An Architecture for Optimal All-to-All Personalized Communication

Susan Hinrichs, Corey Kosak, David R. O'Hallaron, Thomas M. Stricker, and Riichiro Take[‡] [*]
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891, USA

{shinrich,kosak,droh,tomstr,take}@cs.cmu.edu

Abstract

In all-to-all personalized communication (AAPC), every node of a parallel system sends a potentially unique packet to every other node. AAPC is an important primitive operation for modern parallel compilers, since it is used to redistribute data structures during parallel computations. As an extremely dense communication pattern, AAPC causes congestion in many types of networks and therefore executes very poorly on general purpose, asynchronous message passing routers.

We present and evaluate a network architecture that executes all-to-all communication optimally on a two-dimensional torus. The router combines optimal partitions of the AAPC step with a self-synchronizing switching mechanism integrated into a conventional wormhole router. Optimality is achieved by routing along shortest paths while fully utilizing all links. A simple hardware addition for synchronized message switching can guarantee optimal AAPC routing in many existing network architectures.

The flexible communication agent of the iWarp VLSI component allowed us to implement an efficient prototype for the evaluation of the hardware complexity as well as possible software overheads. The measured performance on an 8×8 torus exceeded 2 GigaBytes/sec or 80% of the limit set by the raw speed of the interconnects. We make a quantitative comparison of the AAPC router with a conventional message passing system. The potential gain of such a router for larger parallel programs is illustrated with the example of a two-dimensional Fast Fourier Transform.

1 Introduction

The all-to-all personalized communication (AAPC) step is frequently encountered in parallel programs. In an AAPC step, each processor sends a block of data to every other processor, and every block of data can potentially contain different information. The AAPC step occurs in multi-dimensional convolutions and in array transposes where only one dimension of the array is distributed. Recent implementations of data parallel compilers for High Performance Fortran[10] include directives for general *block-cyclic* array distribution. Changing the distribution of an array often results in a communication where all processors or nearly all processors exchange unique blocks of data[21]. The compiler can often detect when an AAPC step is required, so compile time recognition of AAPC is a reasonable assumption[11].

[*]Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152,and in part by the Air Force Office of Scientific Research under Contract F49620-92-J-0131.

[‡]Author's current address: Riichiro Take, Fujitsu Laboratories Ltd., 1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan. email: riro@flab.fujitsu.co.jp.

Since AAPC steps are so prevalent, many algorithms have been developed to perform AAPC efficiently, though implementations and performance numbers for these algorithms are hard to find. Most algorithms have concentrated on machines with a hypercube topology[16, 28, 26, 4]. More recent work has explored machines with a k-ary n-cube topology. In [28] Varvarigos and Bertekas propose a store and forward algorithm. Theoretically, this algorithm optimally uses network bandwidth. However, to utilize all network bandwidth, each node must be able to source and sink four messages simultaneously, i.e. have twice the memory bandwidth as incoming network bandwidth. Horie and Hayashi[13] and Scott[19] have proposed algorithms that directly send blocks of data to their destinations. These messages are partitioned into contention-free phases. If these phases are separated by global synchronization or some other method, this approach also optimally uses the network bandwidth. Bokhari and Berryman[5] present a hybrid approach for a two-dimensional mesh that does not optimally use the network bandwidth but requires fewer message start ups. This approach requires additional buffer allocation and address calculation on the intermediate nodes.

A number of recent parallel machines have been built with general purpose processors and a routing backplane. The topologies of these machines differ. Thinking Machine's CM-5 architecture relies on a fat tree topology, sends short messages and uses randomized routing to deal with congestion[17]. The Intel Paragon architecture uses a fast two-dimensional mesh connection and routes long messages with a basic routing scheme[15]. Similarly, the Cray T3D uses fast interconnects linked to a three-dimensional torus and a virtual channel wormhole router[1]. The Fujitsu AP-1000 system uses a two-dimensional torus and has a large, structured buffer pool that provides the mechanisms for virtual channel, wormhole routing and for special routers like broadcast and AAPC. The IBM SP-1 uses an Omega-like, multistage switch with flexible but static routing[20].

In spite of the differing interconnect topologies and technologies, the communication architectures of these machines are quite similar. They all use wormhole routing to keep the per-hop hardware latency low. While some of these machines have special support for broadcast communication, none of these machines have support specifically for AAPC.

In [27], Take, Noguchi, and Yokota propose a *dragon switch* for a multistage network that performs all communication as AAPC steps. The dragon switch is reconfigured for the next message when the tail of the current message has passed. This reconfiguration can be safely performed with only local information by relying on the AAPC structure.

We have studied the performance of various AAPC algorithms on iWarp[6, 7], a distributed memory computer connected by a k-ary 2-cube or torus topology. iWarp's communication architecture includes many interesting features included or planned for other systems, including program control of routing, block DMA transfer, and low message overhead.

To be optimal, the AAPC phases proposed in [13] and [19] must be carefully separated to preserve the contention-free schedule. In this paper, we introduce a *synchronizing switch* for torus networks based on the ideas of the dragon switch for multistage networks described in [27]. The synchronizing switch uses the structure of the AAPC phases to perform this phase separation more efficiently and scalably than globally synchronous methods.

We show how the synchronizing switch can be incorporated into a standard wormhole routing communication architecture. With a small addition to the routing hardware, many of today's distributed memory machines can support optimal AAPC.

We implemented and evaluated a prototype of this AAPC architecture on iWarp. In this paper, we present our design and evaluation of this synchronizing switch. We compare the synchronizing switch with a message passing library that uses similar features of the communication architecture.

In Section 2, we describe the decomposition of of AAPC into phases for the torus topology, the basic switching mechanism, and we argue why a simple, local synchronizing switch primitive can guarantee synchronization during the execution of our AAPC algorithm. Section 3 describes an opposing approach that uses a standard message passing library and relies on a randomized message schedule. Our experimental results are given in Section 4. Finally, we present our conclusions in Section 5.

2 The phased AAPC architecture

Since the AAPC step is communication-limited, one can calculate an upper limit on performance by looking at network limitations. The performance estimates in this section assume a $n \times n$ torus connected machine with a system clock of S MHz that can transmit a 4 byte word in T_t cycles. We also assume that all processors exchange messages of B words. This system has $4n^2$ links that can each transmit one word every T_t cycles. In the case of best performance, all messages follow a shortest path and all physical links are busy, so on average each message will cross $n/2$ physical links. The words that make up the n^4 messages of length B will traverse $n^5B/2$ physical links during the AAPC, and all $4n^2$ links of the torus can be used simultaneously. Thus, the best case time to completion is $\frac{(n^5B/2)T_t}{4n^2} = (n^3BT_t)/8$ cycles. The peak aggregate bandwidth is then

$$Agg_{system} = \frac{S \times \text{Total bytes sent}}{\frac{n^3BT_t}{8}} = \frac{4Bn^4S}{\frac{n^3BT_t}{8}} = \frac{32nS}{T_t} \qquad (1)$$

In this section, we present an AAPC schedule and mechanism that approaches this physical limit.

2.1 Optimal AAPC message routes and schedules

In this section, we briefly describe a set of *phases* that achieve optimal AAPC on arrays with unidirectional or bidirectional links.[1] See [12] for an expanded description of these phases. To simplify presentation, we first describe AAPC patterns on a one-dimensional ring with unidirectional links. We then show how to extend these patterns to a two-dimensional torus.[2] Finally we present the extension from unidirectional to bidirectional links. Along the way we introduce a notation that permits the concise description and manipulation of the patterns.

Throughout this section, we use the following terminology. A *message* is a communication transmitted from a source to a destination. A *pattern* is a link-disjoint set of messages. If a pattern is a a communications step in an optimal AAPC, we sometimes refer to it as a *phase*.

First we calculate a lower bound on the number of phases needed. Assume a d-dimensional array with n nodes in each dimension. Since every node needs to send a message to every node (including itself), $(n^d)^2$ messages must be sent. Because there are not enough links to send all of the messages simultaneously, we partition the set of messages into phases. The lower bound on the number of phases needed follows from the bisectional bandwidth:

$$\frac{\textit{number of messages between bisections}}{\textit{number of links between bisections}} = \frac{2 \times (n^d/2)^2}{2 \times n^{d-1}} = \frac{n^{d+1}}{4} \qquad (2)$$

In a bidirectional array there are twice as many links between bisections; in this case the lower bound is $n^{d+1}/8$.

To achieve this lower bound, we begin with the construction of AAPC phases on a one-dimensional ring. Our phases conform to the following constraints, which are sufficient to guarantee optimality:

1. Every possible message appears exactly once in the phases.
2. Each of these messages follows a shortest route to its destination.
3. Every link in the array is used exactly once per phase–that is, there is no contention for links, and no link is ever idle.

There is more than one possible set of phases that conforms to these constraints. The set of phases that we present conforms to the following additional constraints:

4. Because there are $n^2/4$ phases, it follows that we must send *on average* four messages per phase. The phases we present send *exactly* four messages per phase.
5. To limit memory bandwidth requirements, we require that each node send and receive at most one message per phase.

Thus each phase in our set is a *chain* of four messages, in which the destination node of one message serves as the source node for the next, and so on. It is helpful to consider all the communications that must take place; we start by concentrating on the clockwise direction only. Each node must send a message to the node that is: 0 hops away (send-to-self), 1 hop away, ..., $n/2$ hops away. We chain short messages to long ones in the following way: 0 hop messages to $n/2$ hop messages, 1 hop messages to $(n/2) - 1$ hop messages, and so on. Since the total length of any such pair is $n/2$, it will reach halfway around the ring. If we chain two such pairs we will create a pattern composed of four messages that spans the ring, using all the links. This is essentially how all of the phases are constructed. The left side of Figure 1 depicts all the clockwise phases for $n = 8$.

An interesting property of the phases is that given any one of the messages it contains, it is straightforward to infer the other three. Furthermore, each phase has exactly one message that starts and ends inside the first half of the ring. We use the notation (*source,destination*) to refer to the phase which contains that message. The phases in Figure 1 have been labeled using this notation.

A similar set of messages must travel in the reverse direction. The counterclockwise phases are obtained by simply reversing the direction of the messages in the clockwise phases; these are depicted

[1] A unidirectional link connecting node a to node b permits communication either from a to b, or from b to a, but not in both directions simultaneously.

[2] For the sake of simplicity, our discussion is limited to arrays where the number of nodes in each dimension is the same, and a multiple of four (in the unidirectional case) or eight (in the bidirectional case).

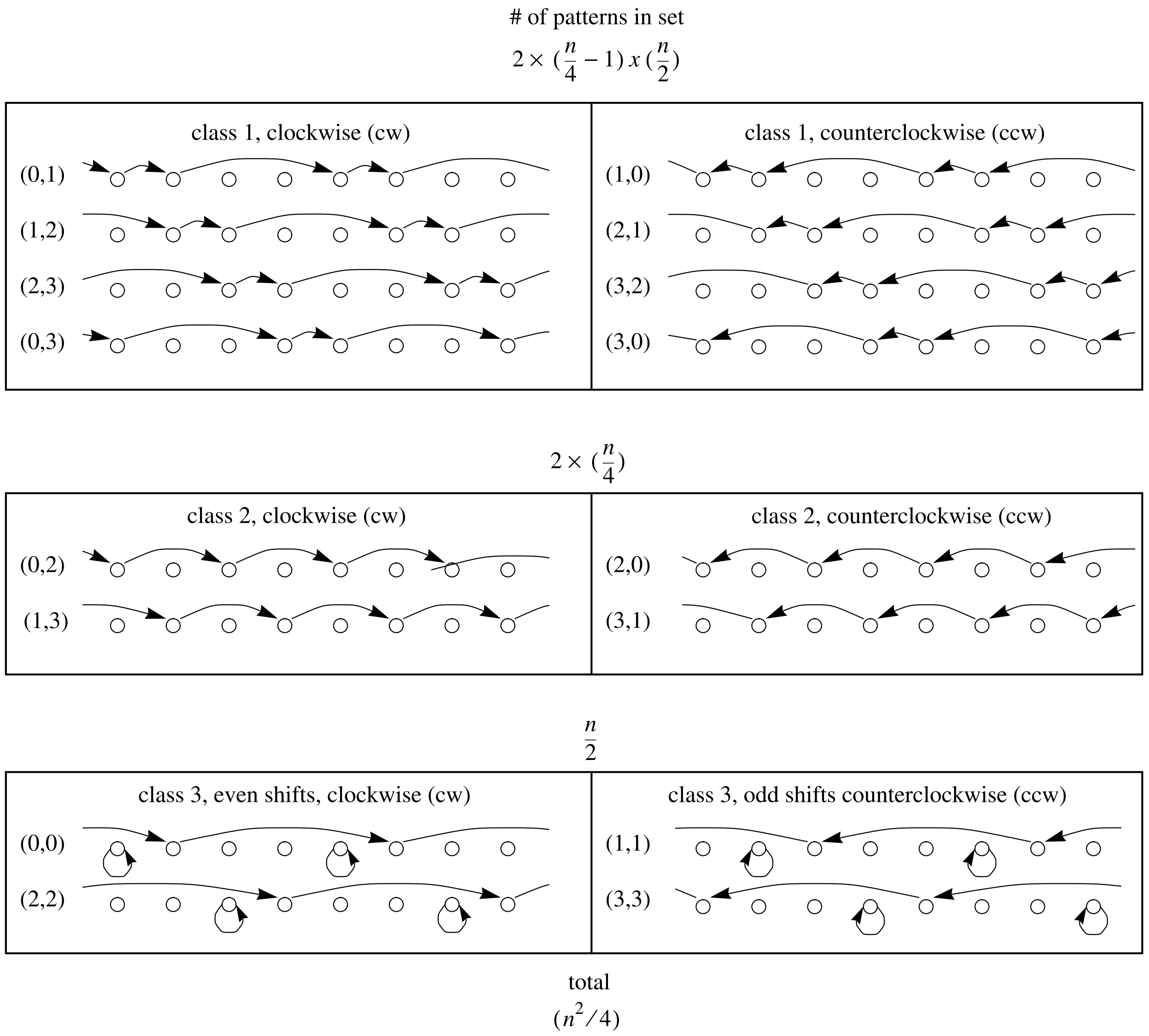


Figure 1: Here we show all one-dimensional phases for $n = 8$.

on the right side of the same figure.[3] The total number of phases is $n^2/4$, which matches the lower bound computed in Equation 2. Since these phases conform to the other constraints as well, they enable an optimal AAPC.

We now show how to extend these phases to achieve optimal AAPC on a two-dimensional $n \times n$ torus with unidirectional links, achieving the lower bound of $n^3/4$ phases.

Every message on the torus can be routed as a horizontal motion followed by a vertical motion. We define the *cross product* of two one-dimensional messages u and v as a two-dimensional message that takes its horizontal motion from u and its vertical motion from v, and we write this as $u \times v$. We define the cross product of two one-dimensional patterns p and q as a two-dimensional pattern composed of cross products of all possible pairs of messages from p and q. The bold arrows in Figure 2 depict the cross product operation on two messages, whereas the figure in its entirety depicts the cross product operation on two patterns.

As can be seen from Figure 2, forming the cross product of two one-dimensional phases results in a two-dimensional pattern that saturates four rows and four columns of the torus. When $n > 4$, this pattern leaves some links idle and is therefore not an optimal AAPC phase. In these cases we must *overlay* multiple two-dimensional patterns which saturate disjoint row and column sets in order to create a phase that saturates all of the rows and columns.

[3] Special care must be taken in the construction of the class 3 phases that chain 0 hop to $n/2$ hop messages: 1. Since the 0 hop messages traverse no links, chaining them in the usual way would cause a node to be the source for two different messages, violating constraint 5. 2. Since all possible messages of sizes 0 and $n/2$ are contained in the clockwise phases, there is no need for the corresponding counterclockwise phases. However, symmetry suggests that there ought to be an equal number of phases in each direction, so the direction of half of these phases is reversed. 3. All of these phases that communicate in the same direction must be node-disjoint; this will be useful in the extension to two dimensions. The bottom of Figure 1 illustrates these phases.

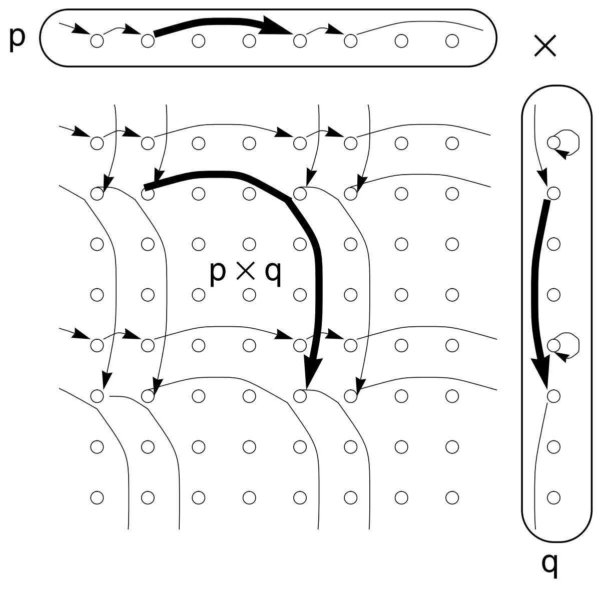


Figure 2: The graphical representation illustrates the algebraic *cross product* operation. The bold arrows depict the cross product of two one-dimensional messages. The figure in its entirety depicts the cross product of two one-dimensional patterns.

The number of patterns that must be simultaneously overlaid is $n/4$.

Here we introduce some notation in order to more conveniently describe the two-dimensional phases. An optimal two-dimensional phase is formed by taking the *dot product* of two ordered tuples, $M_a = (p_0, \ldots, p_{(n/4)-1})$ and $M_b = (q_0, \ldots, q_{(n/4)-1})$, where each of the p_i or q_i is a clockwise one-dimensional phase. We define the dot product $M_a \cdot M_b$ as overlaying $p_0 \times q_0, \ldots, p_{(n/4)-1} \times q_{(n/4)-1}$. We define $\overline{p_i}$ as the counterclockwise pattern which corresponds to p_i, and $\overline{M_a}$ as $(\overline{p_0}, \ldots, \overline{p_{(n/4)-1}})$. The M tuples must conform to the following two constraints:

1. All the one-dimensional phases in a given M must be node-disjoint.
2. Every clockwise one-dimensional phase must appear in exactly one M.

There is a simple algorithm for creating the M tuples. If we think of each node in the upper half of the ring as a player in an arbitrary two-player game (e.g. chess), then each clockwise one-dimensional pattern (a, b) (with $a < b$) can be thought of as a game involving players a and b. If c and d $(c < d)$ are distinct from a and b, we can schedule their game (i.e. pattern (c, d)) simultaneously. Building the M tuples then becomes simply an instance of the well-known tournament scheduling algorithm. Here is one possible tournament schedule for $n = 8$: $M_1 = ((0, 1), (2, 3))$, $M_2 = ((0, 2), (1, 3))$, $M_3 = ((0, 3), (1, 2))$. This schedule contains all the clockwise phases except the (a, a) phases–those involving send-to-self communication. Since these were deliberately constructed to be node-disjoint, they can all be scheduled together. Thus the schedule must include one more entry: $M_0 = ((0, 0), (2, 2))$. In general, the number of M tuples will be $n/2$.

We define a rotate operator $\mathbf{r}$ such that if $M_a = (p_0, p_1 \ldots p_{(n/4)-1})$, then $\mathbf{r}(M_a) = (p_1, \ldots, p_{(n/4)-1}, p_0)$. Also, $\mathbf{r}^2(M_a) = \mathbf{r}(\mathbf{r}(M_a))$, and so on. We use this operator to ensure that every one-dimensional phase is crossed with every other one-dimensional phase in some two-dimensional phase.

Using this notation, the set of all unidirectional AAPC phases on a $n \times n$ torus can be described as:

$$\left\{ \begin{array}{c} M_i \cdot r^k(M_j), \\ M_i \cdot r^k(\overline{M_j}), \\ \overline{M_i} \cdot r^k(M_j), \\ \overline{M_i} \cdot r^k(\overline{M_j}) \end{array} \right\}$$

for i in $\{0 \ldots (n/2)-1\}$, j in $\{0 \ldots (n/2)-1\}$, and k in $\{0 \ldots (n/4)-1\}$.

Thus the total number of phases is $4 \times (n/2) \times (n/2) \times (n/4) = n^3/4$, which matches the lower bound computed in Equation 2.

It is straightforward to extend these phases to the case of a torus with bidirectional channels. To accomplish this, all we need to do is overlay one unidirectional two-dimensional pattern with another pattern that is node-disjoint, and uses the links in the reverse direction. One such set of phases is:

$$\left\{ \begin{array}{c} M_i \cdot r^k(M_j) + \overline{M_i} \cdot r^{k+1}(\overline{M_j}), \\ M_i \cdot r^k(\overline{M_j}) + \overline{M_i} \cdot r^{k+1}(M_j) \end{array} \right\}$$

where the + operator overlays two phases. In this case the total number of phases is $2 \times (n/2) \times (n/2) \times (n/4) = n^3/8$ which again matches the lower bound.

Since the phased algorithm[4] completes the AAPC in $n^3/8$ steps, it will complete in $(n^3/8)(T_s + T_t B)$ cycles, where T_s is the communication start up time. Thus, the algorithm will achieve the following network aggregate bandwidth

$$Agg_{phase} = \frac{S \times \text{Total bytes sent}}{\frac{n^3}{8}(T_s + T_t B)} = \frac{32nBS}{T_s + T_t B} \qquad (3)$$

As the start up time becomes small compared to the message transfer time, this algorithm's performance approaches the machine's peak network aggregate bandwidth calculated in Equation 1.

2.2 A synchronizing switch for phased AAPC

The phased AAPC algorithm described in Section 2.1 achieves optimal aggregate bandwidth only when the different phases are carefully separated. Messages from different phases may have routes that require the same network resources, and this network contention can destroy the optimal use of network bandwidth. Phase separation can be maintained by globally synchronizing after each phase is completed. However, this adds the overhead of $n^3/8$ synchronizations, and global synchronization requires additional communication resources and/or dedicated hardware mechanisms.

With a *synchronizing switch*, the machine can use the structure of the AAPC phases to synchronize between the phases using only local information.

The switching elements of many supercomputer networks use a routing technique called *wormhole routing*. iWarp's communication agent also uses wormhole routing, and the synchronizing switch relies on the details of this wormhole routing hardware. Before describing the synchronizing switch in detail, we first give an overview of iWarp's wormhole routing hardware.

[4]For the remainder of the paper, we discuss the bidirectional, two-dimensional phased AAPC.

2.2.1 Basic wormhole router

The iWarp component consists of a computation agent, a memory agent, and a communication agent. The most interesting part of the iWarp component is the communication agent, which is responsible for switching data streams between four incoming links, four outgoing links, from and to the local memory, and even directly into the arithmetic units of the local computation agent. The switch associates a small input queue for buffering and flow control logic with every incoming stream.

The core unit of the communication agent is a high performance scheduler that continuously forwards data words queued in the input buffers to either the local computation agent or to the input buffer of a neighboring node. The scheduler operates in parallel on all links forwarding up to 40 MB/s per link or 320 MB/s in total. The parts of the switch relevant to our synchronizing switch implementation are illustrated in Figure 3.

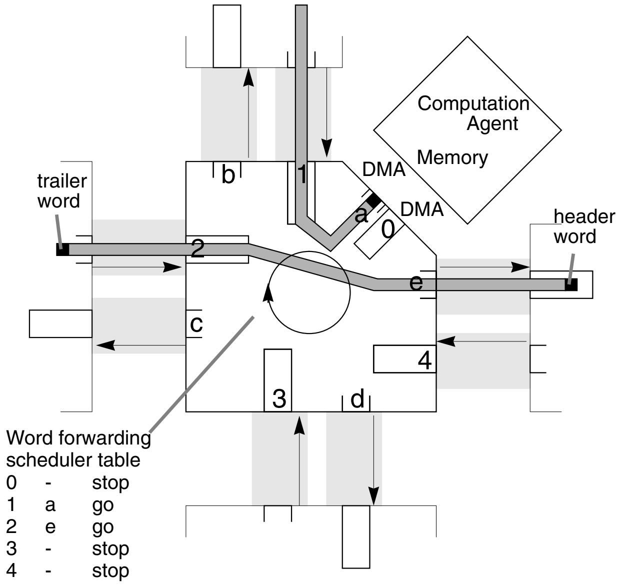


Figure 3: Basic structure of the communication agent of the iWarp VLSI component. Special header and trailer words control the forwarding state of the queues to construct connections without computation agent intervention.

The scheduler works independently of the computation agent and continues to forward data even when the computation agent is stalled or halted. Similarly, the streams from and to local memory can continue to transfer data without intervention from the computation agent after the DMA controllers are set up. The forwarding activity at every node is controlled by state associated with the input queue. The forwarding state of these queues is updated by the communication agent based on detecting specially tagged data items in the data stream.

Special header words are used to establish source defined connections. Based on these header words, the communication agent binds the input queue with a queue on a neighboring node, so the input queue forwards data to the neighboring queue. By default these connections cut straight through a node, but the routing words specify where the route should change direction or where the final connection destination is. In addition to the routing words, a set of *stop conditions* can disable forwarding of an input stream. When

```
1  SetStopCondition(NotInMessage, AAPCinputs)
2  for phase from 0 to num_phases-1 do
3    pattern = ComputePattern(node_id, phase)
4    forall in_queue in Active(pattern) do
5      Forward(in_queue)
6    if SenderAndReceiver(pattern) then
7      SendMessageHeader(Destination(pattern), Route(pattern))
8      ReceiveMessageHeader()
9      StartDMA(DataBlockOut[Destination(pattern)], out_queue)
10     StartDMA(DataBlockIn[Source(pattern)], in_queue)
11     wait for DMAs_complete()
12     SendMessageTrailer()
13     ReceiveMessageTrailer()
14   wait for QueuesComplete(pattern)
```

Figure 4: Pseudo-code that implements the synchronizing switch between AAPC phases on iWarp. The assertion QueuesComplete(pattern) is true whenever all involved queues have signaled NotInMsg status.

the specified condition occurs, the input queue stops forwarding data until the computation agent intervenes.

2.2.2 Synchronizing switch implementation

In the phased AAPC, all physical links leading into a node are used in each phase of the AAPC, either for messages destined for that node or for messages passing through to other nodes. Due to this structure, when a node recognizes that all messages for the current phase have passed over its input links (i.e. the *tails* of all messages have passed), it can safely proceed to the next phase. This assumption is the basis of the synchronizing switch, and the correctness of this assumption is proved in Section 2.2.3.

An outline of the iWarp synchronizing switch is shown in Figure 4. In statement 1, the queues are set to stop forwarding data whenever the link is not in a message, so messages that arrive too early are stalled. Otherwise, a message from phase $i+1$ may pass through a node still sending messages for phase i, destroying the phase synchronization. In statements 4 and 5, the stopped queues are set to forward data until the next time the queues are not in a message. Statements 6 to 13 send and receive data for the nodes actively transferring data from and to their memories in the current phase. All nodes wait at statement 14 until the messages for the current phase have passed through their input queues.

The current best global synchronization implementation for a $n \times n$ iWarp operates in $O(n)$ steps. The synchronizing switch code executes in constant time, but the synchronizing switch must wait for the tails of messages to propagate which is also $O(n)$ steps. However, the global synchronization time does not even start until all the tails of all the messages on the machine have been received, so the local synchronization case overlaps synchronization with the time already needed for tail propagation.

2.2.3 Synchronizing switch correctness

From Section 2.1, we know that the routes of each phase use all links of the machine once and only once. Therefore, one phase uses all of the machine's network bandwidth. If the AAPC phases are separated by global synchronizations, we know that each phase uses the machine's network bandwidth optimally. Given AAPC phases where each node starts in the same phase, a sufficient condition for optimality is

Condition 1 *A message from a later phase will not block progress of a message from an earlier phase.*

By Condition 1, the lowest numbered active phase across the machine is never blocked, so nodes participating in this phase can send and receive data at the network rate. Therefore, Condition 1 is sufficient to ensure optimal performance.

Now we prove that replacing the global synchronization with the synchronizing switch preserves Condition 1 and therefore optimality.

Lemma 1 *In one iteration of the synchronizing switch, exactly one message passes over each input link.*

This is true by construction of the synchronizing switch. The NotInMessage stop condition prevents more than one message from passing over an input link in each phase.

The construction in Section 2.1 shows that exactly one message is sent over each link in each phase, and Lemma 1 says that each node processes exactly one message over each input link in each phase. Since all nodes start in the same AAPC phase, each node must be processing messages from the same phase over its four input links. Since each node processes messages from the phases in order, Condition 1 must hold. Therefore, the synchronizing switch preserves optimality.

2.2.4 Adding support for phased AAPC to other machines

Our iWarp prototype implementation of the synchronizing switch is quite efficient, but this implementation relies on a close interaction between the communication agent and the computation agent, so it is not a practical approach for many of today's distributed memory machines. Specifically, the computation agent explicitly waits for all of the input queues to be NotInMessage before proceeding to the next phase and explicitly forwards the input queues for the next phase. In several other distributed memory systems[17, 15, 1, 14, 20], the communication and computation agents are not as tightly coupled, so the computation agent cannot directly observe and control the input queues.

Changing a traditional wormhole router as described in Section 2.2.1 to support the synchronized switch requires a small change to the switching hardware. An additional constraint must be enforced whenever the assignment of a queue is changed. As an example, consider extending the basic 6×6 switching chip used in the Paragon routing backplane. It supports X+,X-,Y+,Y- links connected as a mesh and two Z+,Z- paths connected to the network interface of the local processor. To support phased AAPC, five input queues must be configured as AAPC queues. For correct execution of the synchronizing switch, these AAPC queues can only change their forwarding assignments whenever all five AAPC queues indicate that they are done forwarding a message. This constraint requires a sticky *NotInMessage* bit for each queue. A simple AND-gate can check whether all queues have been passed by a message and enable the processing of a new message header at this queue. Other distributed memory systems with that use wormhole routing could be extended in a similar fashion (e.g. AP-1000, SP-1).

Figure 5 shows the computation agent pseudo-code for the phased AAPC with the enhanced routing hardware for synchronized switching. This code is a subset of the code needed for our prototype shown in Figure 4. Blocks of data are sent according to the phased AAPC schedule. Unlike the prototype AAPC code, every node sends a (possibly empty) message in every phase. The node must send an empty message to itself if it is not scheduled to exchange data in a particular phase. Otherwise, the synchronizing switch would have to determine whether the node needs to inject a message in each phase, which would require a tighter interaction between the communication and computation agents. With the addition of empty messages, the synchronizing switch can simply inject a message in each phase.

```
for phase from 0 to num_phases-1 do
  pattern = ComputePattern(node_id, phase)
  SendMessageHeader(Destination(pattern), Route(pattern))
  ReceiveMessageHeader()
  if SenderAndReceiver(pattern) then
    StartDMA(DataBlockOut[Destination(pattern)], out_queue)
    StartDMA(DataBlockIn[Source(pattern)], in_queue)
    wait for DMAs_complete()
  SendMessageTrailer()
  ReceiveMessageTrailer()
  wait for QueuesComplete(pattern)
```

Figure 5: Code for sending AAPC messages over an extended AAPC router.

Many routers use multiple pools of buffered queues to support virtual channels (e.g. AP-1000, iWarp). Such routers could adapt one pool of virtual channels for AAPC with synchronized switching and use the other pools for message passing and other communication methods. Once the switch is enhanced for synchronized switching, phased AAPC can easily be incorporated into an existing message passing library.

3 Using a standard message passing system for AAPC

Most distributed memory machines offer message passing communication. Message passing ensures door-to-door delivery, so messages are eventually delivered to the destination without further program action. Common models of message passing quantitatively characterize the message passing system by key parameters like message throughput and processing overhead and suggest that the programmer does not rely on the particular details of the target architecture[8, 2]. Figure 6 shows a simple message passing AAPC program.

```
for i from 0 to NumberOfProcessors-1 do
  NBSendMessage(Destination(i), Route(i), DataBlock[i])
for j from 0 to NumberOfProcessors-1 do
  NBReceiveMessage(DataBlock[j])
```

Figure 6: Message passing AAPC code. We assume NBSendMessage and NBReciveMessages are buffered, non-blocking primitives offered by the message passing library.

Both phased AAPC and message passing AAPC decompose the AAPC pattern into messages and route them through the network. However, in message passing AAPC, a node passes a batch of n messages to the network. The wormhole router then routes these messages to n different processors. Since the network is an independent subsystem with no information about the system-wide communication structure, the router uses a simple, greedy scheduling strategy. Whenever a requested communication link becomes available, a message will proceed.

Routes between nodes in the AAPC phases described in Section 2.1 are the same routes generated by the simple e-cube torus router. By following the communication schedule for the phased AAPC, a basic message passing library should recreate the same

AAPC phases and achieve similar performance. However, not all nodes will finish each phase at the same time even if the messages are the same size, because not all nodes are scheduled to send messages in each phase. On iWarp, the performance of unsynchronized nodes following the phased AAPC schedule and using the basic message passing package was about the same as the performance following a random schedule.

Since message passing libraries do not rely on the communication structure for scheduling or routing, message passing is more adaptable in situations where the communication structure is sparse, changing, or unknown.

3.1 The iWarp message passing system

On iWarp, the basic message passing routers are based on a reverse e-cube scheme[22]. All routes start in the X-direction, eventually turn corners, and continue in the Y-direction. The queues can be assigned to multiple pools and the routers can use date-lines to break circular dependencies, avoiding routing deadlocks. Many other commercial, distributed memory systems use similar simple e-cube or reverse e-cube routing schemes. While most routers are fixed in the communication hardware, the iWarp system enables a variety of virtual channel configurations and permits different router software to determine the precise route of every message before it is sent.

Previous work on mesh routers has shown that the non-adaptive e-cube routers can have severe performance limitations, due to imbalanced congestion. Several advanced, adaptive wormhole routers have been proposed for machines with virtual channels[3], but only few of them have been implemented in hardware. We have tested some of these advanced routers with our iWarp message passing system. Unfortunately the advanced methods delivered only up to 30% of improvement over the basic e-cube scheme, and with only two resource pools, these advanced methods could not use the torus routing capability. Faced with the limitation of only two pools, we chose the torus routing over the randomized, more adaptive router, so our measurements in Section 4 are based on a deterministic e-cube router with the torus routing capability.

To compare a specialized AAPC architecture with a general message passing system, we are most concerned with the issues of routing and data transfer. For this reason, we did not use a standard message passing interface such as PVM[25] or MPI[18]. These standard interfaces have software overheads resulting from error and protection checking in the operating system, buffer management, and standardized synchronization semantics, which result in copying data. These aspects of message passing libraries are an important issue of research, but for this study they would have obscured the architectural insights into the nature of an all-to-all communication. Instead of a standard library, we use the deposit message passing library written for the Fx compiler[24], based on the deposit model[23]. The deposit library allows direct deposit of incoming data to its final destination. At the time it is sent, a message is guaranteed to encounter a receiver that is ready to extract it from the network immediately, minimizing buffering overheads and message congestion at the receiver.

The measured overhead per transferred message is constant at around 400 cycles ($20\mu s$). The network latency is proportional to the network diameter n with a cost of 2 to 4 cycles per hop. Large data blocks are injected as single messages into the network, and once data is flowing, the transfer rate is limited only by the link bandwidth of 40 MBytes/sec. The same maximal transfer rate applies to all of our AAPC implementations.

4 Measurements and performance evaluation

We experimented with several variations of AAPC on an 8×8 iWarp system. Each node of this system runs at 20 MHz (S) and the physical transfer time between two nodes is 2 cycles (T_t). With these parameters, Equation 1 predicts that the peak aggregate bandwidth on this system is 2.56 GB/s.

4.1 Comparison of methods

During our study, we implemented versions of several AAPC algorithms. Figure 7 shows the aggregate bandwidth performance of these algorithms. These measurements were made for AAPC where all messages are the same size.

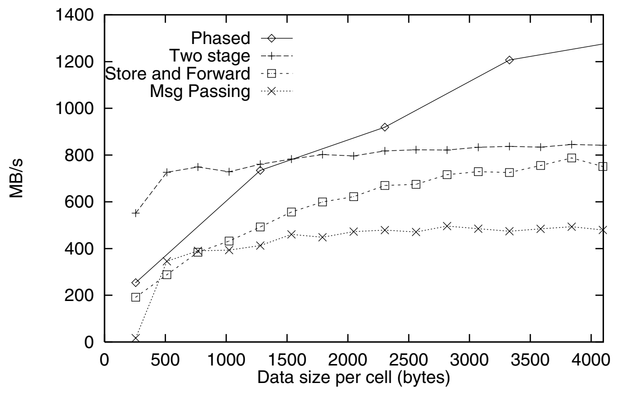


Figure 7: Performance of a variety of AAPC implementations.

We measured the message passing implementation described in Section 3.1. This implementation operates at 500 MB/s, about 20% of optimal.

The performance of the store and forward algorithm described in [28] is constrained by the node's memory bandwidth limitations to half of of the optimal aggregate bandwidth. In fact, our implementation approaches 800 MB/s, about 30% of optimal.

The two stage algorithm is similar in spirit to the hybrid algorithms proposed by Bokhari and Berryman[5], but it is simpler. This algorithm first performs an AAPC over the rows and then an AAPC over the columns. For a $n \times n$ system, the messages will be n times larger, so the startup times are better amortized. However, by only using the rows or the columns simultaneously at most half of the network bandwidth can be utilized. Our measurements show that the two phase algorithm can outperform other methods on small messages but approaches the same performance limit as the single phase store and forward algorithm.

Given our prototype implementations on iWarp, the phased algorithm with the synchronizing switch outperforms other methods for messages greater than 1500 bytes. We expect that adding direct hardware support, as proposed in Section 2.2.1, or even improved machine specific coding can reduce the phased algorithm's overhead significantly and make the phased AAPC more competitive for smaller message sizes.

Figure 8 compares the effect of global and local synchronization on phased AAPC performance for a larger range of message sizes. Due to the lower cost of the synchronized switch, the locally synchronized implementation is more effective over smaller message sizes. As the message size increases, the globally synchronized implementation approaches the same performance.

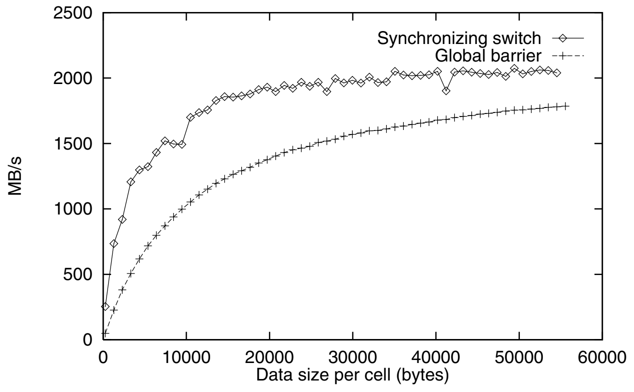


Figure 8: Performance of the phased AAPC algorithm with global synchronization versus local synchronization.

4.2 Message size variation

We also measured the effects of varying message sizes on AAPC performance in two experiments. We compare the performance of the phased AAPC algorithm with the performance of the uninformed message passing. Since the message passing library delays decisions to run time, it should be more adaptable to message size variations.

Figures 9(a) and 9(b) show the results of two probabilistic experiments. In the first experiment, we varied the message size over a range. Given a base message size of B and a variance of V, we chose a new message size for each step and each node over the range of $B - VB$ to $B + VB$ with uniform probability.

Figure 9(a) shows that the phased algorithm performance decreases as the variance increases. The message passing versions are unaffected by the message variation, but for the corresponding block size, the phased algorithm outperforms the message passing algorithm. Even though the message sizes vary, the likelihood of zero length messages is very low.

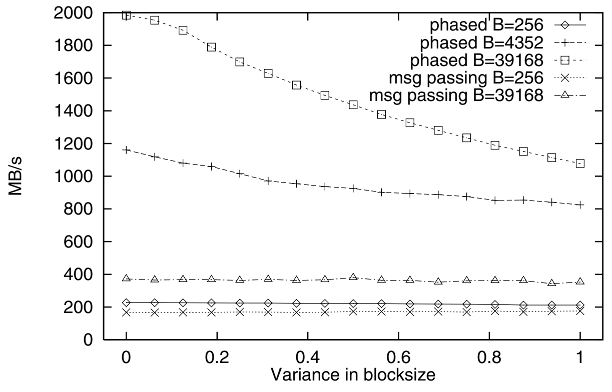
The second experiment explores the effect of zero length messages by probabilistically selecting between a non-zero message length and a zero message length. Given a base message size B and probability P, the new message length is 0 with probability P. Otherwise, it is B.

Figure 9(b) shows that the phased algorithm performance decreases linearly as the probability of the zero length message increases. As in the previous experiment the message passing performance is relatively unaffected by the change in message size. In this case, the phased AAPC algorithm is out performed by the message passing algorithm as P increases.

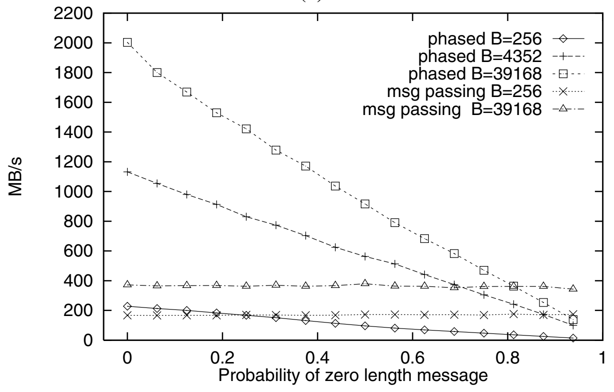
4.3 Common communication steps as AAPC

All communication steps can be executed as subsets of AAPC by inserting empty messages between non-communicating nodes. We measured several common communication steps executed as subsets of AAPC and compared their performance to adaptable message passing versions to evaluate what these communication patterns would lose. We looked at a nearest neighbor communication step, hypercube exchange communication step, and a communication step from a finite element method application described in [9][5]. Table 1 shows the performance of these communication steps.

[5]The performance of the FEM communication step in this paper differs from the performance in [9] because this implementation does not measure the application buffering costs.



(a)



(b)

Figure 9: Performance of phased AAPC algorithm over varying message sizes. In part (a) message sizes vary probabilistically over a range. In part (b) message sizes probabilistically vary between 0 and B. Graphs show the average over 16 different sets of message sizes. The lines are labeled by the base block size sent in bytes.

Pattern	AAPC (MB/s)	Message passing (MB/s)	Factor difference
Nearest neighbor	485	1425	2.9
Hypercube	511	1083	2.1
FEM	84	195	2.3

Table 1: Bandwidth of several common communication patterns implemented as subsets of AAPC and implemented with message passing.

These are all relatively sparse communication patterns. Each node need only communicate with 4 to 15 other nodes. In these implementations, the subset AAPC versions are a factor of 2 to 3 worse than the message passing version. This observation agrees with the second probabilistic experiment. If only 10% of the messages are non-zero, the message passing versions outperformed the AAPC version.

4.4 AAPC communication in 2D FFT

Many parallel programs for medical imaging, radar processing and robot vision rely on two-dimensional fast Fourier transforms (FFTs) for various filtering steps. These are time-critical applications with serious computation requirements. A two-dimensional FFT for a 512×512 video image at 30 frames per second requires roughly 700 MegaFlop/sec.

Most current High Performance Fortran (HPF) compilers generate message passing library calls to execute array transposes and other "dense" communication steps. An HPF compiler exists for iWarp[21], and the communication segment of a two-dimensional FFT generated by this compiler consists of two AAPC steps to execute the array transposes. Here we calculate the impact of improved AAPC performance on this application. The full integration of our new phased AAPC primitive into this compiler is in progress.

In general, the application time will be reduced by $P(F-1)$ % where P is the percentage of the original time spent in communication and F is the factor change of the new communication time. For the 512×512 two-dimensional FFT executed on an 8×8 iWarp, 52% of the time is spent in two AAPC steps that exchange messages of 128 words running in 801,000 cycles. The phased AAPC should execute these two steps in 184,400 cycles, which would be a factor reduction of 0.23 over the original communication time. Thus, the phased AAPC would reduce the 512×512 two-dimensional FFT time by 40%. The message passing program could generate 13 frames/sec, while the program using phased AAPC can generate 21 frames/sec.

Figure 10 shows the performance break down of several two-dimensional FFTs executed on a 8×8 iWarp.

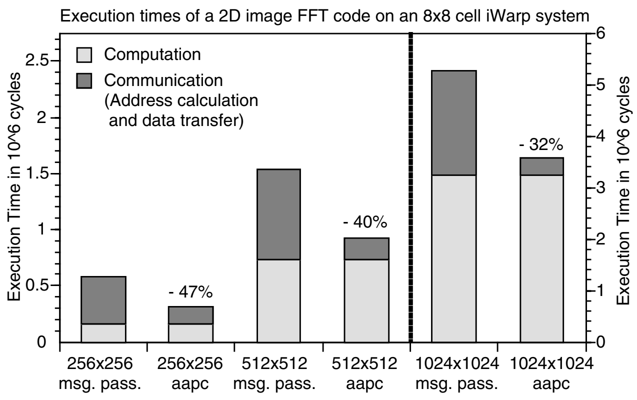


Figure 10: Performance improvements of a two-dimensional FFT code using phased AAPC over a code based on message passing AAPC.

5 Conclusions

In this paper, we have examined a phased AAPC algorithm and the architectural requirements for its efficient execution. Our prototype implementation on iWarp shows that the phased AAPC architecture is practical and efficient. The examination of other popular network architectures showed that our AAPC support requires minimal additional hardware and is compatible with most wormhole routers.

The iWarp implementation reaches an aggregate network bandwidth of over 2 GB/s for balanced AAPC. This is a factor of five faster than a conventional message passing implementation on the same hardware.

The main reason for the performance improvement is the synchronizing switch. For optimal AAPC, the phases must be separated and a contention-free schedule must be maintained. Our synchronizing switch design waits for the tails of all messages to propagate through a node before sending the next phase's messages. It achieves this by enforcing a local constraint rather than invoking a global barrier synchronization operation. Therefore, the locally synchronized switch is more scalable and more efficient than a global barrier, as confirmed by our measurements.

Most of the features required by the synchronizing switch already exist in basic wormhole routers. With a slight modification, the routers of several distributed memory machines can support the synchronizing switch. These hardware additions make the phased AAPC practical for machines like Paragon or the SP-1 where the computation and communication agents are not as tightly coupled as in the iWarp.

One could build a network architecture purely from synchronizing switches, supporting all communication steps as subsets of AAPC. However, our measurements of common communication steps show that while dense communication steps would benefit from phased AAPC, many sparse communication steps would lose a factor of two to three over a conventional message passing architecture. A network architecture optimized for both worlds can configure one set of virtual channels (i.e. a pool) to implement the synchronizing switches for AAPC and leave the remaining sets for message passing or other communication methods. In this case, conventional message passing and phased AAPC communication can co-exist, and the application or compiler can chose the appropriate communication primitive.

Since programmers and compilers can frequently detect AAPC steps, AAPC primitives can be utilized. The primitives execute significantly faster on an architecture for optimized AAPC. The necessary support should be considered in future network designs, and phased AAPC calls should be included into standard message passing libraries.

Acknowledgments

We would like to thank Anja Feldmann and Greg Morrisett for their help on improving this article, Guy Blelloch for his suggestions and Thomas Gross and the rest of the CMU/ Intel iWarp group for their support.

References

[1] ADAMS, D. Cray T3D System Architecture Overview. Tech. rep., Cray Research Inc., September 1993. Revision 1.C.

[2] BAR-NOY, A., AND KIPNIS, S. Designing Broadcating Algorithms in the Postal Model for Message-Passing Systems. In *Symposium on Parallel Algorithms and Architectures* (San Diego, June 1992), ACM, pp. 13–22.

[3] BERMAN, P., GRAVANO, L., PIFARRE, G., AND SANZ, J. Adaptive Deadlock- and Livelock-Free Routing with All Minimal Paths in Torus Networks. In *ACM Symposium on Parallel Algorithms and Architectures* (San Diego, June 1992), ACM, pp. 3–12.

[4] BOKHARI, S. Multiphase complete exchange on a circuit switched hypercube. Tech. Rep. 91-5, ICASE, January 1991.

[5] BOKHARI, S. H., AND BERRYMAN, H. Complete Exchange on a Circuit Switched Mesh. In *Proc. Scalable High Performance Computing Conference* (Williamsburg, VA, April 1992), pp. 300–306.

[6] Borkar, S., et al. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference* (1988), pp. 330–339.

[7] Borkar, S., et al. Supporting Systolic and Memory Communication in iWarp. Tech. Rep. CMU-CS-90-197, Carnegie Mellon University, School of Computer Science, 1990. Revision of a paper that appeared in the 17th Annual Intl. Symposium on Computer Architecture, Seattle, 1990, pp. 70-81.

[8] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., and von Eicken, T. LogP: towards a realistic model of parallel computation. Tech. Rep. UCBC 92-713, Univ. of California, Berkeley, 1992. expanded version of paper in 4th Symp. on PPoPP.

[9] Feldmann, A., Stricker, T., and Warfel, T. Supporting sets of arbitrary connections on iWarp through communication context switches. In *ACM Symposium on Parallel Algorithms and Architectures* (Schloss Velen, Westfalia, Germany, July 1993).

[10] High Performance Fortran Forum. *High Performance Fortran Language Specification Version 1.0.*, May 1993.

[11] Hinrichs, S. Compiler Resource Management for Connection-Based Communication. Internal document, 1994.

[12] Hinrichs, S., Kosak, C., O'Hallaron, D., Stricker, T., and Take, R. An Architecture for Optimal All-to-All Personalized Communication. Tech. Rep. CMU-CS-94-140, Carnegie Mellon University, School of Computer Science, 1994.

[13] Horie, T., and Hayashi, K. All-to-All Personalized Communication on a Wrap-around Mesh. In *Proceedings of CAP Workshop* (Canberra, Austrailia, November 1991).

[14] Horie, T., Ishihata, H., and Ikesaka, M. Design and implementation of an interconnection network for the AP1000. In *Proc. IFIP World Computer Congress* (1992), vol. I, Information Processing, pp. 555–561.

[15] Intel Corp. *Paragon X/PS Product Overview*, March 1991.

[16] Johnsson, S. L., and Ho, C.-T. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Transactions on Computers 38*, 9 (September 1989), 1249–1268.

[17] Leiserson, C., Abuhamdeh, A., Douglas, D., Feynman, C., Ganmukhi, M., Hill, J., Hillis, D., Kuszmaul, B., St.Pierre, M., Wells, D., Wong, M., Yang, S., and Zak, R. The Network Architecture of the Connection Machine CM-5. In *Symposium on Parallel Algorithms and Architectures* (San Diego, June 1992), ACM, pp. 272–285.

[18] The Message Passing Interface Forum. *Draft Document for a Standard Message Passing Interface*, November 1993.

[19] Scott, D. S. Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies. In *The Sixth Distributed Memory Computing Conference Proceedings* (1991), pp. 398–403.

[20] Snir, M. Scalable Parallel Computing - The IBM 9076 Scalable POWERParallel-1. In *ACM Symposium on Parallel Algorithms and Architectures* (June 1993), ACM, p. 42.

[21] Stichnoth, J., O'Hallaron, D., and Gross, T. Generating Communication for Array Statements: Design, Implementation, and Evaluation. *Journal of Parallel and Distributed Computing* (1994). to appear.

[22] Stricker, T. Message Routing on Irregular 2D-Meshes and Tori. In *Proceedings of the 6th Distributed Memory Computing Conference* (Portland, OR, Apr. 1991), pp. 170–177. Also appeared as Technical Report CMU-CS-91-109, Carnegie Mellon School of Computer Science.

[23] Stricker, T., Stichnoth, J., O'Hallaron, D., Hinrichs, S., and Gross, T. Decoupling Communication Services for Compiled Parallel Programs. Tech. Rep. CMU-CS-94-139, Carnegie Mellon University, School of Computer Science, 1994.

[24] Subhlok, J., Stichnoth, J., O'Hallaron, D., and Gross, T. Exploiting Task and Data Parallelism on a Multicomputer. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA, May 1993).

[25] Sunderam, V. S. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience 2*, 4 (December 1990), 315–39.

[26] Take, R. A Routing Method for All-to-all Burst on Hypercube Networks. In *Proc. 35th National Conference of Information Processing Society of Japan* (1987), pp. 151–152. In Japanese.

[27] Take, R., Noguchi, Y., and Yokota, H. An Architecture for Parallel Database Computing. In *Transputing '91* (1991), pp. 1–14.

[28] Varvarigos, E. A., and Bertsekas, D. P. Communication algorithms for isotropic tasks in hypercubes and wraparound meshes. *Parallel Computing 18* (1992), 1233–1257.