[6] H. Meijer and S. G. Akl. Optimal computation of prefix sums on a binary tree of processors. *International Journal of Parallel Programming*, 16:127–136, 1987.

[7] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *JACM*, 29:642–667, 1982.

[8] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2:484–521, 1980.

[9] J. D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, Rockville, MD, 1984.

[10] P. Varman and K. Doshi. Sorting with linear speedup on a pipelined hypercube. Technical Report TR–8802, Rice University, Department of Electrical and Computer Engineering, February 1988.

Ernst W. Mayr received a diploma in Mathematics from the Technical University in Munich (Germany) in 1975, a master's degree in computer science and electrical engineering from MIT in 1977, and his Ph.D. from the Technical University in Munich in 1980. He was then a visiting scientist at MIT and Stanford, and from 1982 through 1988 on the faculty of the Stanford Computer Science Department. Since 1988, he is Professor for Theoretical Computer Science at the Johann Wolfgang Goethe-University in Frankfurt, Germany. His interests are combinatorial algorithms, parallel algorithms, and complexity theory.

C. Greg Plaxton graduated with a B.A.Sc. in Engineering Science (Computer Science option) from the University of Toronto in 1985, and received his Ph.D. in Computer Science from Stanford University in 1989. He was a Postdoctoral Fellow at the MIT Laboratory for Computer Science for one year before joining the faculty of the University of Texas at Austin as an Assistant Professor in the Department of Computer Science. His research interests include parallel computation, the analysis of algorithms, and lower bounds.

Finally, we note that a substantial theoretical advance has been made on the problem of hypercube sorting subsequent to the work described in this paper. Namely, Cypher and Plaxton [4] have developed a hypercube sorting algorithm with running time $O(\lg n(\lg\lg n)^2)$ (assuming one item per processor). On the other hand, the pipelined merging approach described in this paper continues to provide strictly the fastest known algorithms for merging on the pipelined hypercube, and for sorting on the pipelined hypercube when the load per processor (i.e., n/p) exceeds a modest threshold. Furthermore, the multiplicative constant of the $O(\lg n(\lg\lg n)^2)$ algorithm of [4] is much higher than any of the constants appearing in this paper.

References

[1] R. J. Anderson, E. W. Mayr, and M. K. Warmuth. Parallel approximation algorithms for bin packing. Technical Report STAN–CS–88–1200, Stanford University, Department of Computer Science, March 1988.

[2] S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg. Optimal simulations of tree machines. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 274–282, October 1986.

[3] G. E. Blelloch. Scans as primitive parallel operations. In *Proceedings of the 1987 IEEE International Conference on Parallel Processing*, pages 355–362, 1987.

[4] R. E. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 193–203, May 1990.

[5] C.-T. Ho and S. L. Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *Proceedings of the 1986 IEEE International Conference on Parallel Processing*, pages 640–648, 1986.

Algorithm MultiWayMergeSort

1. Mark one input value at each of the processors that holds $\lceil n/p \rceil$ values. Concentrate the resulting set of marked values to the lowest-numbered processors. This can be done in $2l + 2m$ time steps with a prefix sum followed by a concentration route.

2. Each column of the array now contains k values. Route these values to the top of the column (i.e., to row 0). As in Step 7 of MultiWayMerge, this takes $k + l$ time steps.

3. At every processor in row 0, sort the set of k values using an efficient sequential sorting routine. This takes $O(k \lg k)$ time steps.

4. Repeatedly call MultiWayMerge. The length of the sorted lists increases by a factor of 2^l after each call. Thus, after $\lceil m/l \rceil$ iterations all of the values have been sorted. The cost of the ith iteration is $14k+5l+19il$ time steps (see the analysis of MultiWayMerge), for a total cost of approximately $(14k + 5l + \frac{19}{2}m)m/l$ time steps.

5. The values have been sorted, but they are not configured appropriately (i.e., all of the values are in row 0). All of the values can be routed to the correct output locations using k pipelined inverse concentration routes, which takes $k + \lg p$ time steps.

The total running time of MultiWayMergeSort is minimized (to within a constant factor) by setting $k = \lg p$, and for this choice of k the running time is dominated by the cost of Step 4. Observing that $l = \lg(pk/n)$ and $m = \lg p - l \leq \lg p$, we find that for $k = \lg p$ the algorithm runs in $\frac{47}{2} \lg^2 p / \lg((p \lg p)/n) + O(\lg p \lg \lg p)$ time steps. For the case $n = p$, we can set $k = \lg p / \lg \lg p$ and reduce the running time to $\frac{19}{2} \lg^2 p / \lg \lg p + O((\lg p / \lg \lg p)^2)$.

6 Concluding Remarks

In order to better assess the practical speed of the various algorithms presented in this paper, we have computed the coefficient of the leading term of the running time in each case. It is quite possible that one or more of the moderately large coefficients in Section 5 could be improved with only minor modifications to the algorithm.

In other words, route the rank of each value back to the processor that contained the value before Step 3. This is a pipelined inverse concentration, and can be performed in $k+m$ time steps. Where $i=j$, simply label each value with its rank in X^i.

5. Compute the rank of every value in X. The processors of row i are used to perform this computation for the elements of the set X^i, $0 \leq i < 2^l$. For each set X^i_j, we perform a pipelined sum over a subcube of dimension l, adding the ranks computed in Step 4 and routing the results to the first block of 2^m processors in each row. This takes $k+l$ time steps using the MSBT embedding.

6. In row i, route the elements of X^i to the correct output column (given by the floor of the rank computed in Step 5 divided by k), $0 \leq i < 2^l$. This is a pipelined inverse concentration in a subcube of dimension $l+m$, and takes $k+l+m$ time steps.

7. Each column of the array now contains k values. Route these values to the top of the column (row 0). This can be performed in $k+l$ time steps using the MSBT embedding over subcubes of dimension l.

Summing all of the costs stated above, the total running time of MultiWayMerge is readily seen to be $14k+5l+19m$ time steps.

By repeatedly applying MultiWayMerge over successively larger subcubes, we can obtain a fast sorting algorithm for the case $n < p \lg p$. The running time of this algorithm, which we refer to as MultiWayMergeSort, will be shown to be $O(\lg^2 p / \lg((p \lg p)/n)))$, as opposed to $O(\lg^2 p / \lg(p/n))$ for the sorting algorithm of Nassimi and Sahni. For the interesting case $n = p$, the running time of MultiWayMergeSort is $O(\lg^2 p / \lg \lg p)$, a slight asymptotic improvement over that of Batcher's bitonic sort.

We now give a more formal description of the MultiWayMergeSort algorithm, and analyze its time complexity. The algorithm is designed to sort $n = k2^m$ values on a hypercube with $p = 2^{l+m}$ processors. It is useful to view the processors as being arranged in a 2^l by 2^m array, where the processor in row i and column j has ID $i2^m + j$ (row-major order).

be denoted X^i, $0 \leq i < 2^l$, and let the set of k elements of X^i with ranks between jk and $(j+1)k-1$ (inclusive) be denoted X^i_j, $0 \leq j < 2^m$. The set X^i_j is initially stored at processor $i2^m + j$. Let the output list be denoted X. At the end of the merging process, the elements of X with ranks between jk and $(j+1)k-1$ (inclusive) should be stored at processor j, $0 \leq j < 2^{l+m}$. It is useful to view the processors of the given hypercube as forming a 2^l by 2^{l+m} array, where the processor in row i and column j has ID $i2^{l+m} + j$ (row-major order).

Our algorithm makes use of pipelined broadcast and sum operations over entire subcubes. Formally, a pipelined broadcast operation takes k values stored at a single processor and broadcasts them over the entire subcube. For a pipelined sum operation, processor i initially holds k values a_{ij}, $0 \leq i < p$, $0 \leq j < k$. The output is the k sums $\sum_{0 \leq i < p} a_{ij}$, $0 \leq j < k$, all of which are output at a single designated processor. Although such operations can be performed using Prefix, other implementations exist that are more efficient by a constant factor. For example, using the multiple spanning binomial tree (MSBT) embedding of Ho and Johnsson [5] it is possible to perform k broadcasts (or sums) in $k + \lg p$ time steps.

Algorithm MultiWayMerge

1. Broadcast X^i_j to all of the processors in column $i2^m + j$, $0 \leq i < 2^l$, $0 \leq j < 2^m$. Each of the columns is an independent subcube of dimension l. Thus, the broadcasts can be performed in $k + l$ time steps using an MSBT embedding within each column.

2. Replicate list X^i across the ith row, $0 \leq i < 2^l$. In other words, route a copy of X^i_j to each column of the ith row that is congruent to j mod 2^m. This can be done in $k + l$ time steps using the MSBT embedding over subcubes of dimension l.

3. Merge the lists X^i and X^j using the jth block of 2^m processors of row i (i.e., columns $j2^m$ to $(j+1)2^m - 1$), $0 \leq i, j < 2^l$, $i \neq j$. This takes $8k + 17m$ time steps (see the analysis of Merge).

4. In the jth block of 2^m processors of row i, "unmerge" the rank of each element of X^i in X^j (this is the rank of that value in $X^i \cup X^j$ minus its rank in X_i), $0 \leq i, j < 2^l$, $i \neq j$.

5. Assuming the set X_i was routed to processor j_i in the previous step, broadcast X_i to all processors with IDs in the range $j_i + 1$ to j_{i+1}, $0 \leq i < p$. This can be done in $2k + 4 \lg p$ time steps with a single application of Prefix$'$.

6. Assuming the set Y_i was routed to processor j_i in the previous step, broadcast Y_i to all processors with IDs in the range j_{i-1} to $j_i - 1$, $0 \leq i < p$. This can be done with a single application of a "backwards" version of Prefix$'$, and takes $2k + 4 \lg p$ time steps.

7. At this point, processor j contains a copy of Z'_{2j}, Z'_{2j+1}, the largest X_i with $x_i < z_{2j}$ and the smallest Y_i with $y_i > z_{2j+1}$, $0 \leq j < p$. As observed above, the union of these sets contains the desired set Z_j, and the values to be discarded (i.e., those not belonging to Z_j) can be determined by computing the exact rank of either z_{2j} or z_{2j+1}. These sets can be merged, and the rank computation performed, with $O(k)$ local operations. Our definition of a time step allows these local operations to be interleaved with the computations of Steps 5 and 6 at no extra cost.

The total running time of Merge is $8k + 17 \lg p$ time steps. For $k = \Omega(\lg p)$, this running time is within a constant factor of optimal. Furthermore, as observed by Varman and Doshi, this optimal merging routine immediately implies an optimal algorithm for sorting when the number of values to be sorted, n, exceeds the number of processors, p, by a factor k that is $\Omega(\lg p)$. The idea is to sort the set of k values at each processor locally, and then to merge sorted subcubes repeatedly until the entire hypercube has been sorted. At each level, even subcubes are sorted in ascending order and odd subcubes are sorted in descending order. The running time of this algorithm, which we refer to as MergeSort, is

$$\sum_{0 \leq i < \lg p} (8k + 17i) = 8k \lg p + O(\lg^2 p).$$

We now describe a pipelined version of the multi-way merging procedure of Nassimi and Sahni [7] that runs on the pipelined hypercube. The input consists of 2^l sorted lists of length $k2^m$, and the output is a single sorted list of length $k2^{l+m}$. The merging is performed in $O(k + \lg p)$ time steps on a hypercube with $p = 2^{2l+m}$ processors. Let the ith input list

sorted locally.

Our approach is to first merge X' and Y', and then to use the resulting list to guide the merging of X and Y. Let Z' denote the sorted list of length $2p$ that results from merging X' and Y'. Let z_j denote the value with rank j in Z', $0 \leq j < 2p$. Let Z'_j denote the set of k values associated with z_j, that is, either $z_j = x_i$ for some $x_i \in X'$ and $Z'_j = X_i$, or $z_j = y_i$ for some $y_i \in Y'$ and $Z'_j = Y_i$. Note that if $z_j \in X'$ then the rank of z_j in Z is between jk and $(j+1)k-1$, inclusive. The exact rank of z_j in Z can be determined by computing its rank in the set Y_i, where y_i is the least element of Y' exceeding z_j. Similarly, if $z_j \in Y'$ then the rank of z_j in Z is between jk and $(j+1)k-1$, and the exact rank of z_j in Z depends upon the set X_i, where x_i is the largest element of X' that is less than z_j. Furthermore, it is easy to check that the set Z_j is contained in the union of Z'_{2j}, Z'_{2j+1}, the set X_i corresponding to the largest x_i that is less than z_{2j} (if such a set X_i exists), and the set Y_i corresponding to the smallest y_i that is greater than z_{2j+1} (if such a set Y_i exists). For example, it is clear that no element of any set Y_k such that $y_k < z_{2j}$ could belong to Z_j, since the rank of y_k (the largest element in Y_k) is at most $2jk-1$, whereas the rank of the smallest element in Z_j is $2jk$. The foregoing observations lead to the following pipelined merging algorithm.

Algorithm Merge

1. Reverse list Y', routing y_i to processor $p-i-1$, $0 \leq i < p$. This takes $\lg p$ time steps.

2. Merge X' and Y' by simulating a bitonic merge over $2p$ processors. This takes $2\lg p$ time steps.

3. Route the rank of each value in Z' back to the processor which originally held that value. This takes $2\lg p$ time steps.

4. Route each set X_i to the processor that held x_i after Step 2, $0 \leq i < p$. The ID of that processor can be computed from the rank received by processor i in Step 3. This route can be performed in $2k+2\lg p$ time steps using a pipelined inverse concentration. Route the Y_i's similarly, for a total cost of $4k+4\lg p$ time steps.

as well as the hardware implementation details.

We only need the pipelined model of the hypercube for performing pipelined inverse concentration routes. It is interesting to note that we do not require pipelined concentration routes, nor do we require the pipelined inverse concentration *with copy* operation of Varman and Doshi. Concentration and inverse concentration routes were defined by Nassimi and Sahni [7], and it is easy to show that k such operations can be performed in $k + \lg p$ time steps on the pipelined hypercube model. Furthermore, there is no hope of achieving this asymptotic time bound on the 1-port model, since there is a lower bound of $\Omega(k \lg^{1/2} p)$ time steps in this case. To prove this lower bound, consider a set of k monotone routes for which the source processors are exactly those with strictly more 0's than 1's in their IDs, and the destination processors are those with more 1's than 0's. In such a case, $\Omega(kp)$ packets must pass through the $O(p \lg^{-1/2} p)$ processors with an equal number of 0's and 1's (or one more 0 than 1, say, if $\lg p$ is odd), which implies a lower bound of $\Omega(k \lg^{1/2} p)$ time steps for performing k monotone routes. Since a monotone route is equivalent to a concentration route followed by an inverse concentration, and these operations have equal complexity, this lower bound also applies to the pipelined concentration and inverse concentration operations.

We now describe a pipelined algorithm for merging two sorted lists X and Y, each of length pk, on p processors. The algorithm is similar to that proposed by Varman and Doshi [10], but is somewhat simpler. The optimal merging algorithm of Anderson, Mayr, and Warmuth for the EREW PRAM also takes a similar approach [1]. For simplicity, it will be assumed that all of the $2pk$ input keys are distinct (this assumption is not essential). For both X and Y, the values with ranks (numbered from 0) in the range ik to $(i+1)k-1$ are initially stored at processor i, $0 \leq i < p$. The two ordered sets of k values located at processor i will be referred to as X_i and Y_i, respectively. Let x_i denote the least element of X_i, and let y_i denote the greatest element of Y_i, $0 \leq i < p$. Let X' and Y' denote the set of all x_i's and y_i's, respectively. Let Z denote the sorted list of length $2pk$ that results from merging X and Y. Those elements of Z with ranks in the range $2ik$ to $2(i+1)k-1$, denoted Z_i, must be routed to processor i by the end of the computation, $0 \leq i < p$, and must be

Dropping the inner parentheses and simplifying, this amounts to

$$(a_0, b_0, x_0) \oplus' (a_1, b_1, x_1) = (a_0 \textbf{ or } a_1, \textbf{ if } a_1 \textbf{ then } b_1 \textbf{ else } b_0 \textbf{ or } b_1, \textbf{ if } (a_1 \textbf{ or not } b_0) \textbf{ then } x_1 \textbf{ else } x_0).$$

Note that the above formulation allows bit pipelining in the sense described by Blelloch [3]. Finally, we observe that the data distribution operation defined by Ullman [9] is equivalent to a segmented Prefix operation with the Copy operator. Thus, the techniques outlined in this paper immediately lead to efficient pipelined implementations of this primitive for the complete inorder binary tree, hypercube, and shuffle-exchange.

5 Sorting on a Pipelined Hypercube

In this section, we describe a simplified implementation of the optimal merging algorithm of Varman and Doshi [10], and show how this can be used to develop a pipelined version of the sorting algorithm of Nassimi and Sahni [7] for a *pipelined model* of the hypercube.

The Sort operation is defined as follows. Given n $O(\lg p)$-bit values, with $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ located at each processor, rearrange the n values so that every value in processor i is less than or equal to every value in processor j whenever $0 \leq i < j < p$. In addition, we require that there be $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ values at any processor, and that the set of values within any particular processor be sorted. There has been a great deal of previous research related to the problem of sorting on the hypercube and related networks, under a variety of different models of computation. For an overview of the hypercube sorting literature, the reader is referred to [4].

The time bounds for the merging and sorting algorithms described in this section do not apply to the 1-port model of computation that we have been considering up to this point. Instead, we will make use of a restricted form of the less realistic $(\lg p)$-port model, in which a processor can send and/or receive a packet from each of its $\lg p$ neighbors in a single time step. This model, which we refer to as the *pipelined hypercube model*, was originally defined by Varman and Doshi [10], and we refer the reader to their paper for both the strict definition

In Section 5, we will make use of a variant of the Prefix operation, Prefix$'$, defined as follows. Rather than computing $x_0 \oplus \cdots \oplus x_i$ at processor i, $0 \leq i < p$, Prefix$'$ outputs $\mathbf{0}_\oplus$ at processor 0 and $x_0 \oplus \cdots \oplus x_{i-1}$ at processor i, $1 \leq i < p$. This is sometimes more convenient, particularly when the operator $\oplus$ is not invertible. Our Prefix algorithms may be trivially modified to implement Prefix$'$ with the same time bounds.

4 Data Distribution

Consider the binary associative operator $\oplus : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ such that $x \oplus y = x$ for all $x, y \in \mathcal{D}$. This is sometimes referred to as the Copy operator. Observe that the effect of applying Prefix with the Copy operator is to perform a broadcast of a single value from processor 0 to all other processors. Of course, there are simpler techniques for broadcasting a single value over the processors of any of the networks we have considered. However, combining this observation with the results of the previous section immediately implies that k *segmented* broadcasts can be executed in $2k + 4 \lg p$ time steps on the tree or hypercube, and in $6k + 10 \lg p$ time steps on the shuffle-exchange.

In order to fully illustrate the techniques discussed in Section 2, we now study the implementation of segmented Prefix with the Copy operation in greater detail. Assume that processor i initially holds the Boolean value a_i, and $x_i \in \mathcal{D}$, $0 \leq i < p$. Note that there is no identity element for the Copy operation in $\mathcal{D}$. In order to remedy this situation, we can extend the domain of Copy from $\mathcal{D}$ to $\mathcal{B} \times \mathcal{D}$ and define every pair with first component **false**, say, to be an identity element. Formally, we have

$$(b_0, x_0) \oplus (b_1, x_1) = (b_0 \textbf{ or } b_1, \textbf{ if } b_0 \textbf{ then } x_0 \textbf{ else } x_1),$$

for all $b_0, b_1 \in \mathcal{B}$ and $x_0, x_1 \in \mathcal{D}$.

In order to reduce segmented Prefix with operator $\oplus$ = Copy to ordinary Prefix with operator $\oplus'$ = Copy$'$, we define $\oplus'$ as follows:

$$(a_0, (b_0, x_0)) \oplus' (a_1, (b_1, x_1)) = (a_0 \textbf{ or } a_1, \textbf{ if } a_1 \textbf{ then } (b_1, x_1) \textbf{ else } (b_0, x_0) \oplus (b_1, x_1)).$$

Figure 1: A shuffle-exchange embedding for the high-numbered processors.

Figure 2: A shuffle-exchange embedding for the low-numbered processors.

(0 to $p/2 - 1$) at its leaves. We can make use of these embeddings to obtain a pipelined implementation of k Prefix operations as follows. First, we use the embedding of Figure 1 to compute the k sets of partial sums over the high-numbered processors. This takes $2k + 4 \lg p$ time steps. Similarly, the embedding of Figure 2 can be used to perform k prefix sums over the low-numbered processors in $2k + 4 \lg p$ time steps. It is now sufficient to broadcast, in a pipelined fashion, the k total sums over the low-numbered processors to the $p/2$ high-numbered processors, and to add these values to the partial sums computed earlier. This last phase can be performed in $2k + 2 \lg p$ time steps using the embedding of Figure 2 (note that the desired sums are already available at the root), so k Prefix operations can be executed in $6k + 10 \lg p$ time steps on the shuffle-exchange.

its parent. During the downward pass, each processor receives from its parent the sum over all processors with IDs less than those in $T(p)$, computes the sum over all processors with IDs less than those in its right subtree, and sends the appropriate values to its left and right children. The entire algorithm runs in $4 \lg p$ time steps.

Note that in any given time step, only two of the levels of the tree are active, implying that the algorithm can be pipelined level by level. By initiating a new prefix computation every second time step, it is possible to perform k Prefix operations on the inorder complete binary tree in $2k + 4 \lg p$ time steps.

Hypercube. It is straightforward to implement Prefix to run in exactly $\lg p$ time steps on a p-processor hypercube. Unfortunately, this single-pass algorithm cannot be pipelined because it uses all of the processors at every time step. To achieve pipelined speedup we can make use of the dilation 2 inorder complete binary tree embedding [2]. In this embedding, the left child of a non-leaf processor is connected directly to its parent, while the right child is connected to its parent via the left child. It is easy to verify that the pipelined algorithm stated earlier for the inorder complete binary tree can be modified to run in the same time bound on the dilation 2 inorder complete binary tree embedding. Hence, k Prefix operations can be performed in $2k + 4 \lg p$ time steps on the hypercube.

Shuffle-exchange. As in the case of the hypercube, it is straightforward to implement Prefix to run in $\lg p$ steps on a p-processor shuffle-exchange. Once again, however, the basic approach does not lead to an efficient pipelined implementation. Furthermore, certain edges of the inorder complete binary tree correspond to $\Omega(\lg n)$-length paths in the shuffle-exchange. For this reason, a straightforward emulation of the inorder complete binary tree algorithm would also be very inefficient. Instead, we make use of the dilation 2 complete binary tree embeddings depicted, for the case $p = 16$, in Figures 1 and 2. The leaves of the tree in Figure 1 are the high-numbered processors (those with IDs in the range $p/2$ to $p-1$), numbered inorder. In this embedding, the ID of the left child of an internal processor is the shuffle of the ID of its parent, and siblings communicate via the exchange connection. The embedding of Figure 2 is defined in a similar fashion, and has the low-numbered processors

belongs to an "undefined" interval; it is **false** for output index i if and only if $a_0, \ldots, a_i$ are all **false**. As observed by Schwartz [8], this reduces coding segmented prefix to coding ordinary prefix.

3 Network Implementations

In this section, we describe efficient implementations of the Prefix operation for the complete binary tree, hypercube, and shuffle-exchange families of networks. We will be concerned with p-processor network implementations of the Prefix operation where processor i initially contains the value x_i, $0 \leq i < p$, and $n = p$. The computation is considered to be complete when the partial sum $y_i = x_0 \oplus \cdots \oplus x_i$ has been computed at processor i, $0 \leq i < p$.

The model of computation that we adopt for our networks may be defined as follows. Each processor has an infinite local memory configured in $O(\lg p)$-bit words and can perform the usual set of CPU operations in constant time on word-sized operands. Processors communicate with one another by sending *packets* over the links provided by the network. A packet consists of a single word of data. The complexity of our algorithms will be stated in terms of *time steps*. Unless otherwise stated, running times should be assumed to be accurate to within an additive constant. In a single time step, each processor can send and/or receive at most one packet (1-port communication), and can also execute a constant number of CPU operations on local data. All of our algorithms are SIMD. We will assume that the x_i's, as well as all partial sums of the x_i's, are word-sized quantities.

Binary tree. The first implementation of Prefix that we consider is the standard two-pass algorithm for the inorder complete binary tree (see, for example, [3, 6]). Assume that we are given a tree of size $p = 2^d - 1$, with processors numbered "inorder" (i.e., numbered according to an inorder traversal of the tree) from 0 to $2^d - 2$. The first pass of the algorithm is upward, from the leaves to the root, and the second pass is downward. For every processor p, let $T(p)$ denote the subtree rooted at processor p. Note that the IDs of the processors in $T(p)$ form a contiguous block of integers. During the upward pass, each processor receives the sum of its left and right subtrees, computes the sum of $T(p)$, and passes the result to

true and extends up to, but not including, the next highest integer j such that $a_j = \mathbf{true}$. The first interval begins at index 0 regardless of the value of a_0, and the last interval ends at index $n-1$. The segmented Prefix operation executes a prefix operation over each interval. Extending the example of the preceding paragraph, assume that a_2 and a_4 are **true** while a_0, a_1, and a_3 are **false**. Then the x_i values are partitioned into the intervals $\{x_0, x_1\}$, $\{x_2, x_3\}$, and $\{x_4\}$ and the output of the segmented Prefix operation is $y_0 = 5$, $y_1 = 7$, $y_2 = 6$, $y_3 = 10$, and $y_4 = 9$.

We will assume the existence of an identity element for $\oplus$ in $\mathcal{D}$, which we denote $\mathbf{0}_\oplus$. This assumption can be made without loss of generality because if no such element exists, we can simply augment the set $\mathcal{D}$ with an identity element $\mathbf{0}_\oplus$ such that $\mathbf{0}_\oplus \oplus x = x$ and $x \oplus \mathbf{0}_\oplus = x$ for all $x \in \mathcal{D}$.

Definition 2.1 For all pairs of Boolean values a_0, a_1 and all $x_0, x_1 \in \mathcal{D}$, let $\oplus'$ denote the binary operation

$$(a_0, x_0) \oplus' (a_1, x_1) = (a_0 \vee a_1,\ \mathbf{if}\ a_1\ \mathbf{then}\ x_1\ \mathbf{else}\ x_0 \oplus x_1).$$

The operation $\oplus'$ will be referred to as the *segmented* $\oplus$ *operation.*

The following four facts are straightforward to prove: (i) the operation $\oplus'$ has identity $\mathbf{0}_{\oplus'} = (\mathbf{false}, \mathbf{0}_\oplus)$; (ii) the $\oplus'$ operation is not commutative, assuming $|\mathcal{D}| > 1$; (iii) the $\oplus'$ operation is associative; (iv) for $k \geq 0$,

$$(a_0, x_0) \oplus' \cdots \oplus' (a_k, x_k) = (a_0 \vee \cdots \vee a_k,\ x_j \oplus \cdots \oplus x_k),$$

where j is the highest index less than or equal to k such that $a_j = \mathbf{true}$, or 0 if there is no such index. Facts (iii) and (iv) demonstrate that any segmented Prefix computation with operator $\oplus$ mapping $\mathcal{D} \times \mathcal{D}$ to $\mathcal{D}$ is equivalent to an ordinary Prefix computation with operator $\oplus'$ mapping $(\mathcal{B} \times \mathcal{D}) \times (\mathcal{B} \times \mathcal{D})$ to $\mathcal{B} \times \mathcal{D}$, where $\mathcal{B}$ denotes the set of Boolean values $\{\mathbf{true}, \mathbf{false}\}$. The second component of each output pair is the result of the desired segmented Prefix computation, and the first component indicates whether or not that index

1 Introduction

In this research note, we review previously known parallel algorithms for the prefix operation, extend these algorithms to run in a "pipelined" mode, and then apply the pipelined parallel prefix primitive to obtain efficient algorithms for merging and sorting on the pipelined hypercube model.

The paper is organized as follows. Section 2 defines the prefix operation, and reviews a number of simple results related to that operation. Section 3 describes pipelined parallel implementations of the prefix operation for several commonly-studied fixed interconnection networks. Section 4 applies these results to obtain efficient pipelined implementations of the "data distribution" primitive of Ullman [9]. In Section 5, we make use of the pipelined data distribution primitive to obtain a simplified implementation of the optimal merging algorithm of Varman and Doshi [10], which runs on a pipelined model of the hypercube. This merging routine leads to an optimal sorting algorithm for the pipelined hypercube when the number of values to be sorted, n, exceeds the number of processors, p, by an $\Omega(\lg p)$ factor. Section 5 also describes a pipelined version of the multi-way merge sort algorithm of Nassimi and Sahni [7] for the pipelined hypercube.

2 The Prefix Operation

We will now review the basic definitions necessary to understand the prefix and segmented prefix operations. These operations were first introduced and implemented by Schwartz [8], where they are referred to as "summing" and "summing by groups", respectively.

Let $\oplus$ denote a binary associative operator over some domain $\mathcal{D}$. Given $\{x_0, \ldots, x_{n-1}\}$ belonging to $\mathcal{D}$, the Prefix operation computes each of the partial sums $y_i = x_0 \oplus \cdots \oplus x_i$, $0 \leq i < n$. For example, assuming that $\oplus$ denotes ordinary addition, $n = 5$, $x_0 = 5$, $x_1 = 2$, $x_2 = 6$, $x_3 = 4$, and $x_4 = 9$, then the output of Prefix is $y_0 = 5$, $y_1 = 7$, $y_2 = 13$, $y_3 = 17$, and $y_4 = 26$.

Given an additional n Boolean values $a_0, \ldots, a_{n-1}$, we can partition the n given x_i values into contiguous intervals in the following manner: an interval begins at each i such that $a_i =$

Pipelined Parallel Prefix Computations, and Sorting on a Pipelined Hypercube*

Ernst W. Mayr† C. Greg Plaxton‡

Abstract

This paper brings together a number of previously known techniques in order to obtain practical and efficient "pipelined" implementations of the prefix operation for the complete binary tree, hypercube, and shuffle-exchange families of networks. In each case, we provide a scheme for performing k prefix operations in $O(k + \lg p)$ time on p processors. The same bounds are shown to apply to the "data distribution" operation of Ullman [9]. The data distribution primitive leads to a simplified implementation of the optimal merging algorithm of Varman and Doshi [10], which runs on a pipelined model of the hypercube. Finally, a pipelined version of the multi-way merge sort of Nassimi and Sahni [7], running on the pipelined hypercube model, is described. Given p processors and $n < p \lg p$ values to be sorted, the running time of the pipelined algorithm is $O(\lg^2 p / \lg((p \lg p)/n))$.

Proposed running head: Pipelined Parallel Prefix.

*This work was supported in part by a grant from the AT&T Foundation, NSF grant DCR-8351757 and ONR grant N00014-88-K-0166.

†Fachbereich Informatik, J. W. Goethe University, Frankfurt, Germany.

‡Primarily supported by a 1967 Science and Engineering Scholarship from the Natural Sciences and Engineering Research Council of Canada. Department of Computer Science, Stanford University, Stanford CA 94305.