The Case for Application-Specific Communication Protocols

Edward W. Felten

Department of Computer Science & Engineering
University of Washington

Technical Report 92-03-11
March 1992

The Case for Application-Specific Communication Protocols *

Edward W. Felten

Technical Report 92-03-11
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195 U.S.A.

March 24, 1992

Abstract

Message-passing programs are heavily dependent on the performance of communication primitives. While communication hardware has gotten much faster in the last few years, the communication performance achieved by application programs has not improved so dramatically. We argue that this *communication gap* is inherent in the design of conventional message-passing systems, which are based on general-purpose communication protocols implemented in the operating system. Closing the gap requires fundamental changes in system design.

We review the arguments for user-level implementation of communication protocols. We discuss how the architecture and operating system can be structured to provide user-level communication without compromising system protection.

We further argue that general-purpose communication protocols, whether implemented at user level or in the kernel, are inherently slow. Optimum performance requires a protocol designed to take advantage of the behavior of a particular application program. We introduce the notion of a *protocol compiler*, a program that generates an application-specific protocol, based on information about an application's communication behavior. As a concrete example, we sketch the design of a protocol compiler that can generate nearly optimal communication code for a class of data-parallel applications.

*This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Washington Technology Center, Digital Equipment Corporation (the External Research Program and the Systems Research Center), and Apple Computer Company. Felten was supported in part by an AT&T Ph.D. Scholarship and a Mercury Seven Fellowship.

1 The Communication Gap

As message-passing parallel computers have begun to mature, researchers have focused on the critical issues that will determine whether these machines will find wide use in the future. One of these issues is the performance of communication primitives, and the search for alternative semantics and implementations that provide better performance.

We use the term *communication gap* to describe the gulf between the performance of communication hardware, and the performance actually observed by application programs. New routing technology allows networks to achieve very low latency and high bandwidth [Rodeheffer & Schroeder 91, Int 91b, Dally 90a]; for example Intel's Touchstone DELTA system achieves a node-to-node hardware latency less than 1 microsecond. On the other hand, applications on the Touchstone DELTA observe a latency of about 60 microseconds. Although message-passing machines have always exhibited a communication gap, the gap is wider now than ever — a potential factor of 60 in communication performance is apparently being lost.

The communication gap affects application programmers in several ways. First, a relatively large communication latency makes many fine-grained algorithms impractical. Second, even when the underlying algorithm is not fine-grained, a large latency forces the programmer to optimize the program in ways that would not be necessary with a smaller latency; this adds considerably to the difficulty of programming distributed-memory hardware.

1.1 Hardware and Software Assumptions

Before going further, let us stop and enumerate our assumptions about the hardware and software un-

der study. We assume a high-performance message-passing parallel computer, consisting of a set of computing nodes connected by a point-to-point network. The network is based on circuit switching or wormhole routing, and uses a deterministic, deadlock-free routing algorithm. The network delivers messages reliably.

We also assume the system is running a general-purpose operating system on each node, including support for multiprogramming, paged virtual memory, and direct input/output calls. The system as a whole runs a variety of uniprocessor and parallel jobs. We will assume the system does not span administrative boundaries — system policy is set by a single individual or group.

Examples of such systems include the Intel DELTA [Int 91b] or Paragon [Int 91a] machines running the Mach operating system, or a set of dedicated workstations connected by a Nectar [Arnould et al. 89] or Autonet [Rodeheffer & Schroeder 91] network.

Although the machine is assumed to run a general-purpose workload, we will focus on improving the performance of parallel applications, especially data-parallel scientific programs. These programs have relatively simple behavior, and have much to gain from improved communication performance.

1.2 Sources of the Communication Gap

If we examine the sources of the communication gap, we find we can attribute it to two main causes: protection boundaries and protocol processing overhead.

1.2.1 Protection Boundaries

Providing protection is one of the main goals of an operating system. Unfortunately, operating system protection sometimes interferes with applications. For example, in most message-passing machines, communication is a privileged operation — threads from the same application, running on different nodes, cannot communicate without crossing protection boundaries.

Figure 1 illustrates the protection boundaries enforced by the hardware and operating system on a message-passing machine running a single application. The application is running in a separate protection domain on each node of the machine. The operating system kernel and the communication network reside in a single, privileged domain. When the application communicates between nodes, the communicated data must cross two protection boundaries — from the sending thread into the sending node's kernel, and

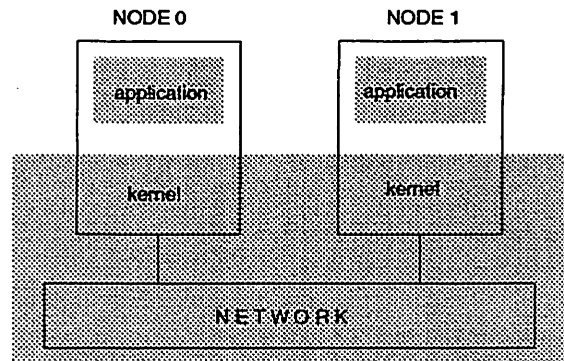

Figure 1: Protection boundaries on a conventional message-passing system. The system is running a single application on two nodes. Each shaded area represents a protection domain. Communicating between the two application threads requires crossing two protection boundaries.

later from the receiving node's kernel to the receiving thread.

Crossing protection boundaries is inherently expensive, for several reasons.

- Crossing between application and kernel protection levels implies a context switch between an application thread and a kernel thread. Context switches are expensive, since register state must be saved, and TLBs and caches may have to be flushed. Context switches are becoming relatively more expensive as CPU technology advances [Ousterhout 90, Anderson et al. 91].

- The entities communicating across the protection boundary do not trust each other, so parameters to cross-domain calls must be checked carefully for validity, and copied from the caller's memory to the callee's memory. For operating system calls, the kernel must typically assume that the application program is *trying* to cause a kernel error or deadlock. This slows down the kernel implementation.

- The interface at the protection boundary is usually part of the system specification, so it cannot realistically be customized to accommodate the application's needs. The application must "work around" the semantic gap between the interface it wants and the interface it is forced to use. If the interface provides too much functionality, the

application must pay for features it doesn't use; if too little functionality is provided, the application must built the abstractions it wants on top of the ones the system provides. In either case, the rigidity of the interface carries a cost.

If the same application is running on several nodes of the machine, there is no logical reason to protect the application's threads on different nodes from each other. If we can find a way to remove these protection boundaries, we will improve communication performance.

1.2.2 Protocol Processing Overhead

On most message-passing machines, the operating system runs a general-purpose communication protocol. This protocol typically buffers messages, often on both the sending and receiving nodes. It also implements a software flow-control strategy, to ensure that nodes have sufficient buffer space for arriving messages. Running the protocol requires extra computation, as well as extra messages to manage flow control.

By reducing or eliminating the protocol processing overhead, we could improve communication performance. Unfortunately, this is difficult in general, since the protocol cannot ensure correct operation without engaging in some buffering and flow control.

2 Eliminating Protection Boundaries

As shown in figure 1, message data must cross two protection boundaries as it travels from sender to receiver. Only the kernel can access the network directly; the application must ask the kernel to communicate for it.

We would like to change this organization, so the application can access the network directly, without operating system intervention. However, we must do this in a way that ensures protection between separate applications.

2.1 Protection in the Presence of User-Level Communication

If we simply unprotected the communication hardware, applications could violate the protection guarantees we require the system to provide. For example, an application thread could send a message directly to another job, or it could clog the network with messages and refuse to remove them. We must make some changes to the architecture and operating system, to ensure that application control of the communication hardware is safe.

We can achieve this goal by using techniques pioneered by Thinking Machines' CM-5 architecture [Thi 91]. The CM-5 provides safe, application-level access to communication hardware; the CM-5's mechanisms can be adapted to other, similar architectures.

We must make the following changes to the architecture and operating system:

- *Divide the machine into partitions.* The operating system must support partitioning of the machine, where a partition is a contiguous set of nodes and the network connecting them. Partitions have the property that sending a message between two nodes in the same partition uses only network channels within that partition; this ensures that mutual deadlock between applications cannot occur. Partitions can be changed in software, but this is a very heavy-weight operation.

- *Validate message destinations in hardware.* The hardware must check the destination of each message, to ensure that it is being sent to a node in the same partition.

- *Use strict gang scheduling.* Each parallel job must run in one partition, but several jobs may be running in the same partition, with the operating system time-slicing the partition's CPUs between them. To guarantee protection, we must use strict gang scheduling in each partition; this means that all nodes in the partition run the same job at the same time — nodes in a partition must all switch jobs at the same time. This prevents a message sent by one job from being delivered to another job in the same partition.

- *Support saving and restoring the network state.* When a partition context-switches from one job to another, we need some way of saving the state of the network. Generally, some messages (and partial messages) from the switched-out job will be in the network at the time of the context-switch, and these messages must be saved somewhere so the network is clear for the newly switched-in job. Later, when the original job is rescheduled, the saved network state must be restored.

  The CM-5 saves network state by putting a region of the network in "all fall down" (AFD) mode. AFD mode causes all messages to be delivered

to the nearest node, rather than to their destinations. Nodes receiving these messages save them in memory, and then resend them when the job is rescheduled.

AFD mode relies on two assumptions about the CM-5 network: message packets are limited to 20 bytes, and in-order delivery of messages is not guaranteed[1]. Both of these assumptions are false in our hardware model, so we cannot use AFD mode. However, it is not hard to design other hardware mechanisms that allow network state to be saved and restored.

- *Provide a separate logical communication network for the kernel.* The machine runs a general-purpose operating system, so the operating system kernel might have to communicate at any time. For example, the kernel might need to service a page fault, or receive the result of an input/output operation, or coordinate a context switch between jobs. The kernel must always be able to communicate, regardless of how application programs use the network.

  The only way to provide this guarantee is to have a separate logical network for the kernel. Each node must have *two* network interfaces, one for user-level programs, and one reserved for the kernel. Fortunately, we needn't implement two separate physical networks — the two logical networks can be multiplexed over a single physical network, using virtual channels [Dally & Seitz 87, Dally 90b] as in the J-machine [Dally 90a].

These strategies, taken together, enable safe, user-level manipulation of the communication network. However, this only solves half of the problem. The question remains: once user-level communication is available, how should use it?

3 Application-Specific Protocols

User-level access to the communication hardware allows us to run conventional, general-purpose communication protocols at the application level. The kernel must still provide communication, but this can be specialized to the case of communication between separate protection domains. Communication within an application would be provided by a runtime library that duplicates the functionality of a standard communication system such as NX [Int] or PICL [Geist et al. 91]. In addition to its organizational advantages, this approach offers better performance. For example, the communication system may allow an ill-behaved application to deadlock the (user-level) network; this implementation choice is not available when there is only a single, protected network.

However, application-level communication does not reduce protocol processing overhead. Running a general-purpose protocol at user-level provides only a small performance gain, since the protocol must still buffer messages and perform high-level flow control. General-purpose protocols cannot be sped up beyond a certain point.

We can improve performance still further by running application-specific protocols. By taking advantage of the programmer's or the compiler's knowledge about the communication pattern of an application, we can design a streamlined protocol that is designed especially for that application. While a general-purpose protocol must be robust in the face of arbitrary application behavior, an application-specific protocol can ignore certain behaviors that the programmer or compiler knows will not occur.

Examples of useful knowledge that a protocol designer might exploit include:

- each node communicates only with a small, known set of other nodes, or
- all communication in a certain phase of the program consists of exchanges of data between pairs of processors, or
- all communication involving a particular node is made up of remote procedure calls following a request-response pattern.

Using this information, we can design an efficient protocol tailored to the application. In the limiting case, the communication pattern of one phase of an algorithm may be completely known in advance.

Although it is profitable to design a protocol for a *class* of applications, we go beyond this to advocate specializing a protocol to a *specific application program.* This allows us to take advantage of any information that is available concerning the specific application's communication behavior. For example, we may know something about the communication behavior of fluid flow simulations in general, but we will have much more detailed information about the behavior of a particular simulation. This extra knowledge can be used to improve the communication protocol's performance.

[1] *In-order delivery* means that messages from a given sender to a given receiver are received in the same order in which they were sent.

3.1 Managing the Complexity of Protocol Design

Designing an efficient, specialized communication protocol is a difficult task, and verifying the correctness of that protocol may be even harder. Clearly, we cannot expect the application programmer to design protocols on a program-by-program basis. To make this approach viable, we must automate the protocol design process.

A *protocol compiler* is a program that generates application-specific protocols. A protocol compiler takes as input a description of an application's communication behavior, and outputs a program that implements a communication protocol specialized for the application. If the programmer is using a high-level, parallel programming language, the protocol compiler can be integrated into the parallel language compiler — the language compiler may generate the communication description for the protocol compiler, or we may view protocol generation as part of the code generation task.

3.2 The Role of Runtime Compilation

In some cases, detailed information about an application's communication behavior is not available at compile time, but becomes known at runtime. In this case, we can wait until runtime, when detailed knowledge is available, then run the protocol compiler and dynamically add the generated code to the program. This is an example of *runtime compilation*, a technique that has proven especially fruitful in parallel computing [Saltz et al. 90].

Consider a parallel program that simulates electronic circuits. When the simulator is compiled, the compiler has no knowledge of what circuits might be simulated. But once a simulation run has begun, the structure of the simulated circuit is known, so detailed information about the simulator's communication pattern is available. By using a protocol compiler at runtime, we can take advantage of this information by generating efficient code for the communication in the main simulation loop.

4 A Simple Case: Straight-Line Systolic Programs

Generating application-specific protocols is a difficult problem, so we will start by attacking a simple case, that of straight-line programs with fully specified communication. A *straight-line* program is a parallel program without any communication enclosed in control flow statements; each processor simply executes its program from beginning to end. *Fully specified* communication means that the program contains `send` and `receive` calls, the destination of every `send` is known in advance, and the types of all messages (if the system uses typed messages) are known in advance.

This case is clearly simpler than any real application, but it provides a reasonable starting point. Once we have solved this case, we will consider how to generalize to more realistic cases.

For maximum performance, we will generate protocols that use *systolic communication* [Kung 88b] whenever possible. In systolic communication, the communication network interfaces to the CPU with a pair of FIFO buffers (one incoming, one outgoing) that are mapped into the CPU's memory. The CPU sends and receives data with ordinary store and load instructions. Often, the CPU can consume a message directly without putting it into memory, or can construct a message directly in the output FIFO rather than marshaling the message in memory. Systolic communication thus can be faster than memory-based communication, because it consumes less memory bandwidth. Systolic communication also implies that arriving messages do not cause an interrupt. This gives an additional speed advantage, since interrupts require time to save and restore processor state, in addition to deciding how to respond to the interrupt.

Systolic communication is *not* limited to the same algorithms as systolic arrays. Systolic arrays are hardware designs that execute the same nearest-neighbor-only communication pattern repeatedly. Systolic communication may implement complex communication patterns, as demonstrated by the Warp [Annaratone et al. 87] and iWarp [Borkar et al. 88, Borkar et al. 90] machines.

4.1 Avoiding Deadlock in Systolic Programs

Deadlock avoidance is the most difficult problem in designing systolic protocols. The routing network is guaranteed to deliver messages to their destinations, provided that arriving messages are removed promptly from the network. This guarantee doesn't necessarily hold for systolic programs, since they may let a message sit in the network until the receiver is ready to receive it. The problem arises when one node is waiting to receive a message, which is blocked by another message whose receiver is not ready for it yet.

In order to avoid deadlock, the systolic program must ensure that whenever communication is taking place, the program will eventually remove a message from the network. This must be true in spite of the fact that only a subset of the messages in transmission may reach their destinations before being blocked.

4.1.1 Deadlock Avoidance Algorithms

The difficulty of avoiding deadlock depends on the particular program and communication topology in question. If the topology is a cross-bar (all pairs of nodes are directly connected), we can check for deadlock using the "crossing-off" procedure of Kung [Kung 88a]. Kung's procedure also works if the network is sufficiently well-connected to realize the program's communication pattern without ever blocking one message behind another [2]. In general, though, deadlock avoidance is difficult.

A deadlock avoidance algorithm must be aware of the network's topology, since that determines which messages can block behind which others. (In this paper, the term "topology" is used to encompass the combination of a physical topology and a routing algorithm.) Programs with systolic communication fall into three classes:

1. *Programs that deadlock on all topologies:* These can be detected by Kung's crossing-off procedure.
2. *Programs that never deadlock on any topology:* These programs are rare, and are mostly trivial cases.
3. *Programs that deadlock on some topologies but not on others:* These programs comprise the vast majority of interesting cases. On topologies where these programs deadlock, they do so in a timing-dependent fashion — they may or may not deadlock depending on details of scheduling and event timing.

Since topology-dependent deadlocks are so prevalent, we need an algorithm to detect them.

We have devised such an algorithm, although space does not allow explaining it in detail here. Given a straight-line systolic program and a topology, the algorithm determines, in polynomial time, whether the program is prone to deadlock on the topology. If the program is deadlock-prone, the algorithm also produces an equivalent systolic program that won't deadlock on the given topology.

A deadlock-prone program is "repaired" by inserting extra *synchronization messages.* These are messages that carry no data, but merely inform the receiving node that it is safe to proceed with its program. We avoid deadlock by waiting for synchronization messages before certain other messages are sent. This strategy of delaying "dangerous" resource demands is used for preventing deadlock in other contexts. For example, the banker's algorithm [Dijkstra 68, Habermann 69] avoids deadlock by preventing a lock acquisition from taking place until it is safe to do so.

[2] Actually, Kung's procedure is able to avoid deadlock for all programs, if we assume that all network routers contain general-purpose processors. Kung presents an algorithm that runs on each router; the algorithm delays routing some messages based on data in the message headers, and on a set of tables generated at compile time and downloaded into the routers. It seems unlikely that Kung's routing algorithm could be implemented efficiently in hardware.

4.2 Buffering

Buffering is another issue that must be dealt with. Standard message-passing semantics imply that data is buffered by the message-passing system. Programmers often take advantage of this; for example, an exchange of messages between two nodes may be expressed as both processors sending, then receiving. This example cannot execute without buffering.

Since systolic programs do not use buffering, we must devise some strategy for dealing with programs that require buffering. Rather than outlawing application programs that require buffering, we can recognize cases where buffering is needed, and insert the necessary buffering.

Given a straight-line program, there is an algorithm that identifies where buffering is needed. The algorithm builds a data structure called the *program graph,* and searches for cycles in it. Each cycle must be broken by buffering a message; buffering a message allows that message's data to arrive at the receiver at one point in time, but not be visible to the receiving program until later. The problem of *optimally* inserting buffering into a program is probably NP-complete, but we can use heuristics to generate good solutions.

A systolic program that deadlocks on *all* topologies can be transformed into a program with topology-dependent deadlocks by inserting buffering. We can then use the algorithm of section 4.1.1 to generate an equivalent deadlock-free program. By combining our deadlock-avoidance algorithm with our algorithm for inserting buffering, we can generate deadlock-free systolic code for *any* straight-line program.

5 Extensions to More General Cases

We have solved the simple case of straight-line programs with fully specified communication, but we haven't dealt with programs outside this narrow domain. We would like to be able to extend our techniques to deal with a wider range of programs, including some real applications. There are two main ways we could extend the model: we could handle programs with more general control flow, and we could handle cases where the compiler has imperfect information about the program's communication pattern.

5.1 Programs with Data-Parallel Control Flow

So far we have assumed straight-line programs, i.e. no control flow. It is difficult to handle general control flow, but we can deal with programs with data-parallel control flow. Data-parallel programs are common in scientific applications on message-passing machines [Fox 88, Hillis & Steele Jr. 86]; such programs are also generated by compilers for data-parallel languages like Fortran-90 [ANS 89, Chatterjee 91], C* [Rose & Steele Jr. 87, Hatcher et al. 91], and MPL [Mas 91].

Programs with data-parallel control flow are structured programs that can be decomposed into a series of *blocks* surrounded by sequential control structures, such that:

- no block contains a control-flow construct with a message-passing operation nested inside it, and
- there is an implied barrier synchronization at the end of each block, and
- all processors are in the same block at all times, and
- messages do not persist across block boundaries.

Because of the simple structure of these programs, we can treat each block as a straight-line program, provided we can generate code to make transitions between blocks.

To handle these transitions, we construct a *block graph* that summarizes the possible control flow between blocks. The block graph tells us which blocks might follow each other block in the execution. We can analyze each potential transition between blocks by using our deadlock avoidance algorithm to decide whether a deadlock is possible at the block transition; if such a deadlock is possible, we must insert synchronization messages at the block boundary to prevent the deadlock.

5.2 Programs with Imperfectly Specified Communication

We would also like to extend our methodology to deal with cases where the communication pattern isn't fully specified in advance. For example, we might not be able to determine in advance the destination or type of all messages.

Handling imperfectly specified communication is a difficult problem in general. If there is *no* information available about the application's communication behavior, our only choice is to run a general-purpose protocol. On the other hand, if *some* information about the application's communication is available, we can probably do better than a general purpose protocol. Devising a general mechanism for this case is an important area for future work.

We note, though, that it is possible to switch communication protocols "on the fly" while the application is running. For example, we might run a fast, specialized protocol in the inner loop of the application, and switch to a general-purpose protocol for other parts of the program.

6 Related Work

An *active message*, as introduced in [von Eicken et al. 92], is essentially an interprocessor interrupt that also carries a data packet. When an active message arrives at its destination, a message handler is run "to integrate the message into the ongoing computation" on that node. Message handlers run at user level, and are intended to be generated by a compiler or a systems programmer. An implementation of active messages on the nCUBE/2 is presented; this implementation achieves a communication latency of 24 microseconds, not including handler execution time.

The nCUBE/2 implementation of active messages is able to achieve low latency because it presents an interface essentially identical to what the hardware provides. The active messages implementation passes the hardware's message-arrival interrupts on to the application program with a minimum of interference. The active messages implementation does not provide buffering or flow control; these functions are left to the user-level program (and its message handlers). Because message handlers run at interrupt time, they must obey strict rules; in particular, a message handler may never block.

It is unclear whether a similarly efficient implementation of active messages is possible on a system with paged virtual memory. The problem arises when an

active message arrives at a node, and the message handler needs to access a page that is not resident in physical memory [3]. If a handler has to wait for a page to arrive from backing store, the active messages implementation must put the handler's message in a buffer, so that other messages can be received. These other messages might also require pages from backing store, so they too may need to be put in buffers. This introduces a buffer management problem, which can be solved either by letting messages back up in the network (which is likely to lead to deadlock), or by discarding messages (causing an unrecoverable error in the application program), or by implementing higher-level flow control in software. This higher-level flow control would necessarily be part of the implementation of active messages, and thereby invisible to the user-level program.

These problems arise because the communication network is a privileged resource, which must be protected by the operating system. By contrast, if the approach of section 2 is followed, the (user-level) communication network can be treated as an ordinary resource like a CPU register — the application can use or abuse these resources in any way whatsoever, without affecting other applications or the operating system.

7 Conclusions

This paper has argued that application-specific communication protocols are an important tool for reducing the cost of communication in parallel programs. General-purpose protocols expend time and memory to protect the system from error or deadlock in the face of arbitrary application behavior. By taking advantage of knowledge about a particular application's communication behavior, we can design a faster protocol, customized for that application.

Before we can use application-specific protocols, we must provide safe, user-level communication. This can be achieved using a known set of architectural and operating system structures. User-level communication allows each application to manage communication in its own way, thus eliminating the semantic gap between application needs and the functionality provided by general-purpose communication systems.

[3]Note that we cannot simply outlaw the problem by locking all pages that might be accessed by handlers into physical memory — this would often require locking most or all of an application's data into memory. Indeed, in the matrix multiply example given in the active messages paper, the entire data set of every node is touched by the message handlers. There is not much point in having paged virtual memory if applications routinely lock themselves into physical memory.

We have introduced the notion of a *protocol compiler*, an automated tool that generates application-specific protocols. Given a description of an application's communication behavior, the protocol compiler generates an optimized communication protocol that is guaranteed to be correct and deadlock-free for that application. We have sketched the design of a protocol compiler that generates highly efficient systolic communications for a subset of data-parallel applications.

8 Acknowledgments

The author is grateful to Ed Lazowska, Rik Littlefield, Richard Ladner, Chandu Thekkath, and Martin Tompa for useful discussions and comments.

References

[Anderson et al. 91] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, 1991.

[Annaratone et al. 87] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Minzilcioglu, and J. Webb. The Warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, Dec. 1987.

[ANS 89] American National Standards Institute. *American National Standard for Information Systems Programming Language Fortran: S8(X3.9-198x)*, March 1989.

[Arnould et al. 89] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1989.

[Borkar et al. 88] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339, 1988.

[Borkar et al. 90] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore,

W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iWarp. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 70–81, 1990.

[Chatterjee 91] S. Chatterjee. Compiling data-parallel programs for efficient execution on shared-memory multiprocessors. Technical Report CMU-CS-91-189, Carnegie Mellon University, Oct. 1991.

[Dally & Seitz 87] W. J. Dally and C. L. Seitz. Deadlock free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[Dally 90a] W. J. Dally. The J-machine system. In P. Winston and S. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, volume 1. MIT Press, 1990.

[Dally 90b] W. J. Dally. Virtual-channel flow control. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 60–68, 1990.

[Dijkstra 68] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, London, 1968.

[Fox 88] G. C. Fox. What have we learnt from using real parallel machines to solve real problems. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 897–955, 1988.

[Geist et al. 91] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A users' guide to PICL, a portable instrumented communication library. Technical Report TM-11616, Oak Ridge National Laboratory, 1991.

[Habermann 69] A. N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7):373–377, July 1969.

[Hatcher et al. 91] P. J. Hatcher, A. J. Lapadula, R. R. Jones, M. J. Quinn, and R. J. Anderson. A production-quality C* compiler for hypercube multicomputers. In *Proceedings of SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 1991.

[Hillis & Steele Jr. 86] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, Dec. 1986.

[Int] Intel Supercomputer Systems Division. *iPSC/860 User's Guide.*

[Int 91a] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview*, 1991.

[Int 91b] Intel Supercomputer Systems Division. *A Touchstone DELTA System Description*, Feb. 1991.

[Kung 88a] H. T. Kung. Deadlock avoidance for systolic communication. In *Proceedings of 15th International Symposium on Computer Architecture*, pages 252–260, 1988.

[Kung 88b] H. T. Kung. Systolic communication. In *Proceedings of Intl. Conference on Systolic Arrays*, pages 695–703, 1988.

[Mas 91] MasPar Computer Company. *MasPar Parallel Application Language (MPL) Users Guide*, March 1991.

[Ousterhout 90] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[Rodeheffer & Schroeder 91] T. L. Rodeheffer and M. D. Schröeder. Automatic reconfiguration in Autonet. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 183–197, 1991.

[Rose & Steele Jr. 87] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, 1987.

[Saltz et al. 90] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. Technical Report 90-59, ICASE, Sept. 1990.

[Thi 91] Thinking Machines Corporation. *CM-5 Technical Summary*, 1991.

[von Eicken et al. 92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. To appear in Proceedings of 19th Intl. Symposium on Computer Architecture., 1992.