$x = 0.11010110_2 = 0.3112_4$ $y = 0.11010111_2 = 0.3113_4 = 1.\bar{1}12\bar{1}_4$ $w[0] = xY_0$ $Z_0 = 0$ $(s_0 = 0)$

j			Z_j	t_{j-1}	s_j	P_{j-1}	$A[j-1]$	$D[j-1]$
1	$w[0] =$	$3.\ 1\ 1\ 2$						
	$xY_1 =$	$0.\ \bar{3}\ \bar{1}\ \bar{1}\ \bar{2}$						
	$Z_1 + 4^{-1}w[1] =$	$3.\ \bar{2}\ 0\ 1\ \bar{2}$	3	1	-1	1	1.	*u.*
2	$w[1] =$	$\bar{2}.\ 0\ 1\ \bar{2}$						
	$xY_2 =$	$0.\ 3\ 1\ 1\ 2$						
	$Z_2 + 4^{-1}w[2] =$	$\bar{2}.\ 3\ 2\ \bar{1}\ 2$	-2	0	-2	-1	1.3	*d.u*
3	$w[2] =$	$3.\ 2\ \bar{1}\ 2$						
	$xY_3 =$	$1.\ 2\ 2\ 3\ 0$						
	$Z_3 + 4^{-1}w[3] =$	$5.\ 0\ 2\ 1\ 0$	5	1	1	-1	1.33	*d.du*
4	$w[3] =$	$0.\ 2\ 1\ 0$						
	$xY_4 =$	$0.\ \bar{3}\ \bar{1}\ \bar{1}\ \bar{2}$						
	$Z_4 + 4^{-1}w[4] =$	$0.\ \bar{1}\ 0\ \bar{1}\ \bar{2}$	0	0	0	1	1.331	*d.dnu*
5	$w[4] =$	$\bar{1}.\ 0\ \bar{1}\ \bar{2}$	-1	0		0	1.3310	*d.dnuu*

Decrement digits of $A[4]$ according to $D[4]$: $M[4] = 0.2310$;
check: $0.3112 \times 0.3113 = 0.2310 + 0.0000\bar{1}0\bar{1}\bar{2}$.

Fig. 5. Example of LRCF multiplication.

IV. Summary

The reported multiplication scheme (LRCF) eliminates the need for a carry-propagate adder. The scheme performs the multiplication most-significant digit first and produces a conventional sign-and-magnitude product (most significant half) by means of an on-the-fly conversion, performed concurrently with the generation of accumulated (redundant) partial products. The scheme is presented for general radix r and a radix-4 signed-digit implementation is described. We estimate that, for a multiplier of 64 bits, the scheme we described produces a reduction of about ten gate levels with respect to a conventional scheme using a carry-look-ahead adder. The speed can be improved by increasing the radix. In [6], we present a radix-16 implementation in which odd and even partial products [11] are computed concurrently.

References

[1] K. Hwang, *Computer Arithmetic.* New York: Wiley, 1978.

[2] M. Uya, K. Kaneko, and J. Yasui, "A CMOS floating-point multiplier," *IEEE J. Solid-State Circuits*, vol. SC-19, no. 5, pp. 697-701, Oct. 1984.

[3] A. Avizienis, "On a flexible implementation of digital computer arithmetic," in *Information Processing* 1962, C. M. Popplewell, Ed. New York: North Holland, 1963, pp. 664-670.

[4] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, part 2, pp. 236-240, 1951.

[5] M. D. Ercegovac and T. Lang, "On-the-fly conversion of redundant into conventional representations," *IEEE Trans. Comput.*, vol. C-36, no. 7, pp. 895-897, July 1987.

[6] —, "Fast multiplication without carry-propagate addition," UCLA Comput. Sci. Dep. Rep., 1986.

[7] A. Avizienis, "Signed-digit number representation for fast parallel arithmetic," *IEEE Trans. Electron. Comput.*, vol. EC-10, pp. 389-400, Sept. 1961.

[8] J. T. Coonen, "An implementation guide to a proposed standard for floating-point arithmetic," *IEEE Comput. Mag.*, pp. 68-79, Jan. 1980.

[9] Annon, "Cray X-MP Computer Systems," *Four-Processor Mainframe Reference Manual*, HR-0097, Cray Research, Inc., 1985.

[10] M. D. Ercegovac and T. Lang, "Alternative on-the-fly conversion of redundant into conventional representations," UCLA Comput. Sci. Dep. Rep. CSD-860027, Nov. 1986.

[11] J. Iwamura *et al.*, "A 16-bit CMOS/SOS multiplier-accumulator," in *Proc. ICCC82*, 1982, pp. 151-154.

[12] S. Kuninobu *et al.*, "Design of high-speed MOS multiplier and divider using redundant binary representation," in *Proc. 8th. Symp. Comput. Arithmet.*, 1987, pp. 80-86.

[13] Y. Harata *et al.*, "High-speed multiplier using a redundant binary adder tree," in *Proc. 1984 IEEE Int. Conf. Comput. Design*, 1984, pp. 165-170.

[14] J. E. Robertson, "A systematic approach to the design of structures for arithmetic," in *Proc. 5th Symp. Comput. Arithmet.*, 1981.

[15] M. D. Ercegovac and T. Lang, "Radix-4 multiplication without carry-propagate addition," in *Proc. IEEE Int. Conf. Comput. Design: VLSI Comput. Processors*, Oct. 5-8, 1987, pp. 654-658.

Fast, Deterministic Routing, on Hypercubes, Using Small Buffers

Bradley C. Kuszmaul

***Abstract*—We propose a deterministic routing scheme for a communications network based on the k-dimensional hypercube. We present two formulations of the scheme. The first formulation delivers messages in $O(k^2)$ bit times using $O(k)$ bits of buffer space at each node in the hypercube. The second formulation assumes that there are several batches of messages to be delivered, and makes certain assumptions about the cost of sending messages along the various dimensions of the cube. In this case, the latency for delivery time is still $O(k^2)$ bit times, but the throughput is increased to one set of messages every $O(k)$ bit times. For the first formulation, we restrict ourselves to routings which are subsets of permutations (i.e., every node sends at most one message and receives at most one message). The second formulation indicates a way to perform routings which are subsets of H-permutations (i.e., every node sends at most H messages and receives at most H messages).**

***Index Terms*—Buffers, complexity theory, deterministic routing, hypercubes, interconnection networks, parallel processing, routing.**

I. Introduction

Several routing schemes based on the hypercube have been proposed [7], [5], [15], [17], [12]. We discuss hypercubes with k dimensions and $2^k = N$ vertices (which we call nodes). A nondeterministic $O(k^2)$ bit time algorithm with $O(k^2)$ bits of storage at each node is described in [17]. In this paper, we describe a deterministic $O(k^2)$ bit time algorithm with $O(k)$ bits of storage at each node. We go on to describe an alternative deterministic algorithm, based on a slightly modified network, with $O(k^2)$ bit time latency for messages traveling through the network, $O(k)$ throughput (i.e., one message every $O(k)$ bit times), and $O(k^2)$ bits of storage at each node.

When describing hypercube networks we define a *node* to be a vertex on the hypercube. When describing multiprocessor computer systems, we define a *processor* to be the hardware which sends and receives messages. In some computer systems (e.g., the connection machine [9]), the processors are associated with the nodes of the hypercube routing network.

In general, we assume that messages are at $\Theta(k)$ bits long (because, for example, it should be possible to transmit a node address in a message). This gives a lower bound for routing of $o(k)$ bit times.

Manuscript received October 6, 1987; revised January 28, 1988.

The author is with the Massachusetts Institute of Technology, Cambridge, MA 02139.

IEEE Log Number 9035138.

A *bit time* is the amount of time it takes to send one bit across one wire.

II. Special Case Routings

This section presents several algorithms and theorems that we will use in our routing algorithm. We will only consider message sets in which every node sends at most one message and receives at most one message. We can thus think of a routing as a partial function f from nodes to nodes.

We identify nodes with integers by using the standard numbering scheme for a hypercube. (I.e., node i is connected to node $i \oplus 2^j$, where $a \oplus b$ is the bitwise logical XOR of the nonnegative integers a and b represented in binary.)

A. Semi-Contractions

First we will consider a special class of routings called *semi-contractions*, and prove that we can route semi-contractions quickly.

Definition 1: An *injective routing* is a routing where no two nodes are sending a message to the same node. I.e., f is injective.

The rest of this paper will assume that a *routing* is an injective routing.

Definition 2: A *packing* is a monotonic routing onto a single interval. I.e., $\exists i, j$ such that $\forall x, y$ in the domain of f, $x < y$ iff $i \le f(x) < f(y) \le j$, and the range of f is exactly $[i, j]$.

Definition 3: A *semi-contraction* is a routing which has the property that the distance between the destinations of two packets is no greater than the distance between their initial locations. I.e., $|i - j| \ge |f(i) - f(j)|$.

Note that by $|i - j|$ we mean the difference between i and j as integers rather than, say, the Hamming distance between i and j.

Theorem 1 is due to [14] which originally proved it for packings. The proof for semi-contractions is due to [13]. A packing is a semi-contraction so we present the stronger theorem for semi-contractions even though we will only need it for packings.

Theorem 1: It is possible to route semi-contractions in bit time $O(k)$ with $O(k)$ bits of storage at each node by Algorithm 1.

Algorithm 1: (*Note that we number cube dimensions from zero to* $k - 1$):

```
for $i = 0$ to $k - 1$ do
    for each node $n$ in parallel
        If ($\exists$ message $m$ at node $n$ which wants to cross
            the $i$th dimension)
        send $m$ across the $i$th dimension.
```

Proof: The proof is based on the following lemma.

Lemma 1: At any point in time, and at every node, there is at most one message at node n which wants to cross the ith dimension.

Clearly, if Lemma 1 holds then the algorithm terminates because there are no collisions.

Proof of Lemma 1: Inductively on i (the loop variable mentioned in the algorithm for Theorem 1). Clearly, Lemma 1 is true before the algorithm begins since there is at most one message at any node. Consider just before the loop for which $i = l$, and a message which is being routed from j to $f(j)$ where f is a semi-contraction. We assume (by induction) that there have been no "collisions," which means that the relative addresses of the message's destinations are all zero in the low order $l - 1$ bits (since the messages have been able to cross every dimension that they wanted to so far). So the address field of a message looks like (writing the low order bits on the right, and denoting bits y through x of a field R where $x \ge y$ as $R[x:y]$, and bit x of R by $R[x]$)

$$\langle \text{INITIAL}[k-1:l+1] \quad \text{INITIAL}[l] \quad \text{FINAL}[l-1:0] \rangle$$

where the INITIAL bits are the bits in the original relative address (since the message has not crossed any of the high-order dimensions, the high-order bits of the relative address have not changed), and the FINAL bits are all zero (by inductive hypothesis).

Now assume that there is to be a collision on this cycle through the loop (i.e., there is a node n with two messages m_1 and m_2 both of which want to cross dimension l). We thus know that the low order $l + 1$ bits of m_1 and m_2 must be the same (i.e., the INITIAL[l] bits must be the same because both want to cross dimension l, and the low order $l - 1$ bits (the FINAL bits) are all zero). We know that since the messages are not going to the same location that the high-order bits must differ; thus, the final destinations of the two messages must differ by at least 2^{l+1}. But because f is a semi-contraction, we know that the starting locations of the two messages must differ by at least 2^{l+1}. This means that the bits INITIAL[$k - 1:l + 1$] must differ for m_1 and m_2, a contradiction to our assumption that they arrive at the same node, n.

Now we need to show that the algorithm presented in Theorem 1 can run in bit time $O(k)$ with $O(k)$ bits of storage at each node. The naive approach to the implementation of this algorithm would require $O(k^2)$ bit time: send complete messages across the ith dimension before sending them across the $i + 1$th dimension. However, we can pipeline the bits through the router to achieve $O(k)$ bit time the way the Thinking Machines connection machine does [9]: note that only one bit of the address is needed in order to make the decision about how to route the messages along any given dimension. We need to arrange that the relevant bit of the address is available at the right time. To do this we send bit number 0 of the address into the hardware logic which decides how to route dimension 0 of the hypercube. While dimension zero is being "switched," we send bit number 1 into the hardware. As soon as possible, the bit number one of the address is forwarded through the hardware on to the next level of logic. Each level of the switch can "consume" one bit of the message, and send the rest of the message on as soon as possible. The pipeline thus built is of depth k, and it will take $O(k)$ bit times to perform the switching. The rest of the message can then be sent through the switches (continuing the pipelining) in $O(k)$ additional time (since the message data size is about the same as the message address size).

B. Enumerating

The next important algorithm we need is an enumeration algorithm which gives selected processors unique numbers. We will use the enumeration to generate addresses for a packing. The algorithm for performing an enumeration in $O(k^2)$ bit time appears in [6], and was improved to $O(k)$ bit time in [3], [4], [11], and [8].

Definition 4: The *enumeration* of a set of nodes $S = \{n_1, n_2, \cdots, n_{|S|}\}$ where $n_i < n_j$ iff $i < j$ is the mapping E_S: $S \to \{1, \cdots, |S|\}$ where $E_S(n_i) = i$.

In order to use the enumeration of a set S of nodes we will need to arrange that every node $n \in S$ knows $E_S(n)$. We call this the computation of E_S. We need to be able to perform this computation quickly.

Definition 5: For i an integer in $\{0, \cdots, k - 1\}$ we define an i-subcube of the network to be a set of nodes which whose addresses in the hypercube are the same in bits 0 through $i - 1$.

In the case of $i = 0$, a 0-subcube is the set of all the nodes. Note that there are 2^i i-subcubes, each containing 2^{k-i} nodes. Thus, a 1-subcube is a set of nodes whose addresses are same in all but bit 0, and there are two 1-subcubes, while there are 2^k k-subcubes, each containing exactly one node.

Theorem 2: Given S, a set of nodes in a hypercube, we can compute E_S in time $O(k)$.

Proof: First we will sketch an algorithm for computing E_S. Then we will prove that an enumeration can be done in $O(k)$ bit times on a butterfly, and then extend the result to apply to hypercubes.

We define a complete butterfly to be a butterfly network with the ability to do certain computations on the internal vertices of the butterfly. The computations needed will be small, and the amount of hardware for each butterfly node is a constant (i.e., the amount of hardware needed for each hypercube node is linear in k). In a butterfly network of dimension k, there are $k2^k$ of these internal nodes. Using the notation of [16], where the vertices of the butterfly network are given ranks 0 through $k - 1$ and there are 2^k vertices on each rank, we define $p_{r,i}$ to be the ith node of the rth rank. We assume that at the begining of the algorithm, all the vertices of the form $p_{0,i}$ are the nodes which need to be enumerated. Let $s(i)$ be the characteristic function of S, i.e., $s(i)$ is one if $i \in S$ (i.e., $p_{0,i}$ wants to be counted in the enumeration), and zero otherwise.

We will go from rank 0 to rank $k-1$ serially, with the invariant that at the end of the computation of rank r, vertex $p_{r,i}$ "knows" two numbers:

• $t_{r,i}$ is the total number of processors which want to be counted in the $(k-r)$-subcube containing $p_{0,i}$. More precisely,

$$t_{r,i} = \sum_{j \in \text{ the } (k-r)\text{-subcube containing } i} s(j).$$

• $o_{r,i}$ is the "offset" of processor i among those which want to be counted in the $(k-r)$-subcube containing $p_{0,i}$. More precisely,

$$o_{r,i} = \sum_{j \in \text{ the } (k-r)\text{-subcube containing } i, \text{ where } j<i} s(j).$$

The following recursion relations will compute E_S:

$$\begin{aligned} t_{0,i} &= s(i) \\ o_{0,i} &= 0 \\ t_{j,i} &= t_{j-1,i} + t_{j-1,i\oplus 2^j} && \text{if } j>0 \\ o_{j,i} &= o_{j-1,i} && \text{if } j>0 \text{ and } i < i\oplus 2^j \\ o_{j,i} &= o_{j-1,i} + t_{j-1,i\oplus 2^j} && \text{if } j>0 \text{ and } i > i\oplus 2^j \end{aligned}$$

where $a \oplus b$ is the integer which in base two is the bitwise Exclusive OR of the two integers a and b.

The proof that these relations are correct is inductive on j:

• The base case is $j = 0$ which is clearly true since the $(k-0)$-subcube containing i is just i itself. The total number of processors which want to be counted in any $(k-0)$-subcube is just $s(i)$ and the offset is zero.

• The inductive case is as follows. The number of the processors which want to be counted in the $(k-j)$-subcube containing i is independent of whether $i < i \oplus 2^j$, and is just the sum of the number of processors in each of the two $k-(j-1)$-subcubes which make up the two halves of the $(k-j)$-subcube containing i, namely the $k-(j-1)$-subcube containing i and the $k-(j-1)$-subcube containing $i \oplus 2^j$. The offset of the processors in the "low" half of the $(k-j)$-subcube is not changed when the "high" half of the $(k-j)$-subcube is thrown in, which shows that $o_{j,i}$ is computed correctly for the case where $i < i \oplus 2^j$. The offset of the processors in the "high" half is incremented by the total number of processors in the "low" half.

A butterfly network can compute these relations because the recursion relation has the form that the values needed to compute t and o for some vertex x of rank j are just the values stored at the vertices of rank $j-1$ adjacent to x in the butterfly. We use pipelining to get that the enumeration can be done in $O(k)$ bit times, because at each rank j, as soon as the lth bit of o and t are received from rank $j-1$, the lth bit of o and t can be computed and sent on to rank $j+1$.

III. The Algorithm upon which All is Routed

Now we can perform our arbitrary routing using Algorithm 2, given below. First we need a bit of notation.

Definition 6: For every i in $[0 \cdots k-1]$, and for every node x define $l_{x,i}$ to be the lowest numbered node in the i-subcube containing x.

Algorithm 2: The routing algorithm.

• **for each dimension** i **in** $[0 \cdots k-1]$ **serially**

1) *All messages which want to cross dimension i do so.* Note: Now we have invariant that every message is in the same i-subcube as its destination. Also, every node has either zero, one, or two messages. Furthermore, there are at least as many nodes with zero messages as there are nodes with two messages, since all of the messages must be routed within that i-subcube.

2) **For each** *i-subcube C* **in parallel**
Enumerate all nodes in C with two messages, $E_{C,\,2\text{-}messages}$.

3) *Every node x which has two messages sends one of the two messages to node* $l_{x,i} + E_{C,\,2\text{-}messages}(x)$. Note that within each i-subcube, this is a packing and that the messages will not ever leave the i-subcube they started in, so the routing will happen in time $O(k)$.

4) **For each** *i-subcube C* **in parallel**
Enumerate all nodes in C with zero messages, $E_{C,\,0\text{-}messages}$.

5) *Every node x which has zero messages sends its own address to node*

$$l_{x,i} + E_{C,\,0\text{-}messages}(x).$$

6) *Every node which received a message in Step 3 sends the message received in Step 3 to the processor whose address was received in Step 5.* Note that every node which received a message in Step 3 above will also have received a message in Step 5 as noted in the invariant in Step 1. Now we have the invariant that every message is in the correct i-subcube and no node has more than one message.

This algorithm has the invariant that at the begining of loop number i, every message is in the correct i-subcube and there is at most one message at every node, as noted in Step 6.

The running time of Algorithm 2 is as follows: In loop number i:

• Step 1 takes $O(k)$ bit times (since the messages must be sent completely across dimension i).

• Steps 2 and 4 each take $O(i)$ bit times (since an enumerate can be done in $O(i)$ bit times on a cube of dimension i). We can round up to $O(k)$ bit times without losing anything since Step 1 has already used up $O(k)$ bit times.

• Steps 3 and 5 each take $O(k)$ bit times because they are packings.

• Step 6 takes $O(k)$ bit times because its routing is simply the routing of Step 5 run backwards in time. The fact that this routing must be run backwards from the other routings is important for hardware designers. The earliest known appearance of this "rendezvous" operation is in [6].

Thus, the running time of Algorithm 2 is $O(k)$ bit times each time around the loop, and there are k times around the loop for a total running time of $O(k^2)$.

The amount of storage required at each node is $O(k)$ bits, since the maximum number of messages which ever are at one processor in Step 1 is two. To perform an enumeration requires $O(k)$ bits of storage and k serial adders. To perform a packing we note that in the proof that semi-contractions can be routed quickly, we proved that nodes never need to store more than one message for the unpipelined case. In the pipelined version, we may be storing one bit from each of k messages at any time, and we will need routing hardware which can perform a "switching" in $O(1)$ bit time, and can push k bits out the k cube wires in $O(1)$ bit time.

IV. Higher Throughput or Lower Cost

If we make certain assumptions about the cost of sending bits across dimension i (namely that dimension 0 is more expensive than dimension k) we can achieve higher throughput by using the higher numbered dimensions more heavily. We note that Algorithm 2 only uses dimension 0 during the first iteration of the loop, and it uses dimension 1 only during the first two iterations of the loop, and in general it uses dimension i only during the first $i+1$ iterations of the loop. This means that if we have redundant wires for the higher dimensions, we can achieve higher throughput. To make this modification work out the best, we will assume that there are $i+1$ wire across dimension i. For this to be cost effective, we require that for $i > 0$ the cost of adding a wire to dimension i is a factor of

$i/(i+1)$ more expensive than the cost of adding a wire to dimension $i-1$. We could call this a *quadratic hypercube* since the number of connections at each vertex of the cube grows quadratically instead of linearly.

If we add more buffers at each node of the hypercube, we can then pipeline several batches of messages by running the successive iterations of the loop in parallel on different batches of messages, achieving a throughput of one batch of messages every $O(k)$ bit times.

V. Complete Butterflys Versus Cube Connected Cycles

Another feature of most machines currently being proposed is that since hypercubes are so expensive to wire, designers often fudge by putting several processors at each network node (e.g., in a cube connected cycles network there are k processors at each vertex). This has two effects.

- It means that we can assume $O(k)$ units of hardware at each vertex (which makes the pipelined enumeration run quickly).
- It increases the contention for network ports, and makes carefully planned algorithms such as the routing of semi-contractions lose a factor of k in time. Even the "quadratic hypercube" strategy mentioned above does not completely deal with this problem, since it is possible that routings which are injective with respect to processors are not injective with respect to the hypercube vertices.

VI. Conclusion

Thus, we have demonstrated an $O(k^2)$ bit time algorithm for routing messages on a hypercube, and proposed a modified cost function which allows us to achieve throughput of one message set every $O(k)$ bit times. The basic algorithm uses $O(k)$ bits of storage at each node of the hypercube and $O(k)$ serial adders at each node to perform the computation required to route messages. The high throughput algorithm needs $O(k^2)$ bits of memory and $O(k^2)$ serial adders at each node.

Several other $O(k^2)$ bit time algorithms which are related to this one appear in the literature, many of them presented in the guise of sorting algorithms.

If we assume that we can sort data in a certain amount of time (say $O(k^2)$ or $O(k)$ bit times), then we can perform a sort on the destination addresses, pack the resulting sorted messages into the low numbered processors, and then run a pack backwards in time to deliver the messages to their actual destinations. Thus, in only $O(k)$ additional bit times we can do routing.

Either a bitonic sort or a radix sort can be pipelined to run time $O(k^2)$ on a complete butterfly [10, pp. 232-237] and [4]. Both the bitonic sort and the radix sort have the property that they use the first dimension once, the second dimension twice, and so on, so that the high-throughput modification noted above will work. Other sorting algorithms, such as are described in [5], [2], and [15] may also be applied routing permutations, by first sorting and then packing.

There are some recent $O(k)$ bit time sorting networks (notably [1]) which might be applied to the routing problem, but those networks suffer from very large constant factors in the time and size of the network. Furthermore, those algorithms do not use a hypercube to perform the sorting, and are thus outside the scope of this paper.

It is not clear whether the algorithms described here are practical for machines currently proposed due to the constant factors involved. Furthermore, none of these strategies behave better for lightly loaded networks than they do for the worst case, while many of the currently proposed routing strategies (e.g., the router on the connection machine, and the randomization strategy given in [17]) seem to behave better for lightly loaded networks than for heavily loaded networks. If it is the case that the random message routings tend to lightly load the network, while most heavy loads on the network are regular and easy to route (e.g., for fast Fourier transform [16]), then the argument seems even stronger that the strategies given in this paper are not practical for machines currently being proposed.

As larger machines are built (e.g., with a billion processors), the asymptotic behavior of the time bounds for routing will become more important. On the other hand, it is very hard to build hypercubes of very large numbers of nodes, which indicates that we may never reach the point where the constants involved become unimportant.

Acknowledgment

Thanks to G. Blelloch for explaining to me how to do enumeration in $O(k)$ bit time, and J. Rose for many helpful comments on this paper.

I would like to thank the referees for their helpful comments.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, "An $o(n \log n)$ sorting network," in *Proc. 15th ACM Symp. Theory Comput.*, Boston, MA, Apr. 1983, pp. 1-9.
[2] K. E. Batcher, Sorting networks and their applications, in *AFIPS Proc. Spring Joint Comput. Conf.*, Atlantic City, NJ, Apr. 1968, pp. 307-314.
[3] G. E. Blelloch, "Parallel prefix vs. concurrent memory access," Tech. Rep., Thinking Machines Corp., Oct. 1986.
[4] ——, "Scans as primitive parallel operations," in *Proc. 1987 Int. Conf. Parallel Processing*, Aug. 1987, pp. 355-362.
[5] A. Borodin and J. E. Hopcroft, "Routing, merging and sorting on parallel models of computation," in *Proc. 14th Annu. ACM Symp. Theory Comput.*, San Francisco, CA, May 1982, pp. 338-344.
[6] D. P. Christman, "Programming The Connection Machine," Master's thesis, Dep. Elec. Eng. Comput. Sci., Massachusetts Institute of Technology, Jan. 1983.
[7] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. Comput.*, vol. C-36, no. 5, pp. 547-553, May 1987.
[8] F. E. Fich, "New bounds for parallel prefix circuits," in *Proc. ACM Symp. Comput.*, Apr. 1983, pp. 100-109.
[9] W. D. Hillis, *The Connection Machine.* Cambridge, MA: MIT Press, 1985.
[10] D. E. Knuth, *Sorting and Searching. Vol. 3 The Art of Computer Programming.* Reading, MA: Addison-Wesley, 1973.
[11] C. E. Leiserson, "Area efficient layouts (for VLSI)," in *Proc. Symp. Foundations Comput. Sci.*, 1980.
[12] G. Lev, N. Pippenger, and L. G. Valiant, "A fast parallel algorithm for routing in permutation networks," *IEEE Trans. Comput.*, 1981.
[13] J. Rose, Personal communication, 1985. J. Rose is currently employed at Thinking Machines Corporation, 245 First Street, Cambridge, MA.
[14] J. T. Schwartz, "Ultracomputers," *ACM Trans. Programming Languages Syst.*, vol. 2, no. 4, pp. 484-521, Oct. 1980.
[15] C. D. Thompson, "The VLSI complexity of sorting," *IEEE Trans. Comput.*, vol. C-32, pp. 1171-1184, Dec. 1983.
[16] J. D. Ullman, *Computational Aspects of VLSI.* Rockville, MD: Computer Science Press, 1984.
[17] L. G. Valiant and G. J. Brebner, "Universal schemes for parallel communication," in *Proc. 13th Annu. Symp. Theory Comput.*, 1981, pp. 263-277.

An Interconnection Network for Distributed Recursive Computations

Alain J. Martin and
Jan L. A. van de Snepscheut

Abstract—**Distributed computations may be viewed as a set of communicating processes. If such a computation is to be executed by a multiprocessor system, the processes have to be distributed over the**

Manuscript received July 23, 1984; revised July 20, 1990.
The authors are with the Department of Computer Science, California Institute of Technology, Pasadena, CA 91125.
IEEE Log Number 9038765.