Reprint from the proceedings of the ACM Symposium on Parallel Algorithms and Architectures, SPAA92
June 29 - July 1, 1992, San Diego, California.

Subset Barrier Synchronization on a Private-Memory Parallel System

Anja Feldmann Thomas Gross David O'Hallaron Thomas M. Stricker

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A global barrier synchronizes all processors in a parallel system. This paper investigates algorithms that allow disjoint subsets of processors to synchronize independently and in parallel. The user model of a subset barrier is straight forward; a processor that participates in a subset barrier needs to know only the name of the barrier and the number of participating processors. This paper identifies two general communication models for private-memory parallel systems: the *bounded buffer broadcast model* and the *anonymous destination message passing model* and presents algorithms for barrier synchronization in the terms of these models. The models are detailed enough to allow meaningful cost estimates for their primitives, yet independent of a specific architecture and can be supported efficiently by a modern private–memory parallel system. The anonymous destination message passing model is the most attractive. The time complexity to synchronize over a *uni-directional ring* of N processors is $O(\log N)$ for common cases, and $O(\sqrt{N})$ in the worst case. The algorithms have been implemented on iWarp, a private–memory parallel system and are now in daily use. The paper concludes with timing measurements obtained on a 64-node system.

1 Introduction

Barrier synchronization is a useful technique for organizing the execution of a parallel program into a sequence of loosely coordinated phases. For example, a phase of a data–parallel program running on a private memory system might consist of a communication step, where processors exchange messages among themselves, followed by a computation step, where the processors operate independently on their local data. In this example, a barrier synchronization between data–parallel phases keeps messages from different phases from being intermixed.

We call a barrier that involves a group of processors a *subset barrier*, and we say a processor *participates* in a barrier synchronization if it has to wait until all other processors in the subset for this barrier have reached the given barrier. To provide a basis for parallel program generators, we require that an arbitrary subset of processors can participate in a barrier. Further, disjoint subsets of processors can participate in different barriers in parallel. This property allows us, for example, to allocate disjoint subsets of processors to different data–parallel tasks, and then synchronize each of these data–parallel tasks independently and in parallel.

Since the mapping of processes to processors may not be known at compile time, we cannot rely on the identifiers (Ids) of each processor to identify different subset barriers. That is, if two processes P_a and P_b participate in one barrier B_1 and three other processes P_x, P_y, and P_z participate in another barrier B_2, then we cannot use the unique Ids of the processors that executed P_a, P_b, P_x, P_y, and P_z to determine if a barrier is reached (even if we map a single process onto each processor), since the barrier is placed in the programs before the mapping of processes to processors is known. Similarly, barriers based on bit masks do not do justice to the range of programs that can be executed on a MIMD computer. Therefore, we introduce *named barriers*. Names do not have be unique; the only restriction is that no two barriers with the same name can be active at the same time.

It is desirable to limit the amount of information that must be exchanged by messages. A processor participating in a named barrier need only know the barrier name and the number of participating processors. (If either of these is omitted, the problem of determining if all processors reached the barrier becomes undecidable.) The identity of the participating processors is determined by the barrier synchronization algorithm in a distributed fashion. Thus, both the identity and the number of processors participating in a named barrier can change over time, without having to maintain any global runtime mapping information.

Furthermore, we require that all processors participating in a named barrier use a *symmetric* protocol, that is, they execute the same code for synchronization. If one processor is to perform extra work (e.g., it determines that all processors have reached the barrier), then the barrier algorithm must dynamically identify one processor out of the subset of processors.

Previous work on barrier synchronization has focused on specific synchronization hardware [13, 2, 14, 8, 9, 11] and on algorithms

*Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by DARPA/CMO to Carnegie Mellon University, in part by the Air Force Office of Scientific Research, Grant F49620-92-J-0131 and in part by the National Science Foundation under grant CCR-8658139.
The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

for synchronizing common memory systems [1, 12, 6, 10, 15]. In this paper we investigate subset barrier synchronization in the context of private memory systems without specific synchronization hardware. We introduce two general models for communication systems: the *bounded buffer broadcast model* and the *anonymous destination message passing model*. We describe subset barrier synchronization algorithms for the general models, and then show how the algorithms for the general models can be adapted to the communication structures of a specific private memory system, in our case, the iWarp computer[3, 4]. For a ring network, we show the somewhat counterintuitive result that the execution time of the algorithm is $O(\sqrt{N})$, where N is the total number of processors. The analysis is supported by measured execution times on iWarp. In addition to providing a convenient package that is used both by iWarp user programs as well as the iWarp runtime system, these algorithms give us an opportunity to evaluate how some of the unique features of the iWarp communication system affect barrier synchronization.

2 Algorithms and communication models

Barrier synchronization algorithms consist of two phases:

Phase 1: Determine that every participating processor has reached the barrier.

Phase 2: Inform every participating processor of the successful completion of Phase 1.

We say Phase 1 ends if one processor has reached Phase 2. Phase 2 can be empty if every processor can determine that every processor has reached the barrier. For any symmetric barrier synchronization protocol on a private memory MIMD system, each processor reaching a barrier must somehow signal this fact by sending some data to another processor. In general, a processor must send a message to at least *one* other participating processor, and at least one participating processor must be able to receive information about *all* of the others. The information contained in receiving a message is that one additional processor has reached the barrier. The algorithms described in this paper have the key property that only *one* processor (determined at runtime) needs to receive information from *all* other participating processors.

We introduce two communication models and develop symmetric subset barrier synchronization algorithms for these models. The algorithms share the following advantages:

- Each processor uses a constant number of message buffers.
- Non–participating processors incur minimal overhead.

2.1 Bounded buffer broadcast

If we guarantee that the processor that reaches the barrier first receives messages from all other participating processors, then all other processors currently not at a barrier can safely ignore (discard) any messages that arrive. After one of the processors has received messages from all participating processors, it enters Phase 2 and broadcasts a successful completion message. This message is then received by all participating processors that are waiting for messages, and can be ignored (discarded) by all other processors. Receiving such a message allows a processor to continue its computation.

The *bounded buffer broadcast* model, which provides for fast broadcasting and for fast and easy discarding of messages, is characterized by the following:

1. Each processor can broadcast a message to all other processors, and this message is guaranteed to arrive at its destination within some fixed time interval.
2. Messages are stored at every processor in FIFO (first in first out) order in a bounded number of buffers. If a message arrives but no buffer is free, the oldest message is retired to make space for the newly received message.
3. Only messages currently buffered at a processor can be read by this processor.

This model includes two communication primitives, *broadcast* and *receive*. The broadcast primitive sends the message to all other processors (i.e. the message is buffered in one of the buffers in every processor). The receive primitive removes a message from the top most buffer and transfers it to the user. Discarding of messages is implicit because messages are thrown out whenever there is no space available. This model allows us even to discard messages without interrupting the computation on the processor that accumulated the messages. Algorithm 1 is a barrier synchronization algorithm for the bounded buffer broadcast communication model.

Algorithm 1 Generic Bounded Buffer Broadcast Barrier

barrier(b_name, count)

b_name.reached and *b_name.complete* are identifiers generated uniquely from *b_name*.

```
i = 0;                                        /* Phase 1 */
/* discard old messages */
broadcast (b_name.reached);
do
    receive(name_rec);
    if (name_rec == b_name.reached)
        i++;
until ((i == count) or
              (name_rec == b_name.complete));
if (i == count)                               /* Phase 2 */
    broadcast (b_name.complete);
    receive();
wait(till broadcast completed);
```

This model is fairly realistic (its reliance on a bounded number of resources (buffers) will be appreciated by any implementor). Section 3 will provide an example of how this model can be implemented on a specific parallel system. Note that the bounded buffer broadcast model makes no assumptions on the number of buffer elements; a single buffer element is sufficient for Algorithm 1.

2.2 Anonymous destination message passing

In the bounded buffer broadcast model, messages are discarded at the destination after being broadcast to all the processors in the system. In some communication systems, broadcasting a message and then removing it at the destination is a costly operation, so we also investigate a model that does not rely on broadcasting and discarding. In this model, a processor receives messages only after it has reached a barrier.

In the generic message passing model, each processor has a unique identifier, and this identifier is used to address the processor. A processor P_s sends a message to a destination processor P_d by providing P_d's identifier (as well as the data of the message) to the communication system. The message then consists of a header H and data D, and the destination identifier (plus maybe some routing information, depending on the router used) is encoded in the header. This message is then transported to the destination processor; it can be received only by this processor. Note that we expect that the destination identifier is encoded in the header of the message; systems that do not meet this requirement may encounter difficulties implementing algorithms based on this model or models derived from it.

We now consider a modification of the generic message passing model: *anonymous destination message passing*. This model differs from generic message passing in the following way: the destination of a message is determined not only by the sending processor but also by the receiving processors. There is exactly one receiver processor, but the sender does not know the exact identifier of the receiver processor at the time the message is sent. Thus this model is characterized by three features:

1. Each message consists of a message header and some data.
2. Every processor can specify the destination header of messages that it wants to receive.
3. If the header of a message does not match the destination header as specified by a processor, the message is automatically forwarded to a default processor. (The default processor could be for example the next processor based on the physical topology.)

A sender processor does not know which processor receives the message, but it decides what "identifier" the receiver must assume to receive the message. The second and the third feature allow a processor to assume different identifiers. The model has two communication primitives:

1. *selective_send* (h, d): send data d with a header h to unspecified receiver.
2. *selective_receive* ($\{h\}$, ha, a): receive a message with a header h' in the set $\{h\}$ and store the header at local address ha and the data at a.

Note that a bounded buffer broadcast model can be rewritten for the anonymous destination message passing model, so Algorithm 1 can still be used. However, barrier synchronization can exploit the fact that in this model a processor can actually send messages only to participating processors. A processor that reaches barrier B sends a message with header H_b to an anonymous processor with Id H_b. (H_b is related to the barrier name.) This message then travels around until another processor decides to receive a message with header H_b. If a processor does this only if it is participating in the same barrier, we have a way to send a message to some processor that is participating in the same barrier and has already reached it.

A barrier synchronization algorithm for the anonymous destination message passing model is given in Algorithm 2. Note that correctness and some details of Algorithm 2 are highly dependent on some properties of the communication system (e.g., what is the "default" processor?). We assume that every processor eventually is able to see every message. For a more detailed discussion see Section 4.2.

Algorithm 2 Generic Anonymous Destination Message Passing Barrier

barrier(b_name, count)

b_name.reached and *b_name.complete* are identifiers generated uniquely from *b_name*.

```
i = 0;                                          /* Phase 1 */
selective_send (b_name.reached, dummy_data);
do
    selective_receive({b_name.reached,
        b_name.complete},name_rec, dummy_adr);
    if (name_rec == b_name.reached)
        i++;
until ((i == count) or (name_rec ==
            b_name.complete));
if (i == count)                                 /* Phase 2 */
    selective_broadcast (b_name.complete,
        dummy_data );
                /* Implemented by selective_send() */
    selective_receive({b_name.complete}, name_rec,
        dummy_adr );
```

3 iWarp communication structures

The iWarp System is based on a single chip VLSI processor [3, 4] developed jointly by Carnegie Mellon and Intel Corp. The iWarp component contains a *computation agent*, and a *communication agent* for transferring data to and from other iWarp processors. Computing rates up to 20 Megaflops and 20 MIPS per cell are possible with the computation agent. The communication agent can sustain a bandwidth of 320 Megabytes/second per cell with a latency of 0.2 μs/hop. iWarp systems are 2–dimensional tori of iWarp processors, ranging in size from 4 processors to 1024 processors.

The communication primitives of the bounded buffer broadcast model and the anonymous destination message passing model use three architectural features: a *broadcast unit*, *logical channels/pathways*, and a *user–configurable address match CAM (Content Addressable Memory)*.

The iWarp communication architecture allows the sharing of the communication resources between the different services of the runtime system and the modules of user applications. In the current iWarp system design, the service of *flexible barrier synchronization* is allocated a minimal set of resources, sufficient for message communication: a *unidirectional ring*. With additional resources, barrier synchronization could take advantage of the bidirectional or of the two dimensional torus structure of the iWarp system.

3.1 Implementation of the bounded buffer broadcast primitives

Each iWarp processor contains a *broadcast unit* (called System Reporting Unit [SRU]), which is connected to a global "wired or" network. The SRU transmits data, receives data and detects collisions using three shift registers on each iWarp processor. (The SRU is described in a simplified form since part of its logic is not used.) For a schematic diagram see Figure 1. To broadcast a one word message a word it is written to the sending register. The value of this register is appended by a leading 0 and then shifted out to the "wired or" network. In a successful transmission the receiving register contains the message and the zero in the jam bit. The jam detection logic is active during the sending process. If it discovers a collision (i.e. a discrepancy between the value the SRU is sending and the value the SRU is receiving), it switches from sending the value of the sending register to sending ones (i.e. sending the value of the jamming register appended by a 1). The receiving processors receives the jammed data and a one in the jam bit. A new message automatically discards any old message that has not yet been read. The SRU implements the bounded buffer broadcast model with a buffer size of one by its *broadcast* and *receive* primitives.

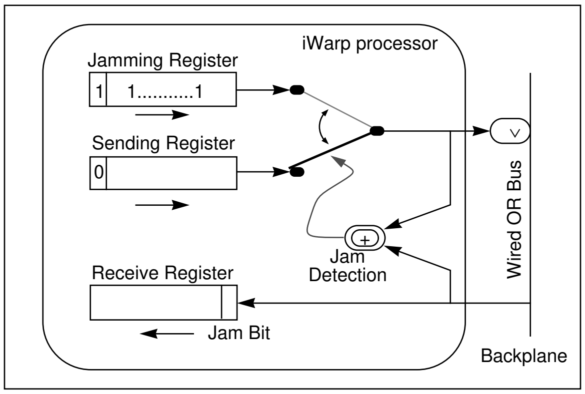


Figure 1: Schematic diagram of the iWarp System Reporting Unit

The costs per operation for iWarp implementations of the bounded buffer broadcast primitives are summarized below.

broadcast: One bit of data is broadcast every 0.8 μs. With 32 bit data, start, stop and status information, a message contains 36 bits. A message is broadcasted in 29 μs.

receive: Receiving is performed simultaneously with sending, 29 μs.

The “wired or” broadcast network is a circuit that implements an unbounded fan-out logical gate (i.e. a gate with an arbitrary number of inputs). The scalability problems of “wired or” networks are well known and have been discussed before. The current design of this unit (operating at 1Mhz) allows for at least 1024 processors.

3.2 Implementation of the anonymous destination message passing primitives

The anonymous message passing model uses the high bandwidth, low latency communication network of the iWarp system. For its operation it does not tie up the whole communication system but only reserves a small subset of the communication resources.

Logical channels

iWarp processors communicate with neighboring processors over structures called *logical channels*, which can be chained together to form *pathways*[4]. Although the physical connection pattern of the iWarp is a 2D torus, logical channels and pathways allow for logical connection patterns such as rings, trees, and hypercubes [16]. A network of logical channels can be created statically at compile time or created and destroyed dynamically at runtime. Each iWarp processor can support at most 20 logical channels at any point in time. A logical channel is a unidirectional communication medium. The network chosen for anonymous message pathway, configured as a unidirectional ring, that passes through all processors in the system.

User configurable cell names and message headers

A pathway through a particular cell starts in *express mode*, (see Figure 2 (b)) which allows data to travel along with a minimal delay (0.2 μs). Each iWarp processor contains an *address match CAM* with four entries that can be set by the program. When a special word called a *message header* arrives over a pathway at a processor, the match CAM hardware automatically compares the message header to its four entries. If any of the entries matches the message header, the pathway is stopped, i.e. the message header is held in the pathway, and the computation agent is informed that a *matched arrival* has occurred (e.g. as in Figure 2 (a)). If no match between a match CAM entry and the header was detected the message continues on to a “default” processor (see Figure 2 (b)). The “default” processor can be any neighbor of the current processor. For the purpose of barrier synchronization the “default” processor at every processor is chosen so that the pathways build a unidirectional ring through all processors of the iWarp system.

Pathway

When a *matched arrival* is signaled, the message passing system *splits the pathway* (see Figure 2 (a)) and consumes the message from the pathway. The message can contain an arbitrary number of words. After a message is received, the pathway is reset to express mode (this is called *joining a pathway*) (see Figure 2. Any further messages are treated in the same way.

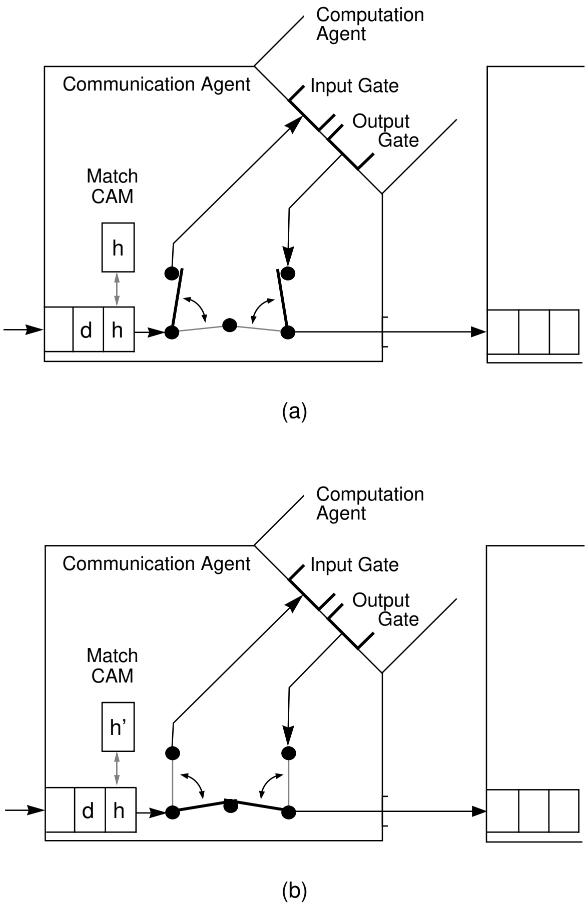


Figure 2: Schematic diagram of the switching logic in iWarp communication agent, to implement the ring based barrier synchronization; (a) pathways split after a matched arrival (b) pathways joined to forward a non-matching message.

The primitives

To implement the *selective_send* primitive on iWarp, the computation agent simply places a message header and data on the ring. The message circulates around the ring until one of the match CAMs in the ring matches the message header. Note that no message circulates forever because the match CAM of a sender is configured to match any message it has placed on the ring, since it must be prepared to receive messages with the same header from other processors participating in this barrier.

To implement the *selective_receive* primitive on iWarp, the computation agent writes the desired header to one of the match CAM entries, waits for a matching header to arrive, consumes the message header and data, and then joins the pathway so that a following message can proceed to the next processor in the ring.

The costs per operation for the iWarp implementations of the anonymous destination message passing primitives are summarized below.

express travel The latency of a message traveling "express" through intermediate processors is small (0.2 μs/hop).

selective_send: The cost of a *selective_send* is the cost of composing a message header (1 μs), the cost of splitting the pathway (12 μs), the cost of sending a header and data out (0.2 μs/word), and the cost of joining the connection (12 μs, provided there is no backlog of messages). The total cost is approximately 25 μs.

selective_receive: The cost of a *selective_receive* is the cost of loading a match CAM (0.6 μs); after the message arrives the overhead of a split (12 μs), the cost of receiving the header and data (0.2 μs/word) and the cost of joining the connection (12 μs). Again, the total cost is approximately 25 μs.

4 Algorithms for iWarp

In this section, we adapt the generic algorithms to the iWarp communication structures and analyze their performance. In Section 4.2 we develop algorithms for the anonymous message passing model which exploit the given topology (unidirectional ring network) whereas Algorithm 2 assumed nothing about the structure of the underlying communication network.

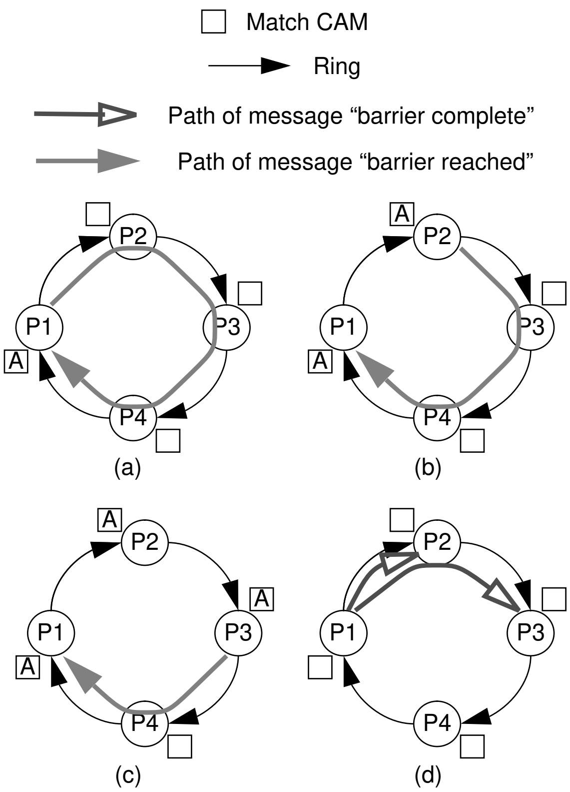
4.1 Broadcast–based barrier synchronization

To adapt Algorithm 1 to the broadcast unit of iWarp, we must only incorporate collision detection and rebroadcasting. The performance analysis is also straightforward; this algorithm takes $O(n)$ steps for one subset barrier with subset size n, and $O(k*n)$ steps for k such barriers (not accounting for rebroadcasting).

4.2 Ring–based barrier synchronization

Algorithm 2 assumes nothing about the structure of the underlying communication network. Here we will develop methods which exploit the topology of the ring network.

Consider the implementation of the *selective_receive* primitive on a ring. The example execution of the barrier algorithm in Figure 3 shows a simple case where the same processor ($P1$) performs a *selective_receive* of all messages. In general however any processor of the subset can receive such messages after reaching the barrier (e.g., if steps (c) and (b) are interchanged in Figure 3, $P3$ receives the "reached A" message of $P2$). So any such message must be passed on, unless the receiving process originally sent the message. This guarantees that the processor that reaches a barrier first can *selective_receive* a message from *each* participating processor.



3–processor barrier A involving processors $P1$, $P2$, $P3$. (a) When $P1$ reaches barrier A, it sets its match CAM to A and sends a message that is subsequently received by itself. (b) When $P2$ reaches barrier A, it sets its match CAM and sends a message that is received by $P1$. (c) When $P3$ reaches barrier A, it sets its match CAM and sends a message which is also received by $P1$. (d) $P1$ has now received 3 messages and therefore completes Phase 1. A *selective_broadcast* informs the other processors; $P1$ resets its match CAM and continues its computation. $P2$ and $P3$ receive the "complete" message, reset their match CAMs, and then continue their computations.

Figure 3: Example of a barrier synchronization in the anonymous destination message passing ring

Implementing the *selective_broadcast* on a ring is easy because we already know that every participating processor is waiting for a message. The processor that ends Phase 1 sends a "barrier complete" message, and all processors receiving the message forward it (unless they are the original sender) and then continue their computations.

The running time of this algorithm is proportional to the number of sequential receiving and sending steps. If n is the number of participating processors, then the number of sequential receiving steps for Phase 1 is n (one processor has to receive n messages). In Phase 2 one message has to be forwarded sequentially around the entire ring, which costs n steps. The drawback of this method is that it unnecessarily forwards messages in Phase 1 and does not overlap the receiving of messages on the different processors in Phases 1 and 2. We will now discuss three ring–based algorithms that improve on this naive approach.

RING1

This algorithm eliminates the forwarding of messages in Phase 1 by using a tournament or combining approach. The processor that wins the tournament is the processor that ends Phase 1. Note that standard tree contraction algorithms are inappropriate here because they require bidirectional communication, which is not allowed in our model and is hard to provide for the barrier synchronization, given the limited number of logical channels available. In our tournament approach, a processor has two possibilities after receiving a message: (1) it continues to receive messages (wins the round) or (2) it forwards a message and waits for notification from the winning processor (loses the round). We choose the processor that ends Phase 1 to be the one with the highest Id (each processor has a unique integer Id) among those processors participating in the barrier. This means that a processor is only allowed to continue receiving messages as long as it does not know about another participating processor with a higher valued Id. Also, we must preserve the information about how many messages have been received by a processor. So in each message we include a counter that accumulates the number of messages the losing processors have received. A detailed outline of the tournament approach can be found in Algorithm 3 (Phase 1).

RING2

This algorithm improves Phase 2 of algorithm RING1 by distributing the completion notification efficiently (in some tree–like fashion). The tree from Phase 1 cannot be reused because of the unidirectional communication network and the highly imbalanced nature of the tree (see e.g. Figure 3). Rather, the processor that finishes Phase 1, denoted P_l, determines the set of participating processors over the course of Phase 1. During Phase 2, it sends one message to the nearest participating processor, and one message to the participating processor halfway around the ring. To apply this approach recursively, the messages sent by P_l must include the set of participating processors. This requires extra work and means that the message length will be linear in the number of processors instead of logarithmic. In reality this works fine for small systems but may create problems for large systems.

RING3

This algorithm gives an alternative approach to Phase 2 with a logarithmic message length, but with potentially decreased performance. During Phase 2, processors are divided into classes. Each processor with Id k belongs to class $c(k) = \lceil \log N \rceil - \lceil \log(N - k) \rceil$. First, processor k (except P_l) selectively receives any message sent to class $l \leq c(k)$. Next, processor k (including P_l) selectively sends a message to all classes $l \leq c(k)$. Finally processor k selectively receives messages sent to classes $l \leq c(k)$ until it receives a message sent to class 0.

Algorithm 3 RING3 (Ring Specific Anonymous Destination Message Passing Barrier)

barrier(b_name, count)

Each processor has a unique Id in the range $\{0, \ldots N-1\}$, where N is the total number of processors. Ids are assigned in bit-reversed order.

b_name.reached and *b_name.class_i* ($i = 0, \ldots$, log(N)) are identifiers generated uniquely from *b_name*.

```
i = 0; class = ⌈log N⌉ − ⌈log(N − Id)⌉;                    /* Phase 1 */
mask_set = {b_name.reached, b_name.class_i (i = 0, ...,
              log (class))}
selective_send (b_name.reached, (Id, 0));
do
   selective_receive(mask_set, name_rec, (Id_rec, count_rec));
   if (name_rec == b_name.class_j)                          /* for any j */
      break;
   i += count_rec;
   if (Id_rec < Id)
      i++;
   if (Id_rec == Id)
      if (i == count − 1)
         break;
      else                                                  /* polling */
         selective_send (b_name.reached, (Id_rec, 0));
   if (Id_rec > Id)
      selective_send (b_name.reached, Id_rec, i);
      break;
until (0);
mask_set = mask_set \ {b_name.reached};                      /* Phase 2 */
if (i == count − 1)                                         /* barrier complete */
   for (i = class, i ≥ 0, i−−)
      selective_send (b_name.class_i, Id, 0);
   do
      selective_receive(mask_set, header_rec, (Id_rec, count_rec));
   while (header_rec != b_name.class_0)
else
   selective_receive(mask_set, header_rec, (Id_rec, count_rec));
   for (i = class, i ≥ 0, i−−)
      selective_send (b_name.class_i, Id, 0);
   while (header_rec != b_name.class_0)
      selective_receive (mask_set, header_rec, (Id_rec,
                count_rec));
```

4.3 Performance analysis

Let N be the total number of processors and n be the number of processors participating in one barrier. We analyze two different costs: (1) the length of the timespan from the moment the last processor reaches the barrier to the moment all processors have left the barrier, and (2) the number of messages as well as the message complexity (total number of wires traveled during the entire barrier synchronization algorithm).

In summary the time complexity of the barrier synchronization is in all common cases $O(\log N)$, which in general is the best we can hope for. The worst case complexity of Phase 1 is $O(\sqrt{N})$, which is provably the best we can expect for this tournament approach. Note that the time complexity of Phase 1 is only dependent on the processors still in the tournament.

In Phase 2 we point out a tradeoff between simplicity, time complexity and message length. The simplest algorithm RING1 has time complexity $O(n)$. The fastest algorithm RING2 has a time complexity $O(\log n)$, but its disadvantage is that it requires messages of length $O(n)$. Algorithm RING3 offers a compromise by having an expected running time of $O(\log^2 n)$, using a random distribution, and sending messages of length $O(\log N)$.

4.3.1 Time complexity of Phase 1

The cost of transmitting a message along the ring is small compared to the cost of a *selective_receive* or a *selective_send*. So we are concerned only with the number of *selective_receive* and *selective_send* operations.

A clear lower bound on the time complexity for this problem is $\log n$ (because it is impossible to compute the "or" of n numbers in time $o(\log n)$).

Note that the time complexity of Phase 1 is only dependent on which processors are still participating in the tournament approach. More exactly: If, during Phase 1, the last l of the n participating processors reach the barrier at time t when $n - k$ processors of the other $n - l$ processor have already been eliminated, then the time complexity of this barrier is the same as if those k processors had reached the barrier at time t. This implies that an n–processor barrier where only one processor reaches the barrier late finishes Phase 1 in time proportional to a 2–processor barrier, which is a desirable property.

The time complexity is proportional to the number of rounds in our tournament. (Each round costs constant time.) In the following we will show that the number of rounds that may be necessary is bounded by the length of the longest increasing subsequence[1] inc and the longest decreasing subsequence dec of Ids along the ring. Then we will show that the length of the longest subsequence of the usual bit-reversal numbering scheme is $O(\sqrt{N})$, which proves that the performance of Phase 1 is $O(\sqrt{N})$.

Note the following simple observations:

- a message from processor i is received by processor j if the Id of processor j is the first Id larger than the Id of processor i.
- a message from processor i eliminates processor j if the Id of processor i is the first Id larger than the Id of processor j.
- before a processor l can be eliminated by the message sent by k, it first must finish all operations that involve the processors between k and l.
- before a processor l can destroy the message sent by k, it first must finish all operations that involve the processors between k and l.

So if one associates every operation (receiving a message and eliminating a processor) with its open interval (k, l), then two operations can execute in parallel if and only if the two intervals associated with them are disjoint. Using these observations we define a conflict tree T_c which is actually an interval graph. The nodes represent the receiving of a message (type a) or the elimination of a processor (type b), and an edge represents a conflict of the two operations. Figure 4 gives an example for such a conflict tree T_c for a barrier in an $N = 16$ processor system involving $n = 8$ processors. The participating processors are at ring positions: $0, 1, 2, 3, 4, 5, 6, 7$ and have the Ids: $0, 8, 4, 12, 2, 10, 6, 14$.

The number of rounds can then be bounded by the length of the longest path in T_c. The length of the longest path in T_c can be bounded by noting that for two nodes along this path the respective intervals have to be subsets of each other, and that for nodes of type (a) the Ids of the processors have to form a decreasing subsequence, and for nodes of type (b) they have to form an increasing subsequence.

[1] A subsequence of a list of numbers is an arbitrary subset of the numbers kept in the same order as the original list of numbers.

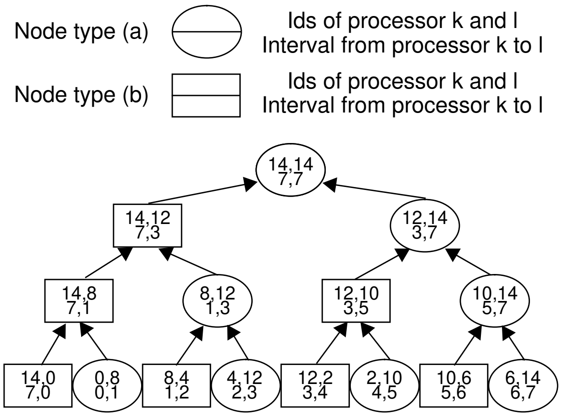


Figure 4: Example of a conflict tree T_c for an 8 processor barrier ($N = 16$). The participating processors are: $0, 1, 2, 3, 4, 5, 6, 7$.

Lemma 4 *Let $id_1, id_2, \ldots, id_N$ be the distinct Ids of the N processors. Let inc be the length of the longest decreasing subsequence and dec be the length of the longest decreasing subsequence of the list of Ids. Then the number of rounds of Phase 1 for an n processor barrier $p_1, \ldots p_n$ is bounded by*

$$\min\{inc + dec, n + 1\}.$$

Proof: For a detailed proof see [7]. □

Unfortunately the longest decreasing or the longest increasing subsequence of any list of N numbers is at least $\sqrt{N}$ (see e.g. [5]). So the best we can hope for is to find a numbering scheme where the longest subsequence is $O(\sqrt{N})$. But we would also like to find a numbering scheme that for commonly used subset barriers gives us a logarithmic number of rounds. Choosing the bit-reversal numbering scheme we get a worst case performance of $O(\sqrt{N})$ and for commonly used subsets an $O(\log N)$ performance.

The bit-reversal numbering scheme (bitrev) is defined as follows: Let x be the processor at the x^{th} ring position (numbered from zero). Then the Id of processor x will be the $bitrev(x)$ with:

$$bitrev(x = x_1, x_2, \ldots, x_{k-1}, x_k) = \overline{x} \quad (\overline{x} = x_k, x_{i-1}, \ldots, x2, x1)$$

The bit-reversal numbering scheme for 16 processors is:

0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15

Lemma 5 *For any subset barrier with n participating processors there exists a numbering scheme such that the number of rounds of Phase 1 is at most:* $\min\{O(\sqrt{N}), n + 1\}$.

Proof: We will only give a proof sketch here. The detailed proof can be found in [7].

Let us consider the bit-reversal numbering scheme. If only n processors participate in a barrier, $inc + dec$ can be at most $n + 1$. So it is sufficient to show that the longest subsequence of the bit-reversal numbering scheme are bounded by $2 * \sqrt{N}$. The longest increasing subsequence can be generated as follows: each member of the subsequence has $\log N$ bits (assume $\log N = 2 * l + 1$). The first $l + 1$ bits form the sequence $\{0, 1, 2, 3, \ldots\}$, and the remaining bits are a mirror image of the first half. Notice that each member of this sequence is its own bit-reverse. □

Lemma 5 gives us a worst case analysis. Consider some common cases of subset barrier synchronization: (a) the processors form a contiguous sequence along the ring, and (b) the set S of participating processors includes every 2^kth processor. (For case (b) let T be the maximal number of rounds necessary for any subset between those processors; e. g. for $k = 3$ we consider the subsets $S \bigcap \{1, 2, \ldots, 7\}$, $S \bigcap \{9, 10, \ldots, 15\}$, etc.)

Lemma 6 *If the subset of processors builds a contiguous sequence of the ring, then the number of rounds is at most $O(\log n)$.*

If the subset of processors S includes every $(2^k)^{th}$ processor, and T is the maximum number of rounds necessary for all subsets between those processors, then the number of rounds is at most $\log N - k + T$.

Proof: Note that the bit-reversal numbering scheme for N processors, where N is a power of 2, implies the following nice properties:

$$\begin{aligned} bitrev(N-1) &= N-1 \\ bitrev(N/2-1) &= N-2 \end{aligned}$$

and furthermore that the subsequence from position 0 to $N/2 - 1$ and the subsequence from position $N/2$ to $N - 1$ use again the bit-reversal numbering scheme of $N/2$ processors. The first subsequence is obtained by multiplying the sequence for $N/2$ processors by 2, and the second is obtained by multiplying by 2 and adding 1. This means that we can prove the logarithmic bounds by induction over the height of the tree. (The size of each interval grows by a multiple of 2 for each step along a path in the conflict graph. See also Figure 4) □

4.3.2 Time complexity of Phase 2

The time complexity of Phase 2 of RING1 is $O(n)$, since the participating processors are notified one after each other. The time complexity of Phase 2 of RING2 is $O(\log n)$, since in every round the number of processors that must be notified by a processor is halved. The analysis of the time complexity of Phase 2 of RING3 is similar to the analysis of Phase 1 (see [7]). However just bounding the longest increasing (decreasing) subsequence is not sufficient any more because we cannot use unique names any more. But we still get a logarithmic bound if the subsets are chosen nicely. Randomization actually helps in this case. If every processor chooses its class randomly according to a certain distribution, we can bound the expected number of rounds by $O(\log^2 n)$.

To summarize, we give the total time complexities for the different algorithms:

a. Worst case analysis.

b. All processors except one have reached the barrier a sufficiently long time before the last processor reaches the barrier.

c. The subset of processors forms a contiguous sequence along the ring.

d. Every 2^kth processor is participating in the barrier and T is defined as above.

Note that (a) and(b) are worst case analyses which are independent of the subset of processors, and (c) and (d) are the most common cases of subset barrier synchronization.

Theorem 7 *This table shows the time complexities for an n processor barrier on an N processor machine for each algorithm.*

	RING1	*RING2*	*RING3*	*RING3 (random)*
a	$O(n)$	$O(\sqrt{N})$	$O(n)$	$O(\sqrt{N} + \log^2 n)$
b	$O(n)$	$O(\log n)$	$O(n)$	$O(\log^2 n)$
c	$O(n)$	$O(\log N)$	$O(\log N)$	$O(\log N + \log^2 n)$
d	$O(n)$	$O(T + \log N)$	$O(n)$	$O(\log N + \log^2 n)$

Proof: See [7] □

4.3.3 Time complexity of h simultaneous barriers

To determine the **time complexity** of h *simultaneous* barriers, we have to refine the proof methods we used to prove the results for the single barrier cases. See [7] for the details. Obviously the number of rounds is bounded by the sum of the rounds of the separate barriers. Note that if the barriers work on separate pieces of the ring the barriers interfere only minimally with each other. This is a nice feature of practical importance.

4.3.4 Number of messages and message complexity

A obvious lower bound on the number of messages is the number N of participating processors, because every processor needs to send at least an initial message.

To find a lower bound on the message complexity of Phase 1 we assume that every processor sends at least one message and this message can only be received by processors which have already reached the barrier.

Lemma 8 *A lower bound for the message complexity of an n processor barrier for Phase 1 is $\Omega(n * N)$.*

Proof: Consider a barrier with n participating processors $P_1, \ldots, P_n$ (numbered in ring order). Processor P_1 reaches the barrier first. No other processor has reached the barrier, so its message travels once around the ring. Processor P_2 reaches the barrier and sends its message. It travels until it reaches processor P_1, i. e. $N - 1$ wires. When processor P_i reaches the barrier, its message traverses $N - i$ wires. So the total message complexity must be at least $n * N - n^2/2 - n/2$. (For an example of such a situation see Figure 3.) □

To bound the number of messages and the message complexity of Phase 1 of our algorithm we observe that every processor is sending one message initially, and every other message which is sent is only being forwarded. Any processor forwards a message only once, and all but one of the messages travel at most one time around the ring.

Note also that the number of messages sent and the way messages are sent only depends on how many processors participate in a certain barrier, and is independent of the number of simultaneous barriers.

So the number of messages of Phase 1 of our algorithm is $2*n$ plus an additive term which is necessary to reduce the time complexity, and the message complexity of Phase 1 of our algorithm is at most $n * N$ plus an equivalent additive term. The additive term depends on the length of the timespan between when the first processor reaches the barrier and when the last processor reaches the barrier; i. e. how often the processor with the current highest Id has to do polling. If the timespan is sufficiently small, the step marked *polling* of Algorithm 3 will never be executed and the message complexity is $n * N$.

The number of messages of Phase 2 for RING1 is 1, so the message complexity of Phase 2 for RING1 is N. This is optimal.

For RING2 we send $\log n$ messages and have a message complexity of $N * \log n$; and for RING3 we send up to $\log N$ messages and have a message complexity of up to $N * \log N$, which is the price we pay for an improved time complexity.

Theorem 9 *Let the timespan between the first processor's and the last processor's arrival at the barrier be t, the time for a message to travel once around the ring be t_r, and the time to receive and send be t_s. Then the message complexity of RING1 for an n processor barrier is at most:*

$$N\left(n+1+\frac{t}{t_r+t_s}\right)$$

and for algorithms RING2 and RING3 the message complexity it is at most:

$$N\left(n+\log N+\frac{t}{t_r+t_s}\right)$$

Note that this is close to optimal.

5 Performance measurements and evaluation

We measured the execution times of different implementations of barrier synchronization on iWarp[2]. All coding (except the bit manipulation code of RING2) was done in C, and the programs were compiled with release 2.4.1 of the C compiler (this is a pre-release without the optimizer.) All data are μs (based on 20 Mhz clock of the production systems).

Figure 5 shows the time for a *single* barrier, as a function of the number of participating processors, if all processors reach the barrier at the same time.

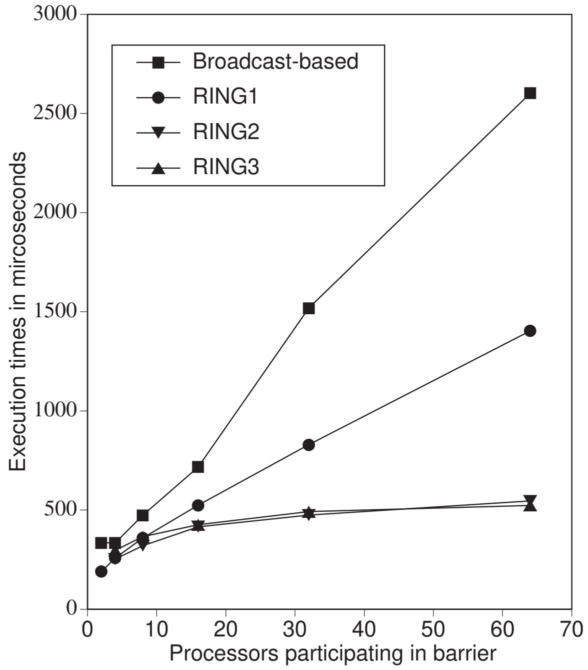


Figure 5: Execution times for different barrier synchronization methods.

Notice that the times for RING2 and RING3 grow logarithmically, while the times for RING1 and for the bounded buffer version grow linearly. This is desirable and allows graceful scaling up of this form of barrier synchronization. It also confirms that the analysis in Section 4.3.1 and 4.3.4 is correct since the observed execution times match the predictions. This means that the decision to base the analysis of the time complexity only on the cost of the *selective_receive* and *selective send* primitives is justified. If the discrepancy between the time to transmit a message along the wires and the cost of the communication primitives gets smaller, the tradeoffs between the algorithms may change (especially for Phase 2). But as long as one considers the same network the same design principles will still apply. We expect that one would still use the tournament approach for Phase 1 because it minimizes the timespan between when the last processor reaches the barrier and the end of Phase 1. For Phase 2 one would choose RING1 because of its optimal message complexity.

Notice that subset barriers incur a cost. For comparison, we coded a fast total barrier using logical channels. Such a total barrier can be implemented on a 64–processor iWarp system in about 60 μs using a ring, and in about 10 μs if more logical channels are used (to build a higher order network). But once we are committed to do subset barriers we can determine the information normally provided by the bitmasks (e.g. the processor that sends the "barrier complete" message) at no additional cost.

Of course the real value of subset barriers is that multiple barriers can be active in parallel. We measured the cost of performing $k = 64/2^i$ barriers in parallel, each with 2^i processors, for $i = 1, 2, \ldots, 6$. On the ring, performing multiple barriers in parallel added (for the considered subsets) at most 25% to the total execution time, whereas the time for the bounded buffer version grows linearly with the number of barriers. In general the ring–based synchronization algorithms are about 2–3 faster than the broadcast–based synchronization algorithm. There is some variation depending on how processors are grouped into subsets; there is less interference between the different subsets if each subset happens to be a contiguous segment of the embedded ring.

If all but one processor has reached the barrier when the last processor arrives, our measurements verify that the only cost we incur is the cost to end Phase 1 and to perform Phase 2, which in broadcast–based methods are two broadcasts (125 μs) and, in the ring–based methods one message plus the cost for Phase 2 (305 μs).

The above comparison between broadcast–based and ring-based synchronization fails to draw attention to one major advantage of the broadcast–based schemes: All participating processors leave Phase 2 at the same time, and this feature can be used to synchronize the local timers on all processors to within 800 ns (the timer resolution is 400 ns). This is an extremely valuable feature which has simplified our measurement tasks significantly.

6 Concluding remarks

Flexible barrier synchronization allows the synchronization of sets of processors, where arbitrary disjoint subsets of processors can synchronize independently and in parallel. We introduced two general communication models that capture important capabilities of modern private–memory systems: the *bounded buffer broadcast model*, based on a global broadcast network, and the *anonymous destination message passing model*, based on a shared general–purpose communication network. These models are simple enough to guide the design of algorithms for subset barrier synchronization, but also provide, based on micro-measurements of a few communication operations, an accurate cost model. We implemented the algorithms

[2]We used a prototype 64–processor system at Carnegie Mellon that runs at half-speed.

on iWarp, a high-performance, commercial parallel system. The measured execution times matched the estimates derived form the model.

Subset barrier synchronization does not have to be expensive. We found an algorithm for a (logical) ring of processors with a time complexity of $O(\sqrt{N})$, rather than the $O(N)$ running time we would initially expect. Furthermore, the algorithms presented here are reasonable: only those processors that participate in a barrier incur any synchronization overhead. Processors that do not participate incur no overhead. The implementation on iWarp preserves this important property of the algorithm by using the appropriate hardware structures (e.g. the match CAM) in a novel way. Even though arbitrary subsets of processors are allowed to synchronize, with only the name and number of processors involved in each barrier known at compile time, this implementation does not impose any overhead on processors that do not participate. This feature makes this design attractive for systems with large numbers of processors, since we cannot allow a small subset of synchronizing processors to impact the rest of the processors.

Efficient barrier synchronization is typically associated with dedicated global hardware resources, such as a broadcast bus. However, the ring-based barriers, which use a general–purpose communication system, consistently outperform the broadcast–based barriers, which used a global broadcast network. This leads us to the conclusion that efficient and flexible barrier synchronization can be supported by a general–purpose communication system, provided sufficient care is taken to ensure fast message transfers between processors and fast direct access to the communication system on each processor.

Acknowledgments

We liked to thank Thomas Warfel and the whole iWarp group for their help with the implementation, Jiří Sgall for many helpful discussions and Jonathan R. Shewchuk and James Stichnot for helpful comments on earlier versions of this paper.

References

[1] ARENSTORF, N., AND JORDAN, H. Comparing barrier algorithms. *Parallel Computing 12*, 2 (1989), 157–170.

[2] BECKMANN, C., AND POLYCHRONOPOULOS, C. Fast barrier synchroniztion hardware. Tech. Rep. ILLS-986, University of Illinois at Urbana-Champaign. Center for Supercomputing Research and Development, November 1990.

[3] BORKAR, S. ET. AL. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88* (Nov. 1988), pp. 330–339.

[4] BORKAR, S. ET. AL. Supporting systolic and memory communication in iWarp. In *17th Annual International Symposium on Computer Architecture* (May 1990), IEEE Computer Society and ACM.

[5] BOSE, R. *Introduction to combinatorial theory*. Wiley series in probability and mathematical statistics. Probability and mathematical statistics. New York : Wiley, 1984.

[6] BROOKS, E. The butterfly barrier. *International Journal Parallel Programming 15*, 4 (1987), 295–307.

[7] FELDMANN, A. Subset barrier synchronization: Communication models and implementation. Tech. Rep. CMU-CS-92-136, School of Computer Science, Carnegie Mellon University, 1992.

[8] GHOSE, K., AND CHEN, D.-C. Efficient synchronization schemes for large-scale shared-memory multiprocessors. In *Proceedings of the 1991 International Conference on Parallel Processing* (August 1991), pp. 153–160.

[9] GUPTA, R. The fuzzy barrier: A mechanism for the high speed synchronization of processors. In *Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (April 1989), pp. 54–63.

[10] HENSGEN, D., FINKEL, R., AND MANBER, U. Two algorithms for barrier synchronization. *International Journal on Parallel Programming 17*, 1 (1988), 1–17.

[11] HWANG, K., AND SHANG, S. Wired-nor barrier synchronization for designing large shared-memory multiprocessors. In *Proceedings of the 1991 International Conference on Parallel Processing* (August 1991), pp. 171–175.

[12] LUBACHEVSKY, B. Synchronization barrier and related tools for shared memory parallel programming. *International Journal of Parallel Processing 19*, 3 (1990), 225–250.

[13] LUNDSTROM, S. Applications considerations in the system design of highly concurrent multiprocessors. *IEEE Transactions on Computers* (November 1987), 1292–1309.

[14] O'KEEFE, M., AND DIETZ, H. Barrier MIMD architecture: Design and compilation. Tech. Rep. TR-EE 90-50, School of Electrical Engineering Purdue University, August 1990.

[15] SCOTT, M., AND MELLOR-CRUMMEY, J. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (Feb 1991), 21–65.

[16] STRICKER, T.M. Supporting the hypercube programming model on mesh architectures (a fast sorter for iWarp). In *1992 ACM Symposium on Parallel Algorithms and Architectures* (San Diego, CA, July 1992).