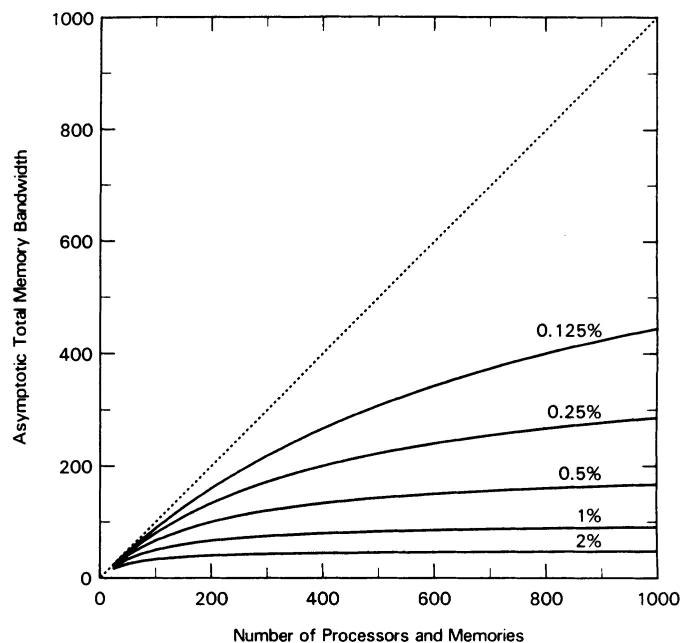



Fig. 4. Asymptotically maximum total network bandwidth as a function of the number of processors for various fractions of the network traffic aimed at a single hot spot.

## VI. EVALUATION

The simulation and analysis above provides strong evidence that tree saturation is a significant problem. What we claim, however, is somewhat more, namely, the *need* for combining and other techniques to avoid tree saturation in *any* large-scale multiple-purpose parallel system. Because this is a phenomenon which cannot occur in serial machines or in small-scale parallelism, it is necessary to ask how pervasive the problem is, and whether it might yield to simpler or less costly solutions.

### *A. Generality*

The tree saturation effect is not dependent on network topology for a multistage blocking network with distributed routing control. All such networks must contain trees from every sink to every source, which can become saturated and thereby delay all references. The existence of multiple paths through the network does not avoid the problem, even with dynamic rerouting. As congestion begins, the messages to the hot spot will themselves be rerouted and in the steady state will saturate all the alternate paths.

It is also clear that there is no requirement that the messages be memory references. Hot spot nonuniformity in the traffic through a purely message-based system can produce similar global degradation. It is in principle possible to use message combining in such systems, but doing so implies that the transport mechanism should be given some knowledge of the message semantics—specifically, how to combine them. How one does this has not been investigated.

An effect entirely analogous to tree saturation as presented here can also afflict circuit switching networks with distributed routing. In this case, there will of course be no effect on data transfer once it has begun, but the time required to complete a circuit prior to initiating the transfer will be af-

fected. The exact manner in which tree saturation occurs depends on the distributed routing technique used.

While the presentation here implies that tree saturation is a result of finite queue lengths, a very similar effect—with somewhat different causes—can be shown to occur under the assumption of infinite queues. We do not discuss this (and the associated analytical model) here due to space limitations.

### B. Realism

Whether the situation modeled is realistic can be divided into two questions.

1) Is there typically only one hot spot (or at most a small number)?

2) Is the traffic to the hot spots typically large enough to cause a problem?

The simulations performed on whole codes at NYU [2], [4], [5] did, in fact, typically contain only one or two hot spots at any given time during execution. We are beginning to augment such experimental results with our own traces of large parallel applications (not just kernels). The experience so far is that one or two hot spots are typical, although more can occur if cacheable data are not designated as such.

On the other hand, it may be argued that one application is not the right place to look for hot spot references, that the operating system will normally generate numerous such references during normal coordination because of the various central queues which are inherent in parallel systems. We therefore have also tested the effectiveness of combining with more hot spots, simulating, for example, five hot spots in separate memory modules. Essentially identical results were obtained; combining is still quite effective, and without it major degradation still occurs. (Were all the hot spots in the same memory module, identical results would be obtained, except that larger queues might be needed. Given more than 100 memory modules, address interleaving, and hashing, it is unlikely that more than one of a small number of hot spots will lie in the same module.)

For the NYU simulations, the percentage of total data requests to memory aimed at the hot spot was typically 1–2 percent [3]. This appears inconclusive since with 100 processors Fig. 4 indicates a maximum speedup of only 30–50 in this range of hot spot requests. But the situation is potentially much worse than that.

For performance reasons, a cache is usually interposed between the processors and the network, and other results indicate that approximately 80–90 percent of all data traffic can be intercepted by the cache. So the total network traffic, which is all we are concerned with, is 10–20 percent of the total data traffic. At the same time, none of the references to a globally accessed lock can be cached. So the 1–2 percent of total data traffic to the hot spot(s) represents 5–20 percent of the total network traffic. Under those conditions, Fig. 4 indicates that maximum performance is severely degraded: with 100 processors, the maximum speedup that can be attained is in the range of 5 to 20.

It can be argued that the NYU results are not representative. Indeed, they may not be since NYU developed this style of combining, was testing the concept, and certainly did not program with the intention of minimizing its use. A factor of 10 reduction in hot spot traffic might reasonably be obtained by coding that attempted to reduce that traffic.

In addition, there are a number of software techniques that can be used to reduce hot spot contention. For example, if a global sum is desired, creating it by filling in a tree of partial sums can, without combining, make the maximum contention a constant (equal to $N$ for an $N$-ary tree of partial sums), rather than proportional to the number of processors, as would be the case if fetch-and-add were naively used. We are, however, unaware of any way to get the full effect of a combining fetch-and-add—including distribution of the incremental sums—by software alone, without combining, and we suspect it can be proved that this is impossible.

### C. Pragmatic Considerations

The above discussion is inconclusive in the absence of a broader range of experience than is presently available; there will always be parallel applications which communicate or synchronize infrequently enough to not cause a problem, and there are applications which will consistently cause tree saturation. However, there are pragmatic points that must be considered.

Without combining, the potential major loss of efficiency that hot spot references can cause must be taken into account in all code written for a highly parallel system. We can scarcely afford to have this additional complexity permeate the programming task.

Furthermore, the running of multiple users in a multiprogramming environment on a highly parallel machine without combining has a serious flaw. Traffic to a single hot spot has a global effect. Therefore, it is possible for a *single* user with a highly parallel task that is not debugged, naive, or even malicious to degrade the *entire system's* performance via hot spot traffic. This is clearly an unacceptable situation. While it may be possible to avoid it by other means, the alternatives we are aware of impose more of a burden than the hardware cost of combining.

### D. The RP3 Combining Network

In addition to the above considerations, the RP3 design is subject to the constraint that, given existing technology, it is physically unrealistic to build a combining network sufficiently fast to support all memory references of 512 processors. The speed required for efficient memory access implies that the network should be built in a bipolar technology. However, combining is a logic-intensive function which benefits greatly from the density available in MOSFET technology.

It is therefore planned to use two different networks in RP3:

1) a high-speed multistage noncombining network with sufficiently low latency to handle the normal memory references of 512 processors, built in bipolar technology;

2) a smaller, slower combining network with characteristics adequate to support the synchronization references of 512 processors, built in MOSFET technology.

The required separation of memory traffic is achieved by diverting synchronization references, such as fetch-and-op [1], test-and-set, etc., into the combining network. All other references will use the noncombining network. For experimental purposes it will be possible to divert all traffic into either network.

Software and hardware measures in RP3 are planned to minimize the potential of tree blockage in the noncombining network. For example, experiments with parallel memory reference traces and cache simulation show that the most common occurrence of hot spots in loads and stores results from not caching global data which could be cached, for example, shared "constants" which are stored only once and then fetched many times. RP3 hardware allowing dynamic control of cacheability can be used by software to alleviate such problems.

## VII. Conclusion

By considering nonuniform *hot spot* memory reference patterns, we have demonstrated that multistage blocking networks have an unfortunate property. A sufficient concentration of references to one server—a "hot spot"—can degrade the response of the network to all references, not just those to the hot spot server, and the potential degradation is sufficient to cripple system performance. The "tree saturation" effect causing this requires only that the network be multistage, blocking, and controlled by distributed routing. Message combining is adequate to deal with this effect.

It is possible to avoid this problem in special-purpose systems, tailored to a particular application. Such systems can usually incorporate hardware that directly addresses this problem if it exists in their context.

It is also possible to avoid this problem in some multiple-purpose systems, if the application is carefully tailored to the communication topology and, by explicitly managing communication, happens to keeps the traffic completely uniform. This is the case, for example, with the "crystalline" mode of operation of the Cosmic Cube [10], the mode in which most of its applications have so far been run.

Message combining provides a solution to this problem. It requires no additional programming overhead, should be effective in a broad range of applications and environments, and should enable applications programmers to code without undue attention to excessive synchronization, with the additional assurance that no one user can degrade a multiuser system with his own excessive synchronization.

The additional cost of a combining network is outweighed by its potential advantages, and such advantages grow with the size of the parallel system. We consider the technique of message combining to be a required part of RP3, or of any other multiple-purpose parallel system of comparable size.

## Acknowledgment

The authors would like to gratefully acknowledge the assistance of A. Gottlieb, who wrote the initial version of the two-way switch simulation code we used, and M. Wong, who performed numerous simulation experiments on the effects of hot spots and combining with a variety of switch designs.

## References


[1] A. Gottlieb *et al.*, "The NYU Ultracomputer—Designing an MIMD, shared memory parallel computer," *IEEE Trans. Comput.*, pp. 175–189, Feb. 1983.

[2] M. Kalos, "Scientific computations on the Ultracomputer," Courant Inst., New York Univ., New York, Ultracomputer Note 30, 1981.

[3] ——, Courant Inst., New York Univ., New York, private communication.

[4] M. Kalos, L. Gabi, and B. D. Lubachevsky, "Molecular simulation of equilibrium properties. Parallel implementation," Courant Inst., New York Univ., New York, Ultracomputer Note 27, 1981.

[5] D. Korn, "Timing simulations for elliptic PDE's run under washcloth," Courant Inst., New York Univ., New York, Ultracomputer Note 31, 1981.

[6] C. Kruskal and M. Snir, "The performance of multistage interconnection networks for multiprocessors," *IEEE Trans. Comput.*, vol. C-32, pp. 1091–1098, Dec. 1983.

[7] W. C. Brantley, K. P. McAuliffe, and J. Weiss, "The RP3 processor/memory element," in *Proc. IEEE 1985 Int. Conf. Parallel Processing*, St. Charles, IL, Aug. 1985.

[8] V. A. Norton and G. F. Pfister, "A methodology for predicting multiprocessor performance," in *Proc. IEEE 1985 Int. Conf. Parallel Processing*, St. Charles, IL, Aug. 1985.

[9] G. F. Pfister *et al.*, "The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture," in *Proc. IEEE 1985 Int. Conf. Parallel Processing*, St. Charles, IL, Aug. 1985.

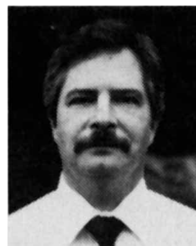
[10] C. S. Seitz, "The Cosmic Cube," *Commun. ACM*, vol. 28, no. 1, pp. 22–23, Jan. 1985.

[11] H. Sullivan, T. Bashkow, and D. Klappholtz, "A large scale homogeneous, fully distributed parallel machine," in *Proc. Fourth Annu. Symp. Comput. Architecture*, 1977, pp. 105–124.

[12] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "CM*—A modular, multimicroprocessor," in *Proc. AFIPS Conf.*, vol. 46, 1977, pp. 637–644.

[13] D. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, pp. 1145–1155, 1975.

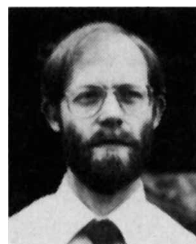
[14] L. A. Cohn, "A conceptual approach to general purpose parallel computer architecture," Ph.D. dissertation, Columbia Univ., New York, 1983.

**Gregory F. Pfister** (S'71–M'74–SM'85) was born in Detroit, MI, on November 29, 1945. He received the S.B., S.M., and Ph.D. degrees from the Massachusetts Institute of Technology, Cambridge, in 1967, 1969, and 1974, respectively, in electrical engineering.

He joined IBM in 1974, working between then and 1978 in several organizations on computer graphics software and remote software service. From 1975 to 1976 he was on the faculty of the Department of Electrical Engineering and Computer Science, University of California, Berkeley. In 1978 he joined the IBM Research Division, Yorktown Heights, NY, as a Research Staff Member. He was Manager of Software Support for the Yorktown Simulation Engine and is presently Manager of the Parallel Systems Architecture Group, in charge of RP3 architecture, performance evaluation, and software. His technical interests include parallel architectures, languages for parallel processing, VLSI design automation, and computer graphics.

Dr. Pfister is a member of Eta Kappa Nu, Tau Beta Pi, and Sigma Xi.



**V. Alan Norton** was born in Salt Lake City, UT, on August 20, 1947. He received the B.A. degree from the University of Utah, Salt Lake City, in 1968, and the Ph.D. degree from Princeton University, Princeton, NJ, in 1976, both in mathematics.

He was an Instructor at the University of Utah from 1976 to 1979 and an Assistant Professor at Hamilton College, Clinton, NY, from 1979 to 1980 before coming to IBM Research. Currently, he is a Research Staff Member at IBM, Yorktown Heights, NY, working on the Research Parallel Processing Prototype (RP3). His research interests include the performance analysis and architecture of parallel computer systems, parallel algorithms, fractals, and computer graphics.

Dr. Norton is a member of the Association for Computing Machinery and the American Mathematical Society.