This paper was jointly awarded the Machtey Award for the most outstanding paper written by a student or students as judged by the program committee, at the 28th Annual Symposium on Foundations of Computer Science, Oct 12-14, 1987.

Yale University
Department of Computer Science

How to Emulate Shared Memory

Abhiram G. Ranade

YALEU/DCS/TR-578
November 1987

This work has in part been supported by the Office of Naval Research under contract N00014-86-K-0564. Approved for public release: distribution is unlimited.

How to Emulate Shared Memory

(Preliminary Version)

Abhiram G. Ranade
Dept. Computer Science
Yale University
New Haven, CT 06520

Abstract—

We present a simple algorithm for emulating an N processor CRCW PRAM on an N node butterfly. Each step of the PRAM is emulated in time $O(\log N)$ with high probability, using FIFO queues of size $O(1)$ at each node. The only use of randomization is in selecting a hash function to distribute the shared address space of the PRAM onto the nodes of the butterfly. The routing itself is both deterministic and oblivious, and messages are combined without the use of associative memories or explicit sorting. As a corollary we improve the result of Pippenger [8] by routing permutations with bounded queues in logarithmic time, without the possibility of deadlock. Besides being optimal, our algorithm has the advantage of extreme simplicity and is readily suited for use in practice.

1 Introduction

Concurrent-read concurrent-write parallel random access machines (CRCW PRAM) allow an arbitrary number of processors to read or write a common memory location in one time step. Complex communications operations, such as broadcast and multicast for example, can be implemented in one step. The facility to succinctly express complex communication patterns greatly simplifies the task of both designing algorithms and writing programs. For this reason, the CRCW PRAM model is favored over weaker

abstract models for which most, if not all, of the algorithmic and programming effort is spent synchronizing the movement of data.

Unfortunately, it is unlikely that CRCW PRAMs will ever faithfully model any real parallel machine. Any real parallel computer will most likely consist of a large number of small processors, each connected to a small number of other processors. For the network to scale in size, we require that the complexity of the individual processors be independent of the size (number of processors) of the network. More specifically, by a realistic parallel computer we mean a network of N processors, with each processor connected to no more than a fixed number (say 4) of processors. Each processor in this network has its own local memory, and processors communicate by sending messages over links to neighboring processors. Finally, each processor can accomodate only a fixed (constant, independent of N) number of messages at any time.

How can we reconcile the convenience of CRCW PRAMs with the limitations of a real computer? The only alternative is to emulate a CRCW PRAM on a real network. Such an emulation has two components:

- **Address Map** — Mapping the address space of the PRAM onto the N memory modules of the network, and
- **Message routing** — Routing memory requests (read/write) from processors to distant memory locations, and data from the location back to the processors. Once we have fixed our address map, each memory access is accomplished by sending a message from the processor requesting the access to the processor holding the memory location.

Three measures determine the efficiency of an emulation: *time,* the number of steps on the network to emulate one step of the PRAM. Second, *queue-size,* the amount of additional hardware per processor required to hold messages in queues while in transit. The third factor is the complexity of managing the queues at each processor: a first-in first-out queue is less complicated than a priority queue, or a queue requiring associative lookups. A simple queueing strategy is clearly preferable to one requiring complex operations.

How much time must an emulation take? Because the diameter of any bounded-degree network on N nodes must be at least $\Omega(\log N)$, this is

clearly a lower bound on the time to emulate one step of a CRCW PRAM. Under fairly general assumptions, Karlin and Upfal [5], and independently, Alt, Hagerup, Mehlhorn and Preparata [2] show that any deterministic emulation must take time at least $\Omega(\log^2 N/\log\log N)$. For any deterministic routing scheme that is also oblivious (the route of each message is completely determined by the source and destination) Borodin and Hopcroft [3] give a worst-case lower bound of $\Omega(\sqrt{N})$.

A number of emulations have been developed in recent years [2,5,6,11, 12]. The best known deterministic strategy [2] for emulating an N node CRCW PRAM with M shared variables on an N node bounded-degree graph takes time $O(\log^2 N)$. Better time bounds are obtained with randomized routing schemes [1,10,13]. Using random hash functions, a randomized routing strategy and Reif-Valiant [9] probabilistic sorting scheme, Karlin and Upfal [5] presented a probabilistic emulation on an N node butterfly. The time complexity of their emulation is $O(\log N)$ and the queue–size is $O(\log N)$. Their queues are required to be built as priority queues, which are expensive.

This paper presents a probabilistic emulation whose time complexity is $O(\log N)$ and queue–size is $O(1)$. The queues are first-in first-out, the simplest possible. We adapt the random hash functions of Karlin and Upfal [5] for the address map. In contrast, however, our routing scheme on the butterfly is completely deterministic. In fact, the routing scheme is also oblivious, which is rather surprising. Thus our scheme only requires $O(\log^2 N)$ random bits, rather than $O(N\log N)$ as in [5]. Besides being optimal with respect to time complexity, our emulation has the advantage of extreme simplicity.

We also note that this is the first emulation of CRCW PRAMs with bounded queue-size. Pippenger [8] showed how to route permutations on a butterfly with bounded queue–size in $O(\log N)$ time. His scheme allowed a small probability of deadlock. We obtain a deadlock–free solution for permutation routing as a simple corollary of our routing scheme. In contrast to [8], our routing scheme as well as the accompanying analysis is considerably simpler.

To summarize, we state the main result below:

Theorem 1 *One step of an N processor CRCW PRAM can be emulated on an N processor butterfly in time $O(\log N)$ with high probability. The*

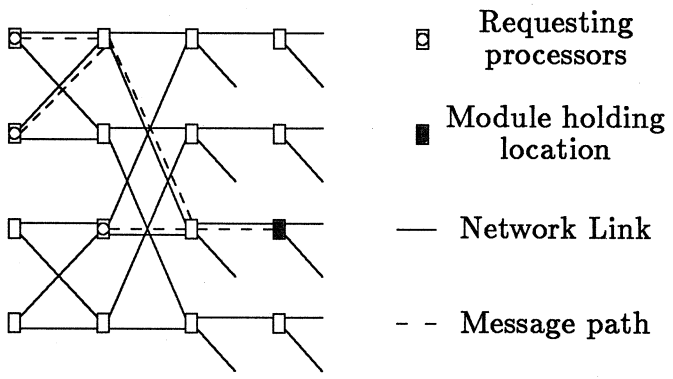


Figure 1: Message paths to a common location form a tree

queue–size at each processor is $O(1)$.

The key idea in this paper is a simple and optimal strategy for combining requests that access the same memory location (section 2). Section 3 presents the emulation. Sections 4 and 5 analyze the routing scheme and the likelihood of large delays.

The address map used in section 3 and in [5] has some drawbacks. Under certain conditions it does not distribute the PRAM address space uniformly over the different memory modules in the butterfly. Further, it only assigns PRAM locations to memory modules, ignoring the problem of where the location is stored *within* the module. We overcome these problems in appendix A. In appendix B we extend our results to emulation of entire PRAM programs.

2 How to combine messages

Suppose that several processors request to read a common memory location.[1] Suppose further that the routes of these messages intersect to form a tree, as in Figure 1. Each message moves along the directed path from its source to the destination. Different messages may, in general, traverse common portions at different times.

[1]Concurrent write requests can be handled similarly, but for the sake of simplicity we will only consider concurrent reads in this paper.

There is, however, no need to send more than one read request along any branch of this tree. If a request simply waits at each tree node until another similar request appears along the other incoming edge (unless it "knows" that future requests along that edge must be for different memory locations), then the two requests can be merged, and one forwarded along the tree. Of course, the reply message must return backwards along each edge of the tree so that each requesting processor receives a reply. To accomplish this we only need, at each node, to store two *direction* bits to direct the reply either along the top branch, the bottom branch, or along both. This simple idea is more efficient than the associative memories discussed by Pfister and Norton [7]. The idea of message combination is not new [4].

How do we know that no future message arriving at a node will request a particular memory location? The key idea is to keep the messages leaving out of each processor sorted by destination. Figure 2 shows a snapshot of processors in a network at some point in time. Each receives messages along two incoming edges and places them into the corresponding FIFO queues. At each step the processor checks the two messages at the head of each queue and compares their destination addresses. The message with the smaller destination address is transmitted along the appropriate outgoing edge, and two direction bits are stored accordingly. If both messages are destined for the same location, one request is sent out. Finally, if only one queue has a message waiting and the other queue is empty, no message is sent out. (If the message were sent, the next message along the other edge could conceivably have a smaller destination, thus violating the sorting requirement).

In our snapshot at time T, processor A in figure 2 selects the message destined for location 35. Then it waits until the message to location 48 arrives, at which point it discovers that the messages at the heads of both the queues are to location 48, and can be combined. Keeping messages sorted by destination also simplifies the task of replicating the reply when it returns (section 3.4).

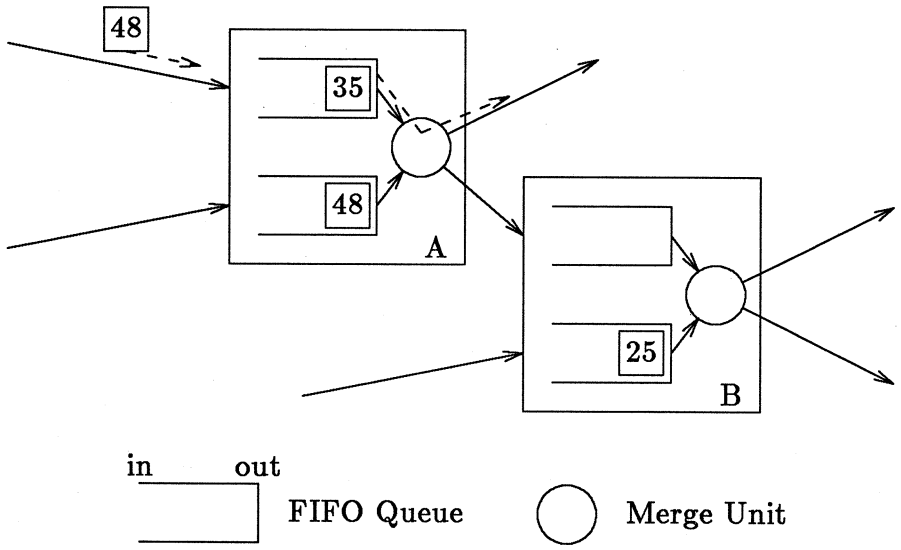


Figure 2: Combining Messages by Merging Streams

2.1 Ghost messages

The simple idea of keeping message streams sorted has one deficiency. Consider Figure 2 again. At time T, processor B cannot transmit the message it holds for location 25, because it must ensure that it will not receive a message to a smaller location in the future. When A selects the message to location 35 for transmission on one link, it can convey this information to B by sending a *ghost* message labelled 35. As soon as B receives the ghost message, it knows that future messages along that edge must be destined for locations greater than 35. Therefore, at the next time step B can forward the message waiting in the lower queue.

Ghost messages simply notify a processor of the minimum location to which subsequent messages can be destined. Ghosts are not used for any other purpose , they "keep the system going." In section 3.5 we specify the mechanisms for transmitting ghosts precisely. This simple idea turns out to be powerful enough to yield our main result.

3 The emulation

The bounded degree network on which we emulate CRCW PRAMs is the *butterfly* (also called the FFT network). The number of nodes in a butterfly with n levels is $N = n2^n$, and we will use this to emulate an N processor PRAM. We assume that levels 0 and n are identified, so that the butterfly is wrapped around. Each node in the butterfly is assigned a unique number $\langle c, r \rangle$ where $0 \leq c < n$, $0 \leq r \leq 2^n - 1$, and $\langle c, r \rangle$ is the binary representation obtained by concatenating the binary representations of c and r. Node $\langle c, r \rangle$ is connected to nodes $\langle c + 1 \bmod n, r \rangle$ and to node $\langle c + 1 \bmod n, r \oplus 2^c \rangle$, where $\oplus$ denotes bitwise exclusive or. Finally, we define $\mathsf{column}\,\langle c, r \rangle = c$ and $\mathsf{row}\,\langle c, r \rangle = r$.

Each node in the butterfly has a processor, a memory module and a small number (6 or 7) of switches, each with upto 2 inputs and 2 outputs. Each input into a switch has a queue that can hold at most b messages. We will specify b later, but it will be a constant, independent of N.

3.1 The Address Map

Suppose that we wish to emulate a CRCW PRAM with N processors and a shared memory of size M, with memory locations $0, \ldots, M-1$. Following [5],[2] we will map shared memory location x into the memory module of the butterfly node whose number is $h(x)$. The function $h(x)$ is chosen at random from the following ζ-universal class of hash functions:

$$H = \left\{ h \mid h(x) = \left(\left(\sum_{0 \leq i < \zeta} a_i x^i \right) \bmod P \right) \bmod N \right\}$$

Each $a_i \in \mathbf{Z}_P$ is chosen randomly, and the number P is a fixed prime no less than the size M of the PRAM address space. The number ζ will be specified later.

3.2 Message structure

In each step of the emulation each processor $\langle c, r \rangle$ accesses a PRAM location, say x, which is placed in memory module $h(x) = \langle c', r' \rangle$. To accom-

[2] Also see appendix A.

plish the memory access, the processor sends a message to module $h(x)$, and the module returns the message to the processor with the required data.

Each message has 3 fields: tag, type, and data. The tag for our message is $\langle h(x), x \rangle$, ie., the number obtained by concatenating the number $h(x)$ of the memory module which contains the shared memory location x, with x itself. The type field is one of REQUEST[3], EOS or GHOST. We assume that each node issues an end-of-stream message of type EOS in the time step right after it issues a memory access message of type REQUEST. The tag field of an end-of-stream message is always ∞. We will keep the messages entering and leaving every switch sorted by the tag field. An end-of-stream message arriving at a switch thus notifies the switch that no more requests will be issued to it from the corresponding incoming edge.

3.3 Message path

Each REQUEST traverses a path from its source processor to the destination module and back. This happens in 6 phases. As seen in Figure 3, each phase is a traversal of the butterfly. In the first three phases, each message traverses the butterfly in the forward direction. In the first phase, the message issued at node $\langle c, r \rangle$ is directed to node $\langle 0, r \rangle$. In Phase 2, the message follows the unique (forward) path in the butterfly from node $\langle 0, r \rangle$ to node $\langle 0, r' \rangle$. This path can be determined by looking at the appropriate bits of the message tag. In Phase 3, the message reaches the node $\langle c', r' \rangle$, where it acquires the required data from the memory module. It continues to move forward through the row until it again reaches node $\langle 0, r' \rangle$.

In the last three phases, the message traverses its path in the reverse direction and returns to the node that initiated the request. The data requested has finally arrived.

For convenience, we describe the routing mechanism in terms of the logical network of Figure 3 instead of the butterfly. The correspondence between the two is clear and each butterfly node does the work of 6 switches in the logical network[4]. The logical network has $(6n + 1)2^n$ switches or-

[3]For simplicity we assume that all requests are read requests. Write requests are handled similarly.

[4]except nodes in column 0, which have 7 switches

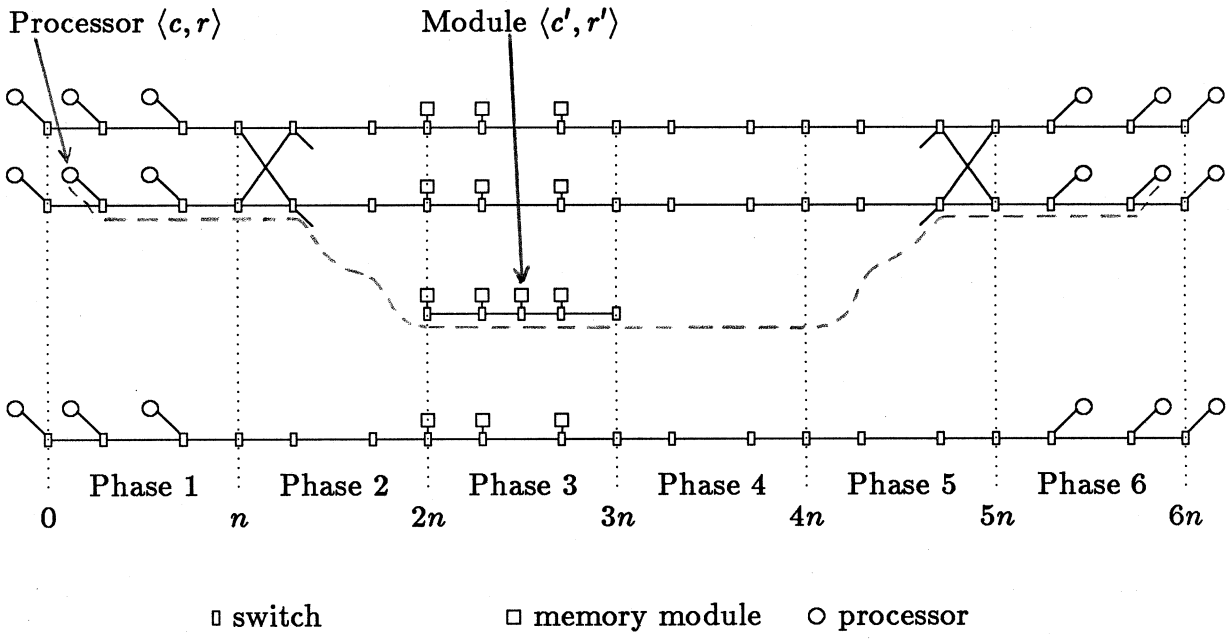


Figure 3: Logical Network

Figure 3: Logical Network

ganized in $6n + 1$ columns. These columns are numbered 0 through $6n$ from the input side, the rows are numbered as in the butterfly. A switch in column c and row r is numbered $\langle c, r \rangle$.

Since there is a unique (forward) path in the butterfly for any pair of nodes, it follows that the path traversed by each request is oblivious (once the address map is fixed).

Lemma 1 *For any processor-memory pair $\langle c, r \rangle, \langle c', r' \rangle$, there is a unique path in the logical network that starts at $\langle c, r \rangle$, passes through $\langle c', r' \rangle$ and ends at $\langle c, r \rangle$. Furthermore, the sequence of nodes traversed in phases 4, 5 and 6 is the reverse of the sequence for phases 3, 2 and 1 respectively.*

3.4 How messages are kept sorted

Using the simple idea of section 2, each switch in the logical network ensures that the messages leaving it are sorted by the tag field. This guarantees that messages to a common memory location are combined as soon as possible.

How do we return the data to all requesting processors? Consider an arbitrary switch s_5 in phase 5. Let s_2 be the phase 2 switch in the same butterfly node as s_5. For each request that passes through s_2, two direction bits are stored, which are used by s_5 to route the reply bearing the data. This can be done because of the crucial observation that replies arrive in s_5 in the same order that requests were sent out of s_2. More precisely:

Lemma 2 *The ith* REQUEST *selected for transmission in a phase 5 switch s_5 is the reply to the ith* REQUEST *selected for transmission in the corresponding phase 2 switch s_2.*

With this observation, it is not necessary to store the identities of the sources, a FIFO queue of the direction bits is adequate. This is also applicable to the switches in phases 1 and 6. Thus, each butterfly node requires additional FIFO queues between the phase 2(1) and 5(6) switches. It will be shown that the total time required by the emulation algorithm is $O(n)$ with high probability, hence queues of $O(n)$ bits are sufficient. This requires no more storage than $O(1)$ messages.

3.5 How ghosts appear and disappear

Suppose that a switch determines that the message m selected for transmission in switch s is required to be transmitted only on one of its outputs. Suppose further that the other output connects s to switch s_1 which has space in its input queue. Then, as described in section 2, a GHOST message is sent to s_1 with its tag equal to the tag of m. Now, switch s_1 is informed that no subsequent messages received from s can have a smaller tag.

This GHOST message might itself be forwarded by s_1 if it has the smallest tag of all the messages at the head of the queues in s_1. If, however, the GHOST is not forwarded immediately, then it need not be retained by s_1. This is because s must send s_1 another message (GHOST or otherwise) at the next step, whose tag cannot to be smaller. The information in the new message is bound to be at least as strong as that in an old GHOST.

Lemma 3 *A* GHOST *will never wait at any switch.*

GHOSTs also help "keep the system going." More precisely, after the first time that any switch sends out a message, it will continue to send out messages (ghosts or otherwise) along all its output edges until the time at which it sends out the end-of-stream message. With this observation, we have the following lemma.

Lemma 4 *Every input queue in column c holds at least one message at time c, and at every subsequent time until the end-of-stream message leaves the queue.*

4 Message polarization and delay

We show that whenever message delivery takes a long time, there exists a long *polarized* sequence of messages. In the next section, we show that long polarized sequences are highly unlikely to occur. The notion of a polarized sequence is similar to that of the delay sequence of [5,10] and the critical path of [1].

A path S in the logical network is a sequence $\{S(i)\}$ of switches with the property that, for every i, the switch $S(i)$ is connected to switch $S(i+1)$, and the switches $S(i), S(i+1)$ are distinct. Note that not all switches along

a path need be distinct, only adjacent pairs. A path which originates in column 0 and ends in column $6n$ is called an *input–output path.*

Definition 1 *Let $M = m_1 m_2 \ldots m_\alpha$ be a sequence of messages such that $\mathsf{tag}(m_i) < \mathsf{tag}(m_{i+1})$ for $1 \leq i < \alpha$. Suppose that S is a path in the network such that during delivery of all messages, message m_i passes through the j_ith switch along S, ie., through switch $S(j_i)$. Suppose, finally, that $j_i \leq j_{i+1}$ for all i, $1 \leq i < \alpha$. Then the message sequence M is said to be polarized along S.*

The main result of this section is the following theorem, which relates the emulation time and queue size to the length of polarized sequences.

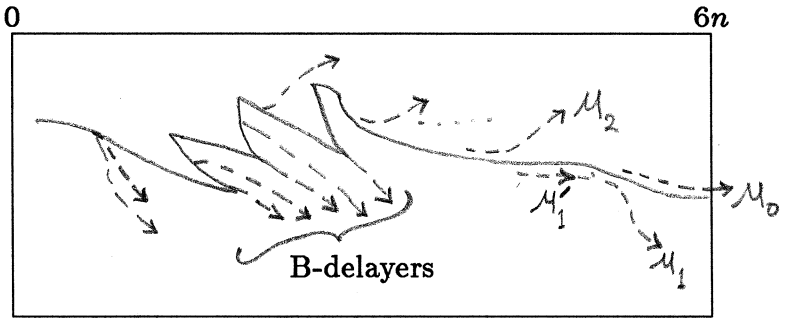
Theorem 2 *Suppose that a set of memory requests takes time $6n + \delta$. If each queue is of size b, then there exists an input–output path S of length $8n$ and a sequence M of $\min(\delta, bn)$ distinct* REQUEST *messages which is polarized along S.*

Before we proceed to a sketch of the proof of this theorem, we need a few definitions that will be used to analyze how messages are delayed in the logical network.

Definition 2 *If message m is in switch s at time t, then the* **lag** *of m at time t, is defined as $\mathsf{lag}(m, t, s) = t - \mathsf{column}(s)$. The lag is an upper bound on the time spent by a message waiting in queues.*

Definition 3 *Suppose that message m_0 entered a switch s at time t_E, and left at time $t_L \geq t_E + 1$. There are three possibilities:*

1. *At time $t_L - 1$, another message m_1 was selected by s for transmission. Then $(m_1, t_L - 1, s)$ is said to* **m-delay** *(m_0, t, s) for $t_E \leq t < t_L$. Note that $\mathsf{lag}(m_1, t_L - 1, s) \geq \mathsf{lag}(m_0, t, s)$.*
2. *At time $t_L - 1$, m_0 was selected for transmission, but not transmitted because the queue which it next entered, say in switch s_1, was full. Since m_0 left s at time t_L s_1 must have had space for it at that time. But this means that a message m_1 must have left s_1 at time $t_L - 1$ to create that empty space. In this case, $(m_1, t_L - 1, s_1)$ is said to* **b-delay** *(m_0, t, s) for $t_E \leq t < t_L$. Note that $\mathsf{lag}(m_1, t_L - 1, s_1) \geq \mathsf{lag}(m_0, t, s) - 1$.*



—— Path S - - - - Message Path

Figure 4: Polarized Sequence

3. *At time $t_L - 1$, no message was selected for transmission because the other queue in s was empty. (This can happen only in Phase 1) By lemma 4* $\mathsf{column}(s) > t_L - 1$, *i.e.* $\mathsf{lag}(m_0, t_L, s) \leq 0$.

4.1 Constructing long polarized sequences

We will first construct a sequence $\{(\mu_i', t_i', \psi_i')\}$ such that $\mathsf{tag}(\mu_i') > \mathsf{tag}(\mu_{i+1}')$. This will give us a path S and a (possibly short) sequence $\{\mu_i'\}$ that is polarized along S. Next, we augment the sequence to get a longer sequence polarized along ths same path S.

To describe how we construct our initial sequence, suppose that a set of memory requests takes time $6n + \delta$. Let μ_0 be an EOS message that arrives in column $6n$ at the last time step $6n + \delta$. Suppose that it arrives at switch ψ_0 (in column $6n$) at time t_0, so that $\mathsf{lag}(\mu_0, t_0, \psi_0) = \delta$. Let us follow μ_0 back in time until it last waited in a queue, say in ψ_0', at time t_0'. Then there is a triple (μ_1, t_1, ψ_1) which delayed (μ_0', t_0', ψ_0') $(\mu_0' = \mu_0)$. Now we follow the path of μ_1 back in time to the point at which it was first delayed, and so on (Figure 4).

In general, given (μ_i, t_i, ψ_i) with $\mathsf{lag}(\mu_i, t_i, \psi_i) > 0$ it is possible to construct $(\mu_{i+1}, t_{i+1}, \psi_{i+1})$ as follows. We follow μ_i back in time starting from t_i. If μ_i is a ghost or a combination and we reach the switch at which μ_i was created, then we continue following one of the messages which caused μ_i to be created. We continue this process until we reach a time at which an ancestor of μ_i was last delayed. Unless $\mathsf{lag}(\mu_i, t_i, \psi_i) \leq 0$, we are certain

to reach some ψ_i' at which some μ_i' (which might be μ_i or an ancestor) is forced to wait at some time t_i'.

Thus there must exist $(\mu_{i+1}, t_{i+1}, \psi_{i+1})$ which delayed (μ_i', t_i', ψ_i'). But there is no waiting between t_i and $t_i' + 1$. Thus we have $\mathsf{lag}(\mu_i, t_i, \psi_i) = \mathsf{lag}(\mu_i', t_i', \psi_i') + 1$.

We repeat this process until we have a message μ_L with $\mathsf{lag}(\mu_L, t_L, \psi_L) = 0$ for some L. Let μ_L' be any ancestor of μ_L at time $t_L' = 0$ and μ_L' be at switch ψ_L' at time 0.

We first estimate L.

Lemma 5 *Let b_i denote the number of b-delayers in the segment (μ_0, t_0, ψ_0) through (μ_i, t_i, ψ_i). Then $L \geq \delta - b_L$.*

Proof: By the construction of the short polarized sequence, $\mathsf{lag}(\mu_i, t_i, \psi_i) = \mathsf{lag}(\mu_i', t_i', \psi_i') + 1$. By definition of delay, $\mathsf{lag}(\mu_{i+1}, t_{i+1}, \psi_{i+1}) \geq \mathsf{lag}(\mu_i', t_i', \psi_i') - x$ where x is 0 or 1 depending upon whether $(\mu_{i+1}, t_{i+1}, \psi_{i+1})$ is an m-delayer or a b-delayer. More specifically, $x = b_{i+1} - b_i$. Therefore we have, $\mathsf{lag}(\mu_{i+1}, t_{i+1}, \psi_{i+1}) \geq \mathsf{lag}(\mu_i', t_i', \psi_i') - (b_{i+1} - b_i)$. Thus, $\mathsf{lag}(\mu_{i+1}, t_{i+1}, \psi_{i+1}) \geq \mathsf{lag}(\mu_i, t_i, \psi_i) - (b_{i+1} - b_i) - 1$. Thus $\mathsf{lag}(\mu_L, t_L, \psi_L) \geq \mathsf{lag}(\mu_0, t_0, \psi_0) - b_L - L$. But $\mathsf{lag}(\mu_L, t_L, \psi_L) = 0$ and $\mathsf{lag}(\mu_0, t_0, \psi_0) = \delta$. Thus the result follows. ∎

Proof Sketch of Theorem 2: Let Γ_i denote the path passing through ψ_0, ψ_0', ψ_1, $\psi_1' \ldots \psi_i$, ψ_i'. Let $|\Gamma_i|$ denote the number of edges in Γ_i.

We shall first identify some Γ_j along which α messages are polarized. We know that for all j $|\Gamma_j| = 6n + 2b_j - \mathsf{column}(\psi_j')$ and we will ensure $b_j \leq n$. Then we can always construct an input–output path Γ of length $8n$ that contains Γ_j. We need to consider two cases.

In the first case, $b_L \leq n$. Assume that (μ_i', t_i', ψ_i') is b-delayed by $(\mu_{i+1}, t_{i+1}, \psi_{i+1})$. Let μ_{i1} through μ_{ib-1} be the other messages present in ψ_{i+1} at t_{i+1}. Clearly, $\mathsf{tag}(\mu_i') > \mathsf{tag}(\mu_{i1}) > \ldots > \mathsf{tag}(\mu_{ib-1}) > \mathsf{tag}(\mu_{i+1}')$. Thus the sequence $\mu_1' \ldots \mu_i', \mu_{i1} \ldots \mu_{ib-1}, \mu_{i+1}' \ldots \mu_L'$ of messages is Γ_L polarized, and has length $L + b - 1$. Similarly, we can include messages that were present in the queues when each of the b_L b-delays occured, giving a Γ_L polarized sequence of length $L + b_L(b-1) \geq \delta + b_L(b-2) \geq \delta$ provided $b - 2 \geq 0$. Thus in this case we choose $j = L$.

In the second case when $b_L > n$, choose j such that $b_j = n$. Then we have at least the b_j b-delayers and $b_j(b-1)$ other message present in the queues at the times of the b-delays i.e. at least bn messages.

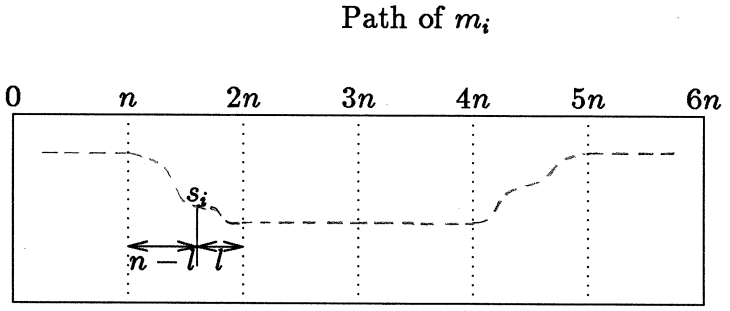


Figure 5: Number of ways of choosing rows of source processor and destination module of message m_i is $2^l 2^{n-l} = 2^n$

We now show that the polarized messages identified above are distinct, and of type REQUEST. Because each (μ_i, t_i, s_i), $i > 0$ delays $(\mu'_{i-1}, t'_{i-1}, s'_{i-1})$, it follows that μ_i cannot be of type EOS. But $\mathsf{tag}(\mu_i) = \mathsf{tag}(\mu'_i)$, thus μ'_i cannot be of type EOS. Also, μ'_i waits, and hence by lemma 3 it cannot be a GHOST. Thus these messages must be REQUESTs. None of the messages waiting in queues can be GHOSTs. All are ahead of other messages, hence none can be EOSs. Distinctness follows because the tags are strictly sorted. ∎

5 Large delays are unlikely

Theorem 3 *For every f, there exists a queue-size b independent of N such that every message is routed in time bn with probability at least $1 - N^{-f}$, with $\varsigma = 8n$.*

Proof: The proof has two steps[5]:
Step A: We first estimate the total number of times arbitrary sequences of ς REQUESTs are T polarized for some input-output path T of length $8n$, over all possible choices of the hash function h. This can be done by counting the choices for T, the choices for switches on T where REQUESTs touch, the choices for touching REQUESTs (i.e. the choice of their source processors, which determines x_i, and the choice of the destination mod-

[5]No attempt has been made here to obtain the smallest value for b, and the value obtained can be substantially improved.

ules which determines $h(x_i)$), and the hash functions consistent with these choices.

1. The path T can originate at any switch in column $6n$, and consists of $8n$ displacements, n of which are forward. Each forward or backward displacement can be along any of two edges. Thus the total number of possible choices is $2^n\binom{8n}{n}2^{8n} = 2^{9n}\binom{8n}{n}$.

2. Let $m_1, \ldots, m_\varsigma$ denote the sequence of messages that is T polarized, with m_i touching T at s_i. All s_i can together be chosen in at most $\binom{8n+\varsigma}{\varsigma}$ ways.

3. Let x_i denote the PRAM location to which message m_i is destined. s_i lies on the path of m_i, hence the row in which m_i originated and the row in which module $h(x_i)$ lies can together be chosen in 2^n ways (figure 5). This is independent of what phase s_i belongs. The column in which m_i originated can be chosen in n ways. Notice that this fixes x_i. Because of the polarization property, $h(x_1) \geq h(x_2) \geq \ldots h(x_\varsigma)$, and hence $\mathsf{column}(h(x_1)) \geq \mathsf{column}(h(x_2)) \geq \ldots \mathsf{column}(h(x_\varsigma))$. But $\mathsf{column}(h(x_i))$ can only take on n different values. Thus $\mathsf{column}(h(x_1))$, $\ldots, \mathsf{column}(h(x_\varsigma))$ can together be chosen in at most $\binom{n+\varsigma}{\varsigma}$ ways. $\mathsf{row}(h(x_i))$ is already known. Thus the total number of choices is $(\prod_{i=1}^{\varsigma} 2^n)(\prod_{i=1}^{\varsigma} n)\binom{n+\varsigma}{\varsigma} = N^\varsigma\binom{n+\varsigma}{\varsigma}$

4. The previous step fixed $h(x_i)$ for ς distinct x_i. We know that for $1 \leq i \leq \varsigma$ and $0 \leq x_i, y_i < P$, there is at most one polynomial p of degree $\varsigma - 1$ over the field $\mathbf{Z}_P$ such that $p(x_i) = y_i$ for all i. Thus in order to keep the above choices of $h(x_i)$ consistent, we need to consider $y_i \equiv h(x_i) \pmod{N}$. Thus each y_i can be chosen in P/N different ways, i.e. there are at most $(P/N)^\varsigma$ different polynomials which allow this.

Therefore the total number of times sequences of ς REQUESTs are polarized is at most

$$2^{9n}\binom{8n}{n}\binom{8n+\varsigma}{\varsigma}N^\varsigma\binom{n+\varsigma}{\varsigma}\left(\frac{P}{N}\right)^\varsigma \leq (6P)^\varsigma\binom{2\varsigma}{\varsigma}$$

assuming $\varsigma = 8n$, and using $\binom{n}{k} \leq (ne/k)^k$.

Step B: Let N_{bn} be the number of hash functions for which the total routing time is at least bn. For each such function it is possible to find $b'n = bn - 6n$ polarized messages. But each subset of size ς of these $b'n$ messages contributes to the above estimation. There are $\binom{b'n}{\varsigma}$ ways of choosing the subset. Thus

$$N_{bn}\binom{b'n}{\varsigma} \leq (6P)^{\varsigma}\binom{2\varsigma}{\varsigma}$$

Because the coefficients a_i can each be chosen in P ways, there are a total of P^{ς} hash functions. Thus the probability of requiring time bn is

$$\frac{N_{bn}}{P^{\varsigma}} \leq \frac{(6P)^{\varsigma}\binom{2\varsigma}{\varsigma}}{P^{\varsigma}\binom{b'n}{\varsigma}} \leq \left(\frac{12\varsigma}{b'n}\right)^{\varsigma} \leq N^{-4\log\frac{b-6}{96}}$$

Thus for arbitrary f, there exists a constant b such that the probability of requiring at least bn time for routing is less than N^{-f}. ∎

5.1 Permutation Routing

These ideas can also be used in routing permutations with constant queue–size [8], and without deadlocks. Suppose processor i wants to send a message to processor $\pi(i)$, where π is a permutation on $\{0,\ldots,N-1\}$. We use the 6 phase scheme discussed above. In the first 3 phases, processor i sends its message to location i. In the last 3 phases, instead of returning to processor i, the message is moved to $\pi(i)$. For all the 6 phases $\langle h(i), i\rangle$ is used as tag. The construction of the polarized sequence and the analysis presented above are applicable with minor modifications.

Corollary 1 *Suppose every processor i is to send a message to processor $\pi(i)$ where π is a permutation. Then for every f, there exists a queue–size b independent of N such that every message is routed in time bn with probability at least $1 - N^{-f}$.*

Acknowledgements

I am immensely grateful to Sandeep Bhatt for his untiring help in writing this abstract. I would also like to thank Lennart Johnssson for many

discussions and support, David Greenberg for critical comments, and Tom Leighton for pointing out an error in an earlier version and for motivating appendix A.

References

[1] R. Aleliunas. Randomized parallel communication. In *PODC*, pages 60–72, 1982.

[2] H. Alt, T. Hagerup, K. Mehlhorn, and F.P. Preparata. Simulation of idealized parallel computers on more realistic ones. Preliminary Report.

[3] A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parallel models of computation. In *Proceedings of STOC 82*, pages 338–344, 1982.

[4] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Coordinating large numbers of processors. In *1981 International conference on Parallel Processing*, 1981.

[5] Anna Karlin and Eli Upfal. Parallel hashing - an efficient implementation of shared memory. In *Proceedings of STOC 86*, 1986.

[6] Kurt Mehlhorn and Uzi Vishkin. *Granularity of Parallel Memories.* Ultracomputer Note 59, NYU, October 1983.

[7] G. F. Pfister and V. A. Norton. Hot-spot contention and combining in multistage interconnection networks. In *Proceedings of Intl. Conf. on Parallel Processing*, pages 790–797, 1985.

[8] Nicholas Pippenger. Parallel communication with limited buffers. In *Proceedings of FOCS 84*, pages 127–136, 1984.

[9] John Reif and Leslie Valiant. *A logarithmic time sort for linear size networks.* Technical Report 36-82, Harvard University, November 1982.

[10] E. Upfal. Efficient schemes for parallel communication. In *PODC*, pages 55–59, 1982.

[11] Eli Upfal. A probabilistic relation between desirable and feasible models of parallel computation. In *STOC*, pages 258–265, 1984.

[12] Eli Upfal and Avi Wigderson. How to share memory in a distributed system. In *FOCS*, pages 171–180, 1984.

[13] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of STOC 81*, pages 263–277, 1981.

Appendices

A Memory module sizes and local addressing

The address map of section 3.1 only specifies the butterfly memory in which a particular PRAM location is to be placed. It does not specify the location within the memory that will hold the address. The map also has the drawback that it might cause too many PRAM locations, as many as $\varsigma = 8n$ to get mapped onto the same memory. This is not acceptable if the size of the shared memory is small, i.e. if $M = kN$, for some constant k. Here we briefly sketch a scheme in which each shared memory location is assigned a unique location in the butterfly memory, and further, the memory required at each node is $O(M/N)$.

Given the polynomial coefficients a_i, define the *hash address* of x to be:

$$a(x) = \left(\left(\sum_{0 \le i < \varsigma} a_i x^i \right) \bmod P \right) \bmod M$$

The hash address $a(x)$ is interpreted as a global address in the butterfly, i.e. it corresponds to location $a(x)/N$ of module $h(x) = a(x) \bmod N$, assuming M is a multiple of N. The function a thus maps the PRAM location x in the range $0, \ldots, M-1$ onto a butterfly address $a(x)$ in the range $0, \ldots, M-1$.

We shall call this region of the butterfly memory the *hash table area.* The butterfly location $a(x)$ cannot in general be used to store the contents of PRAM location x, because as many as $\varsigma = 8n$ PRAM locations may have the same hash address. To handle this, we use the butterfly memory above address $M-1$ as an *overflow area.*

Every location $a(x)$ in the hash table area points to a group of locations in the overflow area. These locations hold the PRAM locations that get mapped into $a(x)$. Each location in the overflow area holds pairs of the form (x, data in PRAM location x). Thus PRAM location x is accessed by searching through the overflow locations associated with the hash table location $a(x)$. We now describe how to allocate overflow area for each hash table location.

A.1 Preliminary Scheme

In this scheme every location $a(x)$ in the hash table is assigned $8n$ locations in the overflow area whether or not there are $8n$ PRAM locations mapped onto it. These $8n$ locations consist of 8 locations from each memory in the row of of the butterfly to which $a(x)$ belongs. Thus if location $a(x)$ is assigned memory locations i through $i+7$ in each memory in its row, then $a(x)$ is set to i.

Figure 6 shows the memory layout for some row in the butterfly, for the case $n = 3$, and $M/N = 2$. Thus locations 2 through 9 are alloted to location 0 of memory M_1, locations 10 through 17 to location 1 of M_1, locations 18 through 25 to location 0 of M_2 and so on. The starred squares indicate the locations that are actually used. The figure shows that 4 shared memory locations are mapped onto location 0 of M_1, 1 onto location 1, and 2 onto location 0 of M_2.

Memory access: In order to access shared memory location x, it is sufficient to search 8 locations in every memory in the row of $a(x)$. Which locations to search is indicated by $a(x)$. This only requires a minor modification to the access scheme described in section 3.3: the location $a(x)$ is read in phase 3; and the search through the overflow area takes place in phase 4. Thus, at each node in phase 4, the 8 locations specified by $a(x)$ are searched. Eventually the message reaches a node that holds PRAM location x, at which point the data field of the message is updated. The

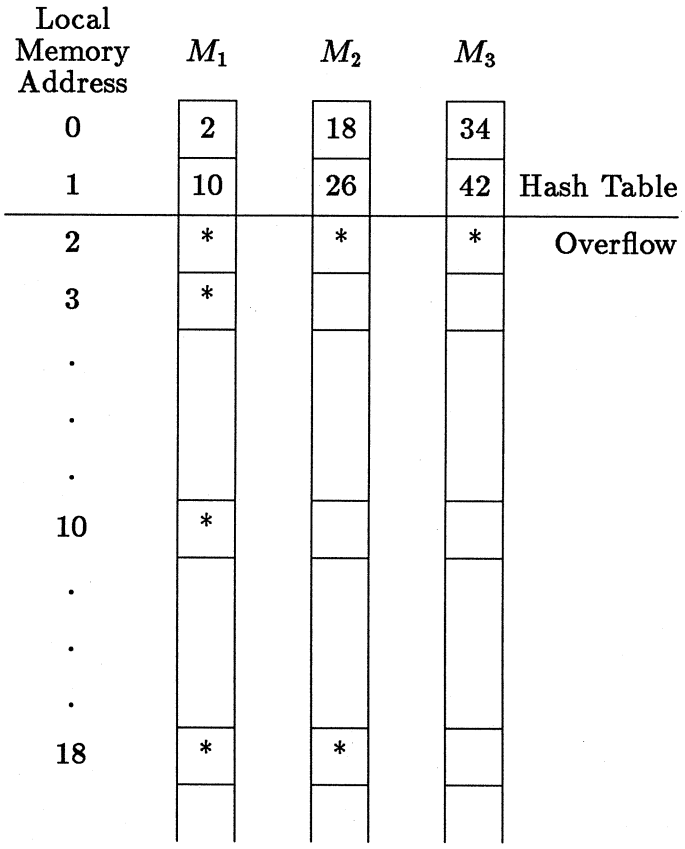


Figure 6: Memory layout for a row in the Butterfly

message movement is not affected, since each memory needs to be accessed a constant number (8) of times per message.

A.2 Improved Scheme

It is possible to show that the empty spaces in the layout of figure 6 can be eliminated. The memory required per row is now proportional to the number of PRAM locations that get mapped into it. This is bounded by the following theorem stated without proof.

Theorem 4 *For any constant f, there exists a constant c (independent of N) such that the probability of mapping more than cMn/N PRAM locations into any row of the butterfly is less than N^{-f}.*

Thus with extremely high probability it is sufficient to have a total memory of $O(Mn/N)$ in every row. Thus $O(M/N)$ memory is sufficient in every module, as desired.

A.3 Address computation

The hash function has $\varsigma = 8n$ coefficients, and just storing all the coefficients on each processor requires $O(n)$ memory. However this is not necessary. The node in column i need only hold coefficients $8i$ through $8i + 7$. The polynomial evaluation can be pipelined, requiring $O(n)$ cycles.

B Emulating sequences of instructions

Our results can be extended to the emulation of multi instruction programs, as done by Karlin and Upfal [5]. For emulating multiple instructions, the same protocol is used, but now we must guard against the possibility that a particular instruction might not complete in the alloted time. If this happens, a new hash function h is chosen, all the variables are sent to their new locations, and the emulation process resumes. We must also consider the time required to initialize the hash table area as described in appendix A.2.

Theorem 5 *Consider an N processor CRCW PRAM with M shared memory locations, M polynomial in N. Then an arbitrary T instruction program, $T \geq M/N$, can be emulated on an N node butterfly in time $O(T \log N)$ with probability tending to 1 as N and/or T tend to ∞. Further the size of the memory required at each butterfly node is $O(M/N)$.*

Proof: (Sketch) Initializing the hash table area and rehashing require time $O(\frac{M}{N} \log N)$ with high probability. Because M is polynomial in N, we see from theorem 3 that any PRAM instruction completes in time $c \log N$ with probability at least $1 - M^f$ for every f and some c independent of M. Thus the interval between successive rehashing operations is at least M^f with probability $\frac{1}{2}$. Thus with high probability no more than $8T/M^f$ rehashing operations are needed. The time required for these is $O((1 + \frac{8T}{M^f})\frac{M}{N} \log N)$. Because $T \geq M/N$, the total emulation time is $O(T \log N)$ with high probability. ∎

The restriction $T \geq M/N$ is not serious because M/N instructions are required just to access all the M locations.