# An Architecture for Optimal All-to-All Personalized Communication

Susan Hinrichs, Corey Kosak, David R. O'Hallaron,
Thomas M. Stricker, and Riichiro Take‡

September, 1994

CMU-CS-94-140

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report is an extended version of a paper that appeared in SPAA '94.

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

This research was sponsored in part by the Advanced Research Projects 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.

**Keywords:** All-to-all personalized communication, AAPC, communication architecture, complete exchange, Cray T3D, IBM SP1, iWarp, parallel systems, TMC CM5

## 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 $8 \times 8$ 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[Hig93] 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[SOG94]. The compiler can often detect when an AAPC step is required, so compile time recognition of AAPC is a reasonable assumption[Hin94].

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[JH89, VB92, Tak87, Bok91]. More recent work has explored machines with a k-ary n-cube topology. In [VB92] 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[HH91] and Scott[Sco91] 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[BB92] 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[LAD$^+$92]. The Intel Paragon architecture uses a fast two-dimensional mesh connection and routes long messages with a basic routing scheme[Int91]. Similarly, the Cray T3D uses fast interconnects linked to a three-dimensional torus and a virtual channel wormhole router[Ada93]. 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 SP1 uses an Omega-like, multistage switch with flexible but static routing[Sni93].

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 [TNY91], 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[B$^+$88, B$^+$90], 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 [HH91] and [Sco91] 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 [TNY91]. 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 other approaches to performing AAPC. 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 bandwidth limitations. The performance estimates in this section assume a $n \times n$ torus connected machine and assume that all processors exchange messages of $B$ bytes. This system has $4n^2$ links that can each transmit one $f$ byte flit every $T_t$ microseconds. 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. $n^4$ messages of length $B$ will be exchanged over the system. The flits of all the messages will traverse $n^5B/(2f)$ physical links during the AAPC, and in the best case all $4n^2$ links of the torus can be used simultaneously. Thus, the best case time to completion is $\frac{(n^5B/(2f))T_t}{4n^2} = (n^3BT_t)/(8f)$ microseconds. The peak aggregate bandwidth is then

$$Agg_{system} = \frac{\text{Total bytes sent}}{\text{AAPC Time}} = \frac{Bn^4}{\frac{n^3BT_t}{8f}} = \frac{8fn}{T_t} \tag{1}$$

In this section, we present a message schedule and AAPC mechanism that asymptotically matches the physical performance limit for two-dimensional tori.

## 2.1 Optimal AAPC message routes and schedules

First we present a tutorial overview of the construction of optimal, contention-free phases used in AAPC. Throughout this section, we use the following terminology. A *message* is a block of data transmitted from a source to a destination node. A *pattern* is a link-disjoint set of messages. If a pattern is an optimal communications step in an AAPC, we sometimes refer to it as a *phase*.

We also make the distinction between *unidirectional* and *bidirectional* links. A unidirectional link connecting node $a$ to node $b$ permits communication from $a$ to $b$, or from $b$ to $a$, but not in both directions simultaneously. A bidirectional link permits communication in both directions simultaneously. For ease of presentation, we limit the discussion 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).

### 2.1.1 AAPC on a ring of unidirectional links

We begin with a description of the simplest case: AAPC patterns on a one-dimensional ring of $n = 4i : (i \in \mathcal{N})$ processors connected by unidirectional links.

**Communication requirements** It is helpful to consider all the messages that must be exchanged. First we consider communications in the clockwise direction only.

$$\text{Each node sends a message to the node that is: } \left\{ \begin{array}{l} \text{0 hops away (send-to-self),} \\ \vdots \\ \frac{n}{2} \text{ hops away} \end{array} \right\} \text{ clockwise.}$$

A similar set of communications must take place in the counterclockwise direction as well:

$$\text{Each node sends a message to the node that is: } \left\{ \begin{array}{l} \text{1 hop away,} \\ \vdots \\ (\frac{n}{2} - 1) \text{ hops away} \end{array} \right\} \text{ counterclockwise.}$$

Note that the counterclockwise communications are analogous to the clockwise communications, except for the 0 hop and $(n/2)$ hop messages. Since these messages reach the same destinations regardless of the direction of travel, there is no need to include both clockwise and counterclockwise versions of these messages.

Every necessary communication (i.e. a (*source,destination*) pair) for the AAPC is represented by some message in the above two sets, and every message follows a shortest route to its destination.[1]

The system cannot transmit all of these messages simultaneously, since many of them share the same communication links. At the other extreme, the system could transmit the messages one at a time, but doing so would leave most links idle, clearly not an optimal AAPC. However, since link-disjoint messages can be transmitted simultaneously without conflict, it is natural to try to combine messages in a logical way to form communication *patterns*. Figure 1 shows one possible division of AAPC on a ring of four nodes into contention-free patterns. Our goal is to find a set of patterns that is optimal. (Using our terminology, we will refer to these optimal patterns as *phases*.)

To guarantee optimality, our phases must conform to the following three constraints:

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.[2]

Because network access bandwidth is limited and usually a bottleneck in common machines, our phases also adhere to the following constraint:

4. Each node must send and receive at most one message per phase.

[1]For example, on an 8 node ring, it is possible to reach the same destination either by moving 5 hops in the clockwise direction or 3 hops in the counterclockwise direction. Our messages will always take the shortest route.

[2]When the number of processors in a dimension is not a multiple of 4 (8) for the unidirectional (bidirectional) case, it becomes necessary to leave some links idle, violating constraint 3.

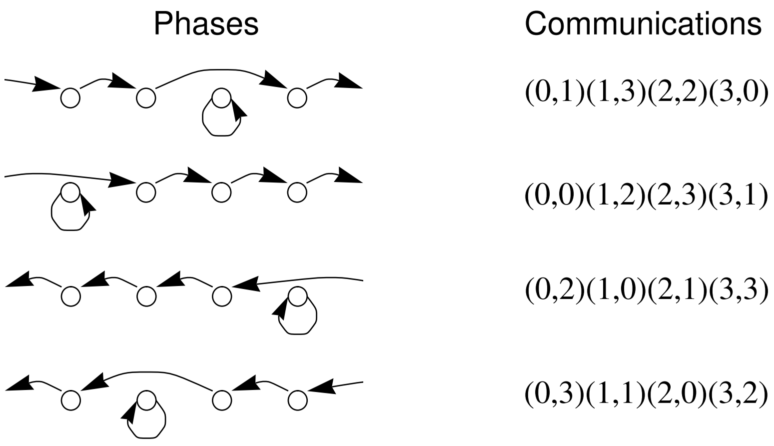



Figure 1: One possible division of the communication needed for a four node AAPC into contention-free phases.

**Lower bound on number of phases** It is helpful to 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:

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

A bidirectional array has double the communication bandwidth between bisections; in this case the lower bound is $n^{d+1}/8$.

In the one-dimensional, unidirectional case, $n^2$ messages need to be sent. Formula 2 shows that the lower bound on the number of phases needed is $(n^2/4)$. This implies that we will need to send on average four messages per phase in order to meet this lower bound.

**Constructing the phases** Now we present a method for constructing phases that adheres to our four optimality constraints. Each phase is a circular *chain* of four messages, in which the destination node of one message serves as the source node of the next, and so on. In order to meet the constraints, we pair up short messages with long ones in the following way: 0 hop messages with $n/2$ 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. Figure 2 shows one such phase.

One interesting property of these 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 both 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. For example, Figure 2 shows the (0,1) phase, since the message

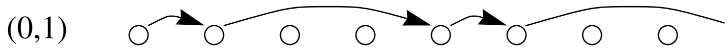


Figure 2: One circular phase consisting of four messages.

contained in the first half of the ring is from node 0 to node 1. Using this notation, we can concisely describe the set of all phases for $n = 8$ as:

$$\{(i,j) \mid i = 0 \ldots (\frac{n}{2} - 1), j = 0 \ldots (\frac{n}{2} - 1)\}$$

Special care must be taken when constructing the phases that chain 0 hop messages to $n/2$ hop messages. If we were to chain them according to the previous guidelines, the result would be that two processors would send and receive two messages each, in violation of Constraint 4. In these cases, we modify the chaining rule so that the source node of an 0 hop message is the node *before* the destination node of a $n/2$ hop message. (Likewise, the source node for the next $n/2$ hop message is the node *after* the destination node of a 0 hop message.) Figure 3 shows one such phase constructed by this augmented rule.

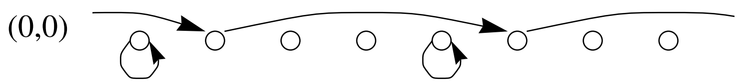


Figure 3: One circular phase combining 0 hop and $n/2$ hop messages.

Given the chaining rule, constructing the phases is straightforward. The greedy algorithm shown in Figure 4 suffices. Figure 5 shows all phases generated by this algorithm for $n = 8$.

**Making the transition to two dimensions** The two-dimensional phases we describe in Section 2.1.2 are easily derived from the one-dimensional phases. However, they impose two additional constraints on those phases:

5. The number of phases in each direction must be equal

6. In a given direction, the phases that pair 0 hop and $(n/2)$ hop messages must be node-disjoint with respect to one other.

The one-dimensional phases described in the previous section do not satisfy these constraints, but can be easily modified to do so. From Section 2.1.1 and the greedy algorithm shown in Figure 4, it should be clear that the number of phases in both directions would be the same, were it not for the phases that pair 0 hop and $(n/2)$ hop messages. These phases all communicate in the clockwise

*Messages* $\leftarrow$ The set of all messages that must be sent except 0 hop and $n/2$ hop messages
**while** *Messages* $\neq \emptyset$
  **remove** some message *m=(source,destination)* from *Messages*
  *Phase* $\leftarrow \{m\}$
  **repeat** 3 times
    **remove** a message *m′=(source′,destination′)* from *Messages*, such that:
      **direction**$(m')$ = **direction**$(m)$ and
      **length**$(m')$ = $(n/2)-$**length**$(m)$ and
      *source′ =destination*
    *Phase* $\leftarrow$ *Phase* $\cup \{m'\}$
    $m \leftarrow m'$
  **output** (*Phase*)

*Messages* $\leftarrow$ The set of all $n/2$ hop messages
**while** *Messages* $\neq \emptyset$
  **remove** some message *m=(source,destination)* from *Messages*
  **remove** some message *m′=(source′,destination′)* from *Messages*, such that:
    *source′ =destination*
  *Phase* $\leftarrow m \cup m'$
  $m \leftarrow (source - 1, source - 1)$
  $m' \leftarrow (source' - 1, source' - 1)$
  *Phase* $\leftarrow Phase \cup m \cup m'$
  **output** (*Phase*)

Figure 4: Greedy algorithm for creating optimal one-dimensional phases from the list of required communications.

direction. The solution is straightforward: we reverse the direction of half of those phases, being careful to satisfy constraint 6.

We enforce constraint 6 to simplify the creation of dense two-dimensional phases for the 0 hop $(n/2)$ hop messages described in the next section. With these additional constraints on the 0 hop $(n/2)$ hop messages, Figure 6 depicts all the phases for $n = 8$.

#### 2.1.2 AAPC with two-dimensional, unidirectional links

We now show how to extend these one-dimensional 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.

**Cross products** 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

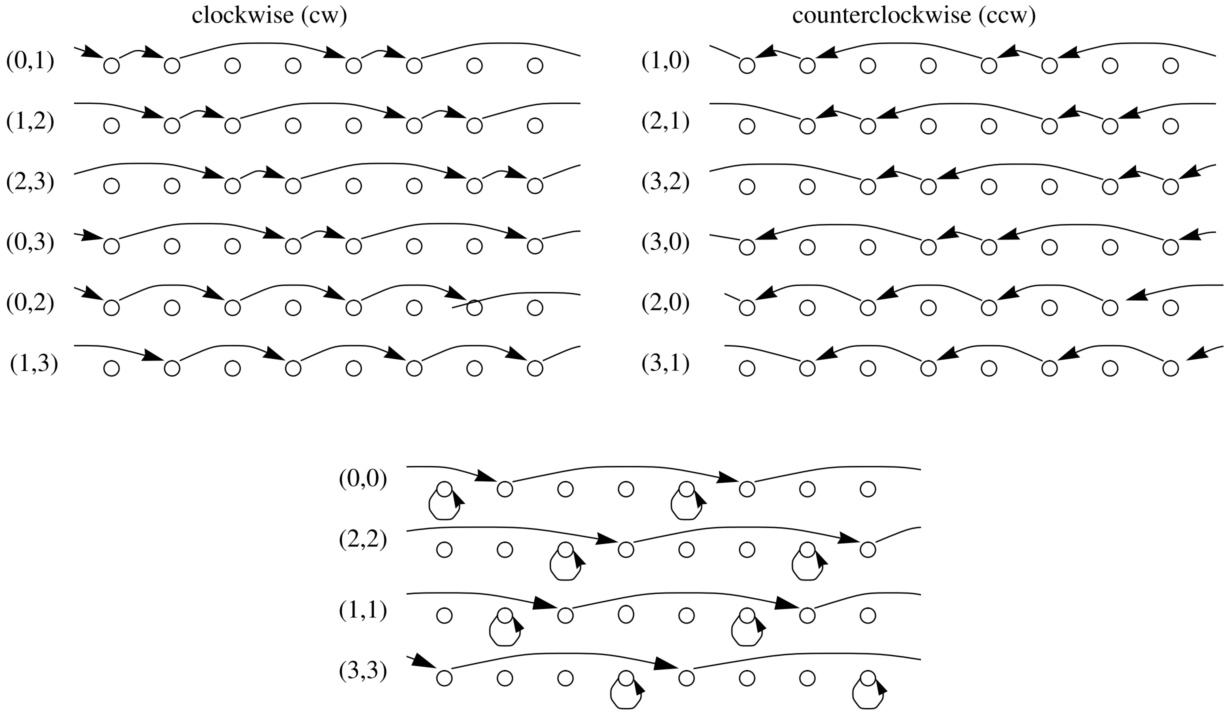



Figure 5: All phases generated by the greedy algorithm for $n = 8$.

two-dimensional pattern composed of cross products of all possible pairs of messages from $p$ and $q$. The bold arrows in Figure 7 show the cross product operation on two messages, and the entire figure shows the cross product operation on two patterns.

**Creating dense phases** As can be seen from Figure 7, 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. 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 $M_a \cdot M_b$ taking the cross produce of corresponding patterns $p_i$ and $q_i$ and overlaying the results:

$$p_0 \times q_0 + \ldots + p_{(n/4)-1} \times q_{(n/4)-1}$$

where $+$ is the pattern overlay operation. We define $\overline{p_i}$ as the counterclockwise pattern which

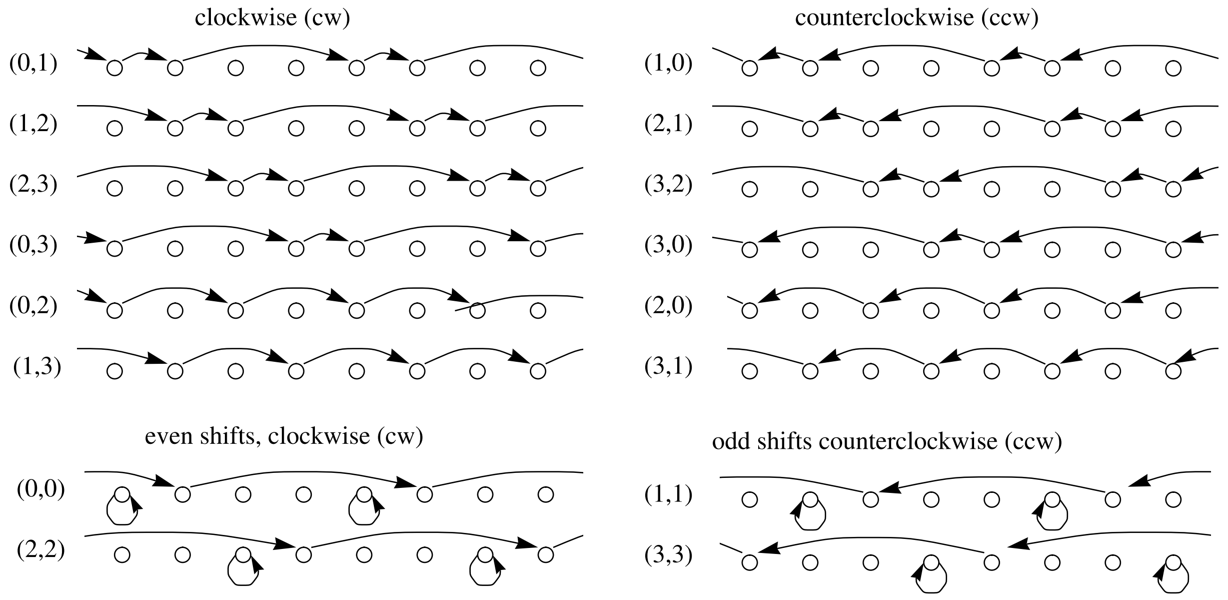



Figure 6: Here we show all one-dimensional phases for $n = 8$ following the additional constraints for the 0 hop, $n/2$ hop messages.

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 **r** such that if $M_a = (p_0, p_1 \ldots p_{(n/4)-1})$, then $\mathbf{r}(M_a) = (p_1, \ldots, p_{(n/4)-1}, p_0)$, and $\mathbf{r}^k(M_a)$ is **r()** applied to $M_a$ $k$ times. 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

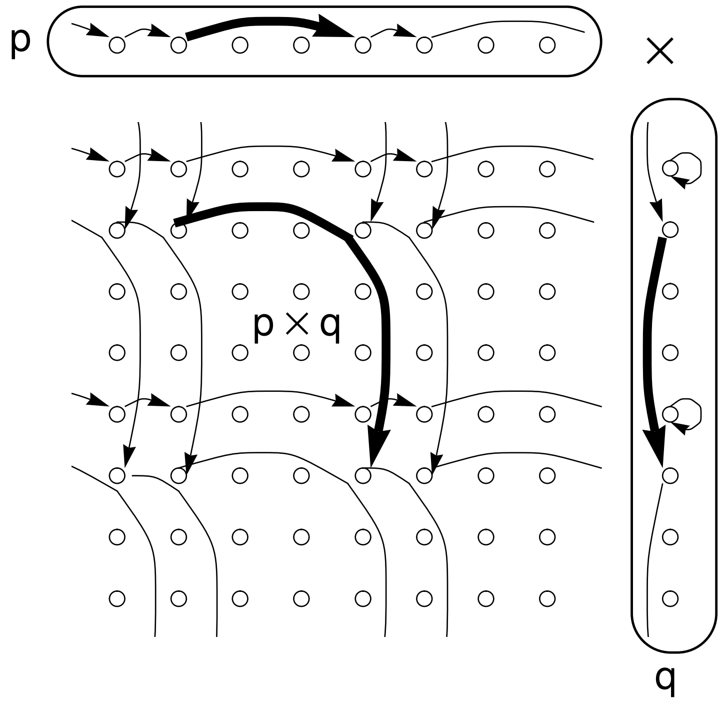



Figure 7: Illustrating the *cross product* operation to create a message in two dimensions based on two one-dimensional messages.

as:

$$\left\{\begin{array}{c} M_i \cdot r^{\,k}(M_j), \\ M_i \cdot r^{\,k}(\overline{M_j}), \\ \overline{M_i} \cdot r^{\,k}(M_j), \\ \overline{M_i} \cdot r^{\,k}(\overline{M_j}) \end{array}\right\} \tag{3}$$

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.

### 2.1.3 Phases with bidirectional links

It is straightforward to extend these unidirectional phases to the cases of rings and tori with bidirectional links. For bidirectional rings, look at the node disjoint $M$ tuples calculated in the previous section. Overlay elements of $M_i$ with elements of $\overline{M_i}$ as below:

$$\forall p_k \in M_i : p_k + \overline{p_{k+1}}$$

The elements that are overlaid must be node disjoint. All unidirectional phases are paired to form the bidirectional phases, so the number of bidirectional phases is half the number of unidirectional phases.

For the case of bidirectional tori, we must 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:

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

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 as calculated in Equation 2.

#### 2.1.4 Communication performance for phased AAPC

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

$$Agg_{phase} = \frac{\text{Total bytes sent}}{\frac{n^3}{8f}(T_s + T_t B)} = \frac{8fnB}{T_s + T_t B} \tag{4}$$

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.

#### 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.

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

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 8.

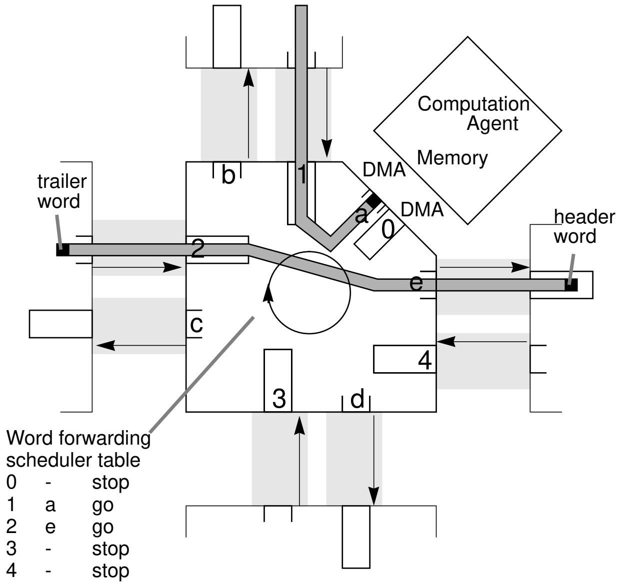



Figure 8: Basic structure of the communication agent of the iWarp VLSI component. Special header and trailer words control the forwarding state of the queues to construct connections without computation agent intervention.

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

```
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 9: 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.

### 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 9. 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[LAD+92, Int91, Ada93, HII92, Sni93], 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 $6 \times 6$ 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 10 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 9. 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 10: 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.

### 2.3 Overheads of synchronizing switch on iWarp

Here we outline the overheads in the synchronizing switch prototype implemented on an $8 \times 8$ iWarp system. We relate the overheads in cycles for a system running at 20 MHz. The speed of the synchronizing switch is of great importance, since the overheads of the local switches propagate through the network, so the overheads are multiplied by the network diameter. Every 2 cycles of overhead increases the message size needed for half peak bandwidth by 4 bytes.

By measuring the time to perform an AAPC with no data, we determined that the synchronizing switch overhead is 333 cycles/phase. This time includes the time to propagate the message headers. The header takes 2 cycles per node and 2 to 4 cycles per link. For a network with diameter 8, this propagation delay accounts for 32 to 48 cycles in overheads.

Both the phased and the message passing implementations include overheads to start sending a messages. On iWarp this includes setting up the message, generating a route, and putting the router in the proper state to send and receive messages. This requires an additional 120 cycles.

The remaining 25 cycles/link are delays due to the software implementation of the synchronizing switch. We anticipate that this overhead would be eliminated if the queues enforced the local constraint in hardware as described in Section 2.2.4. Figure 11 shows a breakdown of processing overhead incurring in all iWarp implementations of AAPC.

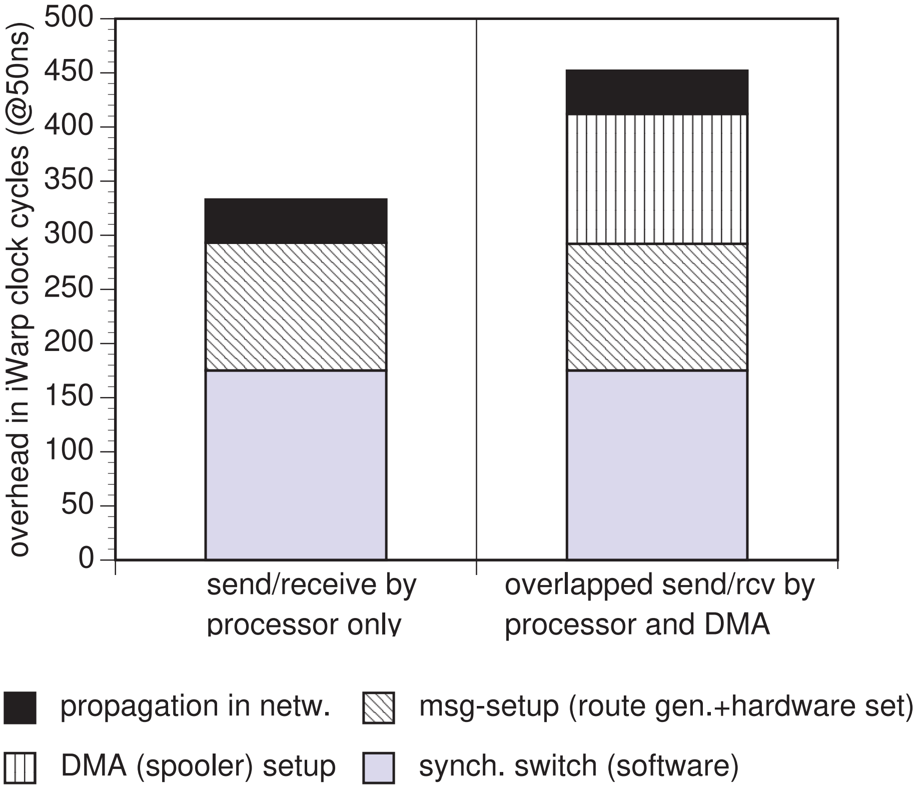



Figure 11: Breakdown of the per message processing overhead on a 64 cell iWarp system. The setup overheads apply to both, phased AAPC and e-cube message passing. The synch-switch overhead and the network-delay apply only to phased AAPC. The graph does not show congestion delay for the e-cube message passing, since it is not a constant processing overhead.

In addition to the synchronizing switch overheads, each phase incurs overheads to start sending data. Each iWarp router has 8 DMA controllers to move data to and from the node's memory, and our implementation of phased AAPC uses these DMAs. Starting the DMA agents and testing for the DMA completion adds an additional 120 clocks to the per phase overhead. Therefore, the complete per phase overhead on an $8 \times 8$ iWarp system is 453 cycles.

In principle the iWarp architecture can source and sink messages at 40 MBytes/s by either systolic or memory communication [GHH+94]. Systolic communication reads and writes data directly from the computation; network communicaiton looks like another register access. Memory computation relies on DMA agents (spoolers) to move data between the node's memory and the network without additional action by the computation agent. The phased AAPC routines can be implemented with either communication mode; but message passing must use memory communication and DMAs to handle the non-deterministic message arrival of long messages.

# 3 Alternative AAPC implementations

There are many characteristics one can use to categorize AAPC methods. One division of AAPC algorithms is between algorithms that utilize wormhole routing and those that only rely on store and forward routing. The phased AAPC algorithm requires a wormhole router to support sending messages directly between processors that are not physically adjacent.

The algorithm proposed in [VB92] only requires a very simple network interface that supports communication between physically adjacent processors. In this algorithm, all processors communicate to the same set of relative processors in each step. To send to relative processor (x,y) takes $|x| + |y|$ steps. In the first $|x|$ steps all processors send and receive data along the X axis, and in the last $|y|$ steps all processors send and receive along the Y axis. By communicating to relative destinations simultaneously, this algorithm will utilize all of the machine's network bandwidth. However, this requires a machine that has twice the memory bandwidth as network bandwidth. This is not the case with iWarp and many other distributed machines are also more balanced with respect to network and memory bandwidth. iWarp can only support two simultaneous relative destinations, so this algorithm will at best reach half of iWarp's peak aggregate network bandwidth.

Also, AAPC algorithms differ by whether data is sent directly to the final destination or not. The phased AAPC algorithm sends each message directly to its final destination. This algorithm requires $N^2$ message start ups and sends messages of size $B$.

In other algorithms, the processor sends several blocks of data destined for several different processors in one message. Generally, these blocks of data are destined for the same dimension or area of the machine. While a block of data may be sent to several intermediate processors, the size of each message is larger and the actual number of message start-ups is smaller.

Several of these multistep algorithms for a mesh are described in [BB92]. Here is one simple two step algorithm for a torus. First, perform an AAPC along the rows to arrange all the data in its target column. Then perform an AAPC along the columns to move all the data to its target row. This algorithm needs only $2\sqrt{N}$ message startups and moves blocks of size $\sqrt{N}B$. The larger blocks better amortize the cost of the message start-up, but this algorithm requires more buffer management and only uses half of the network bandwidth.

If message start-up is very expensive, the multistep algorithm is a good approach. However, the trend for newer distributed memory machines is to lower the message start-up time through specialized communication hardware and user level access to the network among other techniques.

Another major division is between *informed* and *uninformed* methods. Informed methods such as the phased AAPC algorithm use compile time information about the structure of the communication to aid in developing good routing and scheduling information.

Message passing can be used to implement an uninformed communication step. 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[CKP+92, BNK92]. Figure 12 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 12: Message passing AAPC code. We assume NBSendMessage and NBReciveMessages are buffered, non-blocking primitives offered by the message passing library.

Uninformed algorithms are for more adaptable and may be the only alternative if no compile time information is available. However, with dense communication steps uninformed methods tend to have problems with hot spots and congestion. In [Val82], Valiant proves that uninformed or oblivious routing can statistically avoid these hot spots by first routing to a random location and then routing to the final destination. On average this will double the message route length, so this approach will at best get within half of the optimal network usage for AAPC.

Run time adaptive routing methods have been proposed and studied as a means to avoid hot spots. However, today's generation of distributed memory machines do not incorporate adaptive routing, so the value of adaptive routing algorithms can not be evaluated. However, it seems unlikely that the local view available to an adaptive router is sufficient to avoid all hot spots that can occur in an AAPC.

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. Figure 13 compares the performance of AAPC message passing using the phased schedule with and without synchronization.

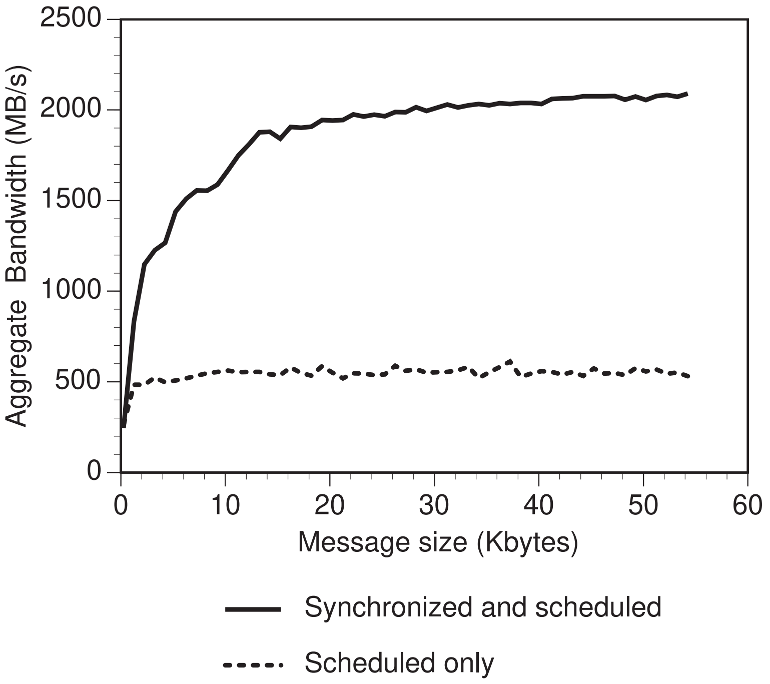



Figure 13: Performance of two message passing AAPC programs both following the phased schedule. One synchronizes between phases, and the other does not.

### 3.1 The iWarp message passing system

On iWarp, the basic message passing routers are based on a reverse e-cube scheme[Str91]. 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[BGPS92], 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[Sun90] or MPI[MPI93]. 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[SSOG93], based on the deposit model[SSO+94]. 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 $8 \times 8$ iWarp system. Each node of this system runs at 20 MHz (S) and The physical transfer time between two nodes is 0.1 $\mu$seconds ($T_t$), and data is transferred in flits of 4 bytes ($f$). 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 the AAPC algorithms described in Section 3. Figure 14 shows the aggregate bandwidth performance of these algorithms. These measurements were made for AAPC where all messages are the same size.

The message passing implementation operates at 500 MB/s, about 20% of optimal. In theory, this message passing implementation could reach the machine's optimal bandwidth, but the uninformed routing approach makes network congestion quite likely.

The performance of the store and forward algorithm 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.

Since the two stage algorithm uses at most half of the links simultaneously, it also constrained to half of the optimal aggregate bandwidth. Our measurements show that the two phase algorithm can out-perform other methods on small messages but approaches the same performance limit as the single stage store and forward algorithm.

Given our prototype implementation on iWarp, the phased algorithm with the synchronizing switch outperforms other methods for messages greater than 512 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.

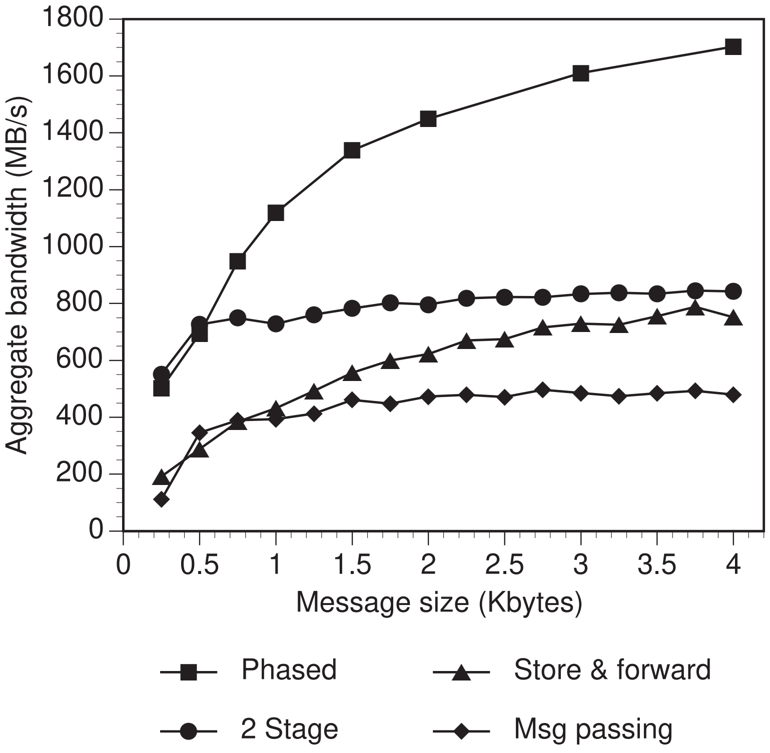



Figure 14: Performance of a variety of AAPC implementations.

## 4.2 Synchronization support

Figure 15 compares the effect of global and local synchronization on phased AAPC performance for a larger range of message sizes. In addition to the local switch, we measured the performance of phased AAPC using two versions of a global synchronization. The hardware-based global synchronization completes in 50 $\mu$ seconds. The software-based global synchronization completes in 250 $\mu$ seconds.

The AAPC program with local synchronization consistently out-performs the globally synchronized implementations as one would expect from the lower-overhead local synchronization constraints. However, the implementation using hardware global synchronization shows quite similar performance. The implementation that used software-based global synchronization shows a distinct performance penalty, though it too approaches the same bandwidth utilization.

Since the hardware global synchronization works so well, it makes sense to use global synchronization for phase separation if the target architecture already has a fast, hardware global synchronization mechanism. However, if the machine must rely on software for synchronization, it may make sense to consider added support for a local synchronization as described in Section 2.2.4. AAPC phase separation does not require the power of full global synchronization, and the synchronizing switch should be a simpler and more scalable design addition.

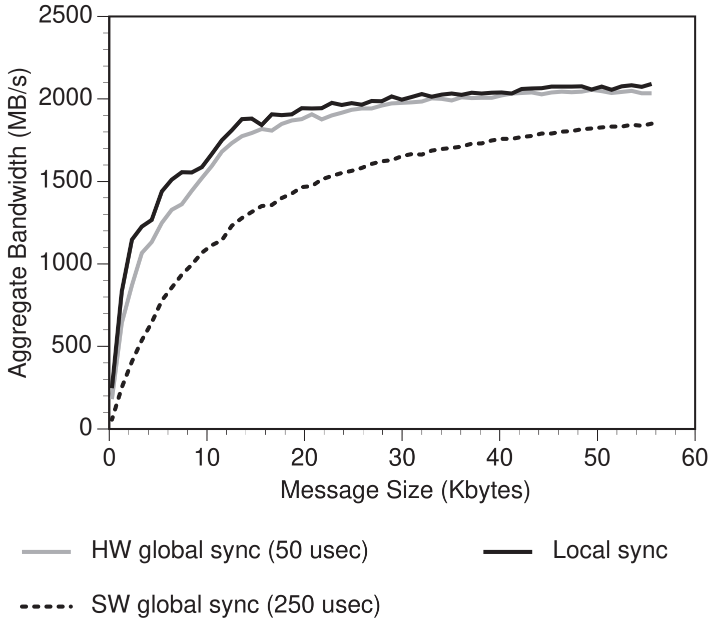



Figure 15: Performance of the phased AAPC algorithm with global synchronization versus local synchronization on iWarp.

### 4.3 AAPC on other machines

In addition to measuring AAPC performance on iWarp, we compared the iWarp AAPC performance with all-to-all communication performance on three other commercial distributed memory systems: the Cray T3D[Ada93], Thinking Machine's CM-5[L$^+$93], and IBM's SP1[Sni93]. Figure 16 compares the performance of these systems over a range of message sizes. All systems use 64 nodes. The T3D is a $2 \times 4 \times 8$ 3-dimensional submesh with a bisection bandwidth of 1.6 GB/s. The CM-5 is a 64 node fat tree with a bisection bandwidth of 320 MB/s.

The AAPC algorithm on the CM-5 was optimized for use in the CM-5 scientific library. The algorithm and performance numbers are due to Unger[Ung94].

The SP1 AAPC algorithms and measurements are from [BHKW94]. This algorithm attempts to minimize endpoint processing. Since SP1 is a switch-connected machine, it does not attempt to optimize network use.

We made two preliminary measurements on the T3D. In the first naive implementation (labeled "unphased"), each node sends messages with no synchronization. This implementation works well until it reaches an aggregate bandwidth of 2 GB/s where network congestion appears to be an issue. The second implementation (labeled "phased") divides the messages into 64 simple phases and synchronizes the nodes between each phase. With this synchronized implementation, the aggregate bandwidth continues on beyond 3 GB/s. This comparison shows that even new, high bandwidth interconnects benefit from careful utilization of network bandwidth for all-to-all communication.

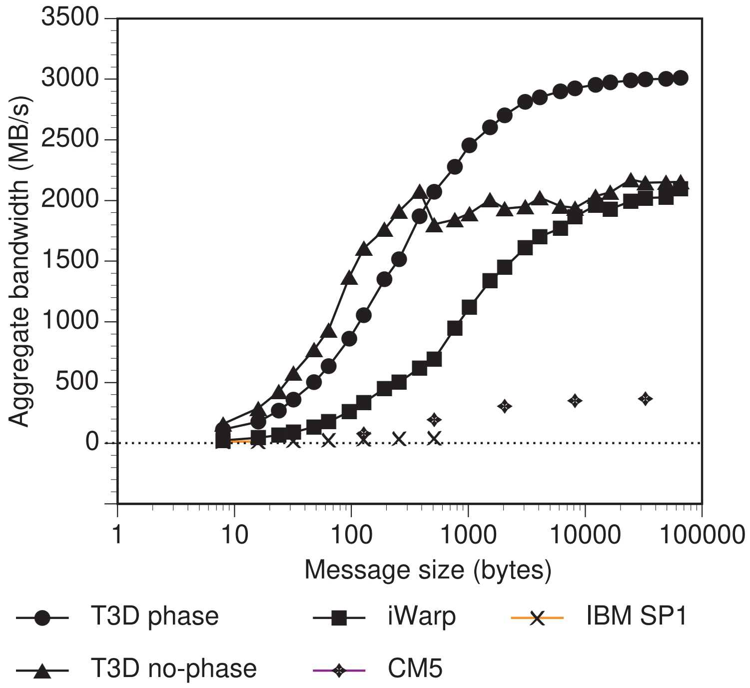



Figure 16: Performance of the phased AAPC algorithm on several machines.

### 4.4 Message size variation

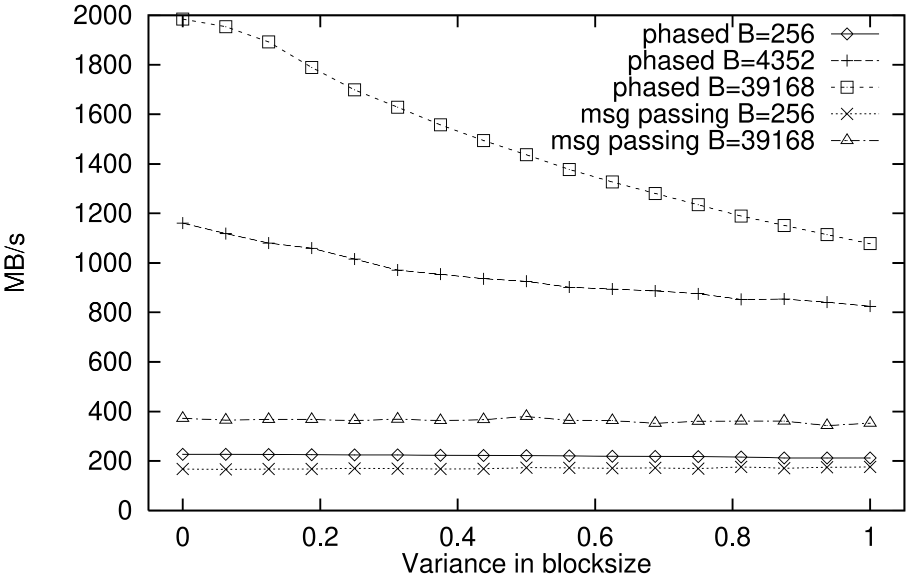
We also measured the effects of varying message sizes on AAPC performance in two experiments. We compare the performance of the phased AAPC algorithm with the performance of the uninformed message passing. Since the message passing library delays decisions to run time, it should be more adaptable to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

Figures 17(a) and 17(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.

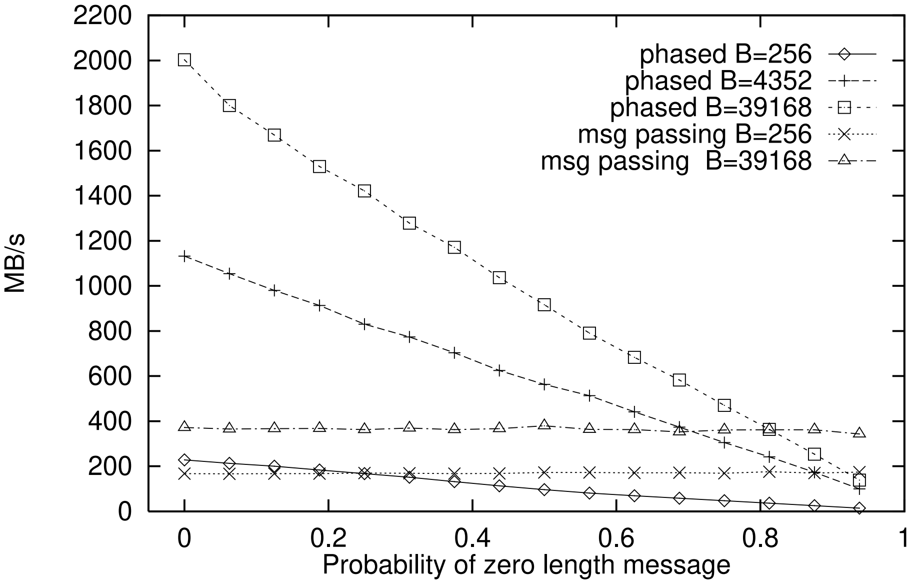
Figure 17(a) shows that the phased algorithm performance decreases as the variance increases. The message passing versions are unaffected by the message variation, but for the corresponding block size, the phased algorithm outperforms the message passing algorithm. Even though the message sizes vary, 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 17(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.




(a)




(b)

Figure 17: 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.

### 4.5 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 [FSW93][4]. Table 1 shows the performance of these communication steps.

| 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.

### 4.6 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 $512 \times 512$ 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[SOG94], 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 $512 \times 512$ two-dimensional FFT executed on an $8 \times 8$ 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 $512 \times 512$ 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 18 shows the performance break down of several two-dimensional FFTs executed on a $8 \times 8$ iWarp.

[4]The performance of the FEM communication step in this paper differs from the performance in [FSW93] because this implementation does not measure the application buffering costs.

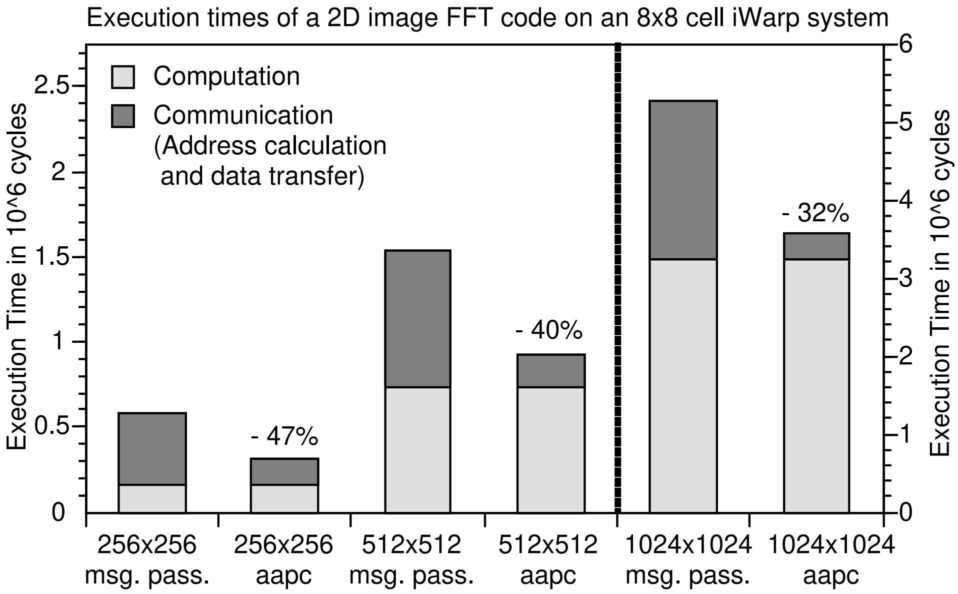



Figure 18: Performance improvements of a two-dimensional FFT code using phased AAPC over a code based on message passing AAPC.

# 5 Conclusions

In this report, 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 report, Guy Blelloch for his suggestions and Thomas Gross and the rest of the CMU/ Intel iWarp group for their support.

## References

[Ada93] D. Adams. Cray T3D System Architecture Overview. Technical report, Cray Research Inc., September 1993. Revision 1.C.

[B+88] S. Borkar et al. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330–339, 1988.

[B+90] S. Borkar et al. Supporting Systolic and Memory Communication in iWarp. Technical Report 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.

[BB92] S. H. Bokhari and H. Berryman. Complete Exchange on a Circuit Switched Mesh. In *Proc. Scalable High Performance Computing Conference*, pages 300–306, Williamsburg, VA, April 1992.

[BGPS92] P. Berman, L. Gravano, G. Pifarre, and J. Sanz. Adaptive Deadlock- and Livelock-Free Routing with All Minimal Paths in Torus Networks. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 3–12, San Diego, June 1992. ACM.

[BHKW94] J. Bruck, C. T. Ho, S. Kipnis, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, Cape May, NJ, June 1994. ACM.

[BNK92] A. Bar-Noy and S. Kipnis. Designing Broadcating Algorithms in the Postal Model for Message-Passing Systems. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 13–22, San Diego, June 1992. ACM.

[Bok91] S. Bokhari. Multiphase complete exchange on a circuit switched hypercube. Technical Report 91-5, ICASE, January 1991.

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

[FSW93] A. Feldmann, T. Stricker, and T. Warfel. 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.

[GHH+94] T. Gross, A. Hasegawa, S. Hinrichs, D. O'Hallaron, and T. Stricker. The Impact of Communication Style on Machine Resource Usage for the iWwarp Parallel Processor. *Computer*, December 1994. To appear.

[HH91] T. Horie and K. Hayashi. All-to-All Personalized Communication on a Wrap-around Mesh. In *Proceedings of CAP Workshop*, Canberra, Austrailia, November 1991.

[Hig93] High Performance Fortran Forum. *High Performance Fortran Language Specification Version 1.0.*, May 1993.

[HII92] T. Horie, H. Ishihata, and M. Ikesaka. Design and implementation of an interconnection network for the AP1000. In *Proc. IFIP World Computer Congress*, volume I, pages 555–561. Information Processing, 1992.

[Hin94] S. Hinrichs. Compiler Resource Management for Connection-Based Communication. Internal document, 1994.

[Int91] Intel Corp. *Paragon X/PS Product Overview*, March 1991.

[JH89] S. L. Johnsson and C.-T. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, September 1989.

[L+93] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 272–85, 1993.

[LAD+92] C. Leiserson, A. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, D. Hillis, B. Kuszmaul, M. St.Pierre, D. Wells, M. Wong, S. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, San Diego, June 1992. ACM.

[MPI93] The Message Passing Interface Forum. *Draft Document for a Standard Message Passing Interface*, November 1993.

[Sco91] D. S. Scott. Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies. In *The Sixth Distributed Memory Computing Conference Proceedings*, pages 398–403, 1991.

[Sni93] M. Snir. Scalable Parallel Computing - The IBM 9076 Scalable POWERParallel-1. In *ACM Symposium on Parallel Algorithms and Architectures*, page 42. ACM, June 1993.

[SOG94] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating Communication for Array Statements: Design, Implementation, and Evaluation. *Journal of Parallel and Distributed Computing*, 1994. to appear.

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

[SSOG93] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. 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.

[Str91] T. Stricker. Message Routing on Irregular 2D-Meshes and Tori. In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 170–177, Portland, OR, April 1991. Also appeared as Technical Report CMU-CS-91-109, Carnegie Mellon School of Computer Science.

[Sun90] V. S. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–39, December 1990.

[Tak87] R. Take. A Routing Method for All-to-all Burst on Hypercube Networks. In *Proc. 35th National Conference of Information Processing Society of Japan*, pages 151–152, 1987. In Japanese.

[TNY91] R. Take, Y. Noguchi, and H. Yokota. An Architecture for Parallel Database Computing. In *Transputing '91*, pages 1–14, 1991.

[Ung94] Leo Unger. Optimized Matrix Transpositions on the Connection Machine CM-5. Personal communication, February 1994.

[Val82] V. G. Valiant. A Scheme for Fast Parallel Communication. *SIAM Journal on Computing*, 11:350–361, 1982.

[VB92] E. A. Varvarigos and D. P. Bertsekas. Communication algorithms for isotropic tasks in hypercubes and wraparound meshes. *Parallel Computing*, 18:1233–1257, 1992.