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 implementations of the prefix operation for the complete binary tree, hypercube and shuffle exchange families of networks. For each of these networks, we also provide a "pipelined" scheme for performing k prefix operations in $O(k + \log p)$ time on p processors. This implies a similar pipelining result for the "data distribution" operation of Ullman [16]. The data distribution primitive leads to a simplified implementation of the optimal merging algorithm of Varman and Doshi, which runs on a pipelined model of the hypercube [17]. Finally, a pipelined version of the multi-way merge sort of Nassimi and Sahni [10], running on the pipelined hypercube model, is described. Given p processors and $n < p \log p$ values to be sorted, the running time of the pipelined algorithm is $O(\log^2 p / \log((p \log p)/n))$. Note that for the interesting case $n = p$ this yields a running time of $O(\frac{\log^2 p}{\log \log p})$, which is asymptotically faster than Batcher's bitonic sort[3].

*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, West 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.

1 The Prefix Operation

We begin by reviewing the basic definitions necessary to understand the prefix and segmented prefix operations. These operations were first introduced by Schwartz, where they are referred to as "summing" and "summing by groups" [14].

Let $\oplus$ denote a binary associative operation mapping $\mathcal{X} \times \mathcal{X}$ to $\mathcal{X}$, for some domain $\mathcal{X}$. Given n values $x_0, \ldots, x_{n-1}$ belonging to $\mathcal{X}$, the Prefix operation computes each of the partial sums $y_i = x_0 \oplus \cdots \oplus x_i$, $0 \le i < n$. For example, assume that $\oplus$ is 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 $a_i = \textbf{true}$ and extends up to, but not including, the next highest integer j such that $a_j = \textbf{true}$. The first interval begins at processor 0 regardless of the value of a_0, and the last interval ends at processor $n-1$. 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 $y_4 = 9$.

When we give implementations of the Prefix operation in Section 2, it will be convenient to assume that there is an identity element for $\oplus$ in $\mathcal{X}$, 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 $\mathcal{X}$ with an identity element $\mathbf{0}_\oplus$ by defining $\mathbf{0}_\oplus \oplus x = x$ and $x \oplus \mathbf{0}_\oplus = x$ for all $x \in \mathcal{X}$.

Definition 1.1 *For all pairs of boolean values a_0, a_1 and all $x_0, x_1 \in \mathcal{X}$, let $\oplus'$ denote the binary operation*

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

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

Remark 1 *The operation $\oplus'$ has identity $\mathbf{0}_{\oplus'} = (\textbf{false}, \mathbf{0}_\oplus)$.*

Remark 2 *The $\oplus'$ operation is not commutative, assuming $|\mathcal{X}| > 1$.*

Remark 3 *The $\oplus'$ operation is associative.*

Remark 4 *For $k \ge 0$,*

$$(a_0, x_0) \oplus' \cdots \oplus' (a_k, x_k) = (a_0 \textbf{ or } \cdots \textbf{ or } a_k,\ x_j \oplus \cdots \oplus x_k),$$

where j is the highest index less than or equal to k such that $a_j = \textbf{true}$, or 0 if there is no such index.

Remark 1 is an immediate consequence of Definition 1.1. For Remark 2, let x_0, x_1 be distinct elements of $\mathcal{X}$ and note that $(\mathbf{true}, x_0) \oplus' (\mathbf{true}, x_1) = x_1$ while $(\mathbf{true}, x_1) \oplus' (\mathbf{true}, x_0) = x_0$. Remark 3 follows from the observation that for all boolean values a_0, a_1, a_2 and $x_0, x_1, x_2 \in \mathcal{X}$ we have

$$\begin{aligned}
&((a_0, x_0) \oplus' (a_1, x_1)) \oplus' (a_2, x_2) \\
&\quad = (a_0 \textbf{ or } a_1,\ \textbf{if } a_1 \textbf{ then } x_1 \textbf{ else } x_0 \oplus x_1) \oplus' (a_2, x_2) \\
&\quad = (a_0 \textbf{ or } a_1 \textbf{ or } a_2,\ \textbf{if } a_2 \textbf{ then } x_2 \textbf{ else if } a_1 \textbf{ then } x_1 \oplus x_2 \textbf{ else } x_0 \oplus x_1 \oplus x_2) \\
&\quad = (a_0 \textbf{ or } (a_1 \textbf{ or } a_2),\ \textbf{if } (a_1 \textbf{ or } a_2) \textbf{ then } X \textbf{ else } x_0 \oplus X) \\
&\quad = (a_0, x_0) \oplus' (a_1 \textbf{ or } a_2,\ X) \\
&\quad = (a_0, x_0) \oplus' ((a_1, x_1) \oplus' (a_2, x_2)),
\end{aligned}$$

where X denotes the conditional expression: **if** a_2 **then** x_2 **else** $x_1 \oplus x_2$. Finally, Remark 4 may be easily established by induction on k.

Remarks 3 and 4 demonstrate that any segmented Prefix computation with operator $\oplus$ mapping $\mathcal{X} \times \mathcal{X}$ to $\mathcal{X}$ is equivalent to an ordinary Prefix computation with operator $\oplus'$ mapping $(\mathcal{B} \times \mathcal{X}) \times (\mathcal{B} \times \mathcal{X})$ to $\mathcal{B} \times \mathcal{X}$, 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 processor belongs to an "undefined" interval; it is **false** at processor i if and only if $a_0, \ldots, a_i$ are all **false**. By making use of the fact that segmented Prefix is equivalent to ordinary Prefix, we can avoid coding up a potentially messy direct implementation of segmented Prefix.

2 Network Implementations

In this section, we develop 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, $0 \le i < p$, 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, $0 \le i < p$.

The model of computation that we adopt for our networks may be defined as follows. Each processor has an infinite local memory configured in $O(\log p)$-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 is allowed to send and/or receive at most one packet (1-port communication), and execute a constant number of CPU operations on local data. We will assume that the x_i's, as well as all partial sums of the x_i's, are word-sized quantities.

All interprocessor communication in our programs is specified using the pair of routines Send and Receive. Send takes two arguments: the first specifies the word of data to be transmitted,

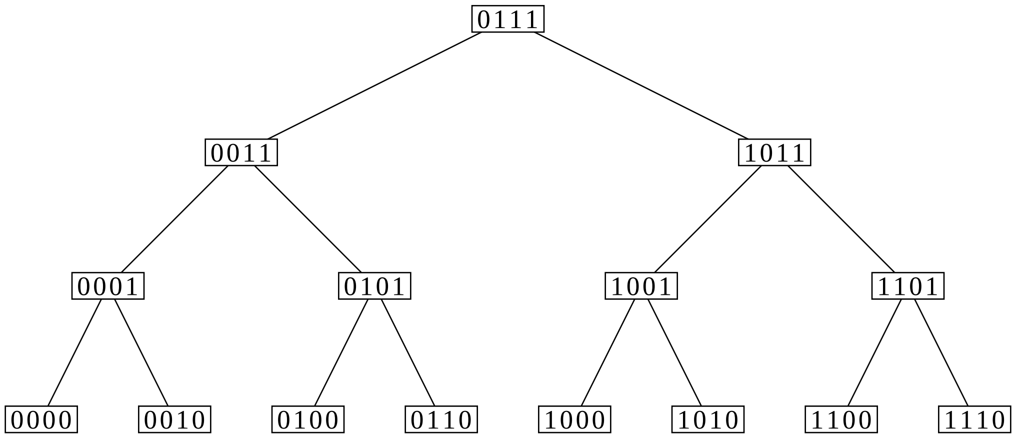

Figure 1: An inorder complete binary tree.

and the second specifies the id of the destination processor. Receive is a function with one argument, which specifies the id of the source processor. Once a packet arrives from the source, the word of data contained in that packet is returned as the value of the function. In order to comprise a valid source/destination pair, two processors must be adjacent in the network.

2.1 Binary Tree

The first implementation of Prefix that we consider is the standard two-pass algorithm for the inorder complete binary tree. Assume that we are given a tree of size $p = 2^d - 1$, with processors numbered inorder from 0 to $2^d - 2$. An example of such a network is given in Figure 1, where the processor ids have been written in binary, and $d = 4$. Our code for this algorithm assumes that each processor has initialized the variables *Root*, *Leaf*, *LeftChild*, *RightChild* and *Parent* in the following manner. The boolean variable *Root* (*Leaf*) is **true** if and only if the processor represents the root (a leaf) of the tree. The integer variables *LeftChild*, *RightChild* and *Parent* hold the ids of the neighboring processors, and are undefined whenever such a neighbor does not exist.

begin Prefix($\oplus$, x)
  $x_{\mathrm{L}} \longleftarrow$ **if** *Leaf* **then** $\mathbf{0}_\oplus$ **else** Receive(*LeftChild*);
  $x_{\mathrm{R}} \longleftarrow$ **if** *Leaf* **then** $\mathbf{0}_\oplus$ **else** Receive(*RightChild*);
  if not *Root* **then** Send($x_{\mathrm{L}} \oplus x \oplus x_{\mathrm{R}}$, *Parent*);
  $y_{\mathrm{L}} \longleftarrow$ **if** *Root* **then** $\mathbf{0}_\oplus$ **else** Receive(*Parent*);
  $y_{\mathrm{R}} \longleftarrow y_{\mathrm{L}} \oplus x_{\mathrm{L}} \oplus x$;
  if not *Leaf* **then** Send(y_{L}, *LeftChild*);
  if not *Leaf* **then** Send(y_{R}, *RightChild*);
  return(y_{R});
end Prefix

As mentioned above, the program makes two passes over the tree. The first pass 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 (x_{L} and x_{R}), computes the sum of $T(p)$, and passes the result to its parent. During the downward pass, each processor receives from its parent the sum over all processors with ids less than those in $T(p)$ (y_{L}), computes the sum over all processors with ids less than those in its right subtree (y_{R}), and sends the appropriate values to its left and right children (y_{L} and y_{R}). The correctness of the program is easily established by induction on the depth of the tree, and it runs in $4 \log p$ (all logarithms in this paper are base 2) 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 $2k + 4 \log p$ time steps.

2.2 Hypercube

For the hypercube, the following FFT-like computation executes Prefix in $\log p$ time steps:

```
begin Prefix(⊕, x)
    y ⟵ x;
    for i ⟵ 0 to d − 1 do
        Send(y, i);
        if MyId_i = 0 then
            y ⟵ y ⊕ Receive(i);
        else
            temp ⟵ Receive(i);
            x ⟵ temp ⊕ x;
            y ⟵ temp ⊕ y;
        end if
    end for
    return(x);
end Prefix
```

The variable *MyId* holds the id of the processor, and $MyId_i$ denotes the ith bit of the id (the least significant bit is bit 0). The source and destination arguments of Send and Receive specify the bit position in which the two communicating processors differ.

The program runs in $\log p$ time steps, and functions in the following manner. In addition to the partial sums demanded by the Prefix operation, the total sum is computed at every processor. The local variables x and y accumulate the partial and total sums, respectively. For a hypercube consisting of a single processor, the computation is trivial. Given $p = 2^d$, $d \geq 1$, processors with associated x_i values, the program first recursively computes partial and total sums for the upper and lower halves of the values independently, and then exchanges the total sums between halves. This enables the revised partial sums for the upper half to be computed, as well as the new total sums.

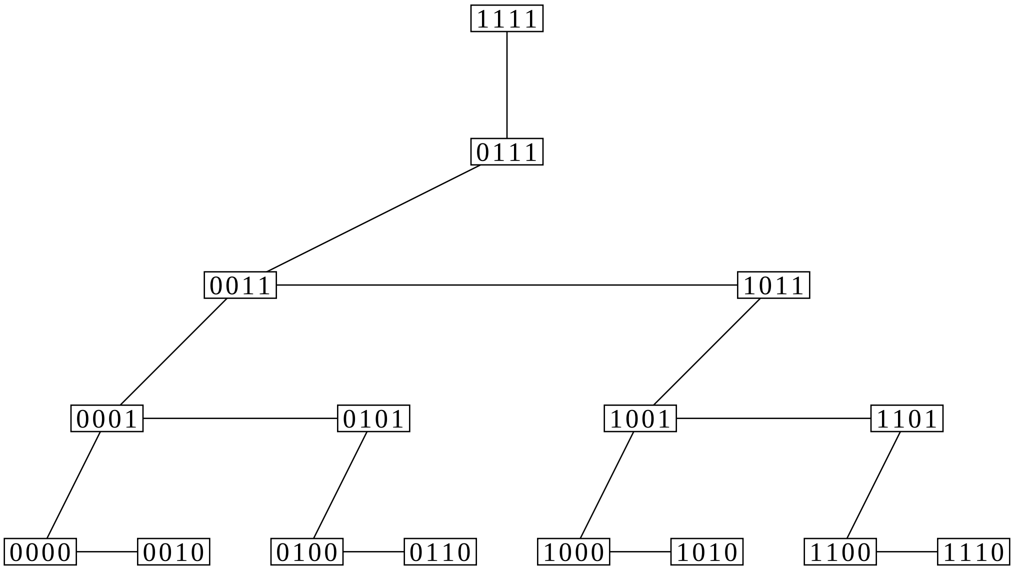

Figure 2: Embedding the inorder binary tree in the hypercube.

Unfortunately, the above program does not lead to a pipelined implementation of the Prefix operation 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 [5]. Figure 2 gives this embedding for the case $p = 16$, where the "extra" processor (with id $p - 1$) has been added as an extra level above the root. The edges depicted in Figure 2 are physical hypercube edges. 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 given for the inorder complete binary tree in Section 2.1 can be modified to run in the same time bound on the dilation 2 inorder complete binary tree embedding. In particular, note that processor $p - 1$ is in an appropriate location to receive the sum over all of the other processors. To summarize, we have shown that k Prefix operations can be performed in $2k + 4 \log p$ time steps on the hypercube.

2.3 Shuffle Exchange

The hypercube code given in the preceding section for performing a single Prefix operation can be easily adapted to the shuffle exchange:

```
    begin Prefix(⊕, x)
(1)   y ⟵ x;
(2)   repeat d times
(3)       Send(y, Exchange);
(4)       if MyId_0 = 0 then
(5)           y ⟵ y ⊕ Receive(Exchange);
(6)       else
(7)           temp ⟵ Receive(Exchange);
(8)           x ⟵ temp ⊕ x;
```

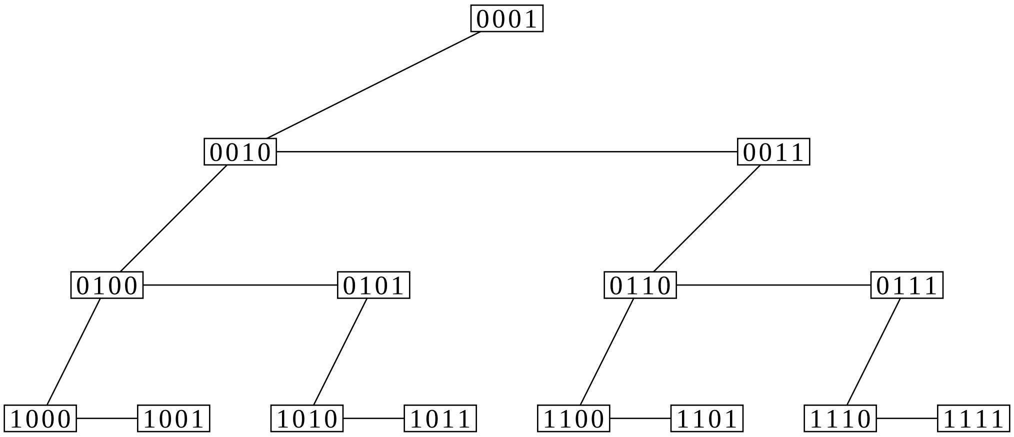

Figure 3: A shuffle exchange embedding for the high-numbered processors.

```
            y ⟵ temp ⊕ y;
        end if
        Send(x, Unshuffle);
        x ⟵ Receive(Shuffle);
        Send(y, Unshuffle);
        y ⟵ Receive(Shuffle);
    end repeat
    return(x);
end Prefix
```

The above program runs in $3 \log p$ time steps. As we saw for the hypercube, however, a different approach is needed in order to obtain a pipelined implementation of the Prefix operation. Unfortunately, it is not possible to embed the inorder complete binary tree in the shuffle exchange with constant dilation. Instead, we make use of the dilation 2 complete binary tree embeddings depicted, for the case $p = 16$, in Figures 3 and 4. The leaves of the tree in Figure 3 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 4 is defined in a similar fashion, and has 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, use the embedding of Figure 3 to compute the k sets of partial sums over the high-numbered processors. This takes $2k + 4 \log p$ time steps. Similarly, the embedding of Figure 4 can be used to perform k prefix sums over the low-numbered processors in $2k + 4 \log p$ time steps. At this point, all that remains to be done is 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 $2k + 2 \log p$ time steps using the embedding of Figure 4 (note that the desired sums are already available at the root), so k Prefix operations can be executed in $6k + 10 \log p$ time steps on the perfect shuffle.

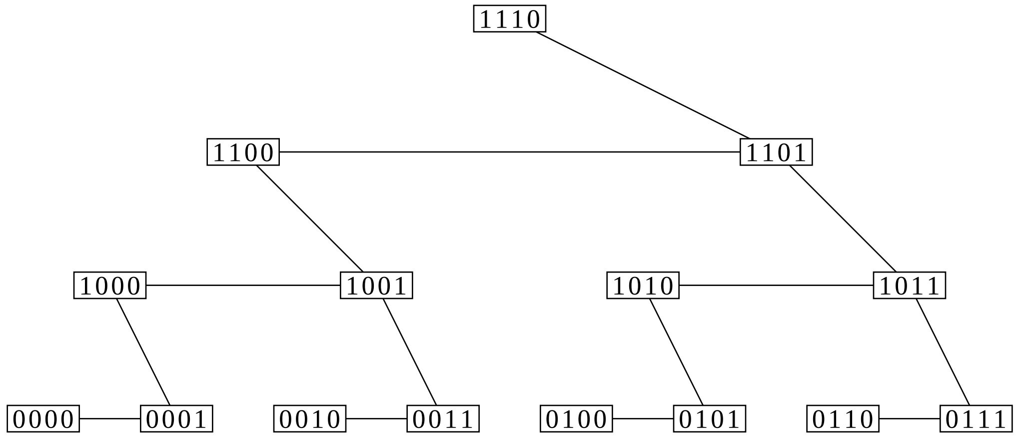

Figure 4: A shuffle exchange embedding for the low-numbered processors.

2.4 A Useful Variation

In Section 4 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. Note that all of our implementations of Prefix may be trivially modified to implement Prefix′ with precisely the same time bounds. For example, in the complete binary tree program of Section 2.1, it suffices to change the return value from y_{R} to $y_{\mathrm{L}} \oplus x_{\mathrm{L}}$.

3 Data Distribution

Consider the binary associative operator $\oplus$ defined over $\mathcal{X}$ by $x \oplus y = x$, for all $x, y \in \mathcal{X}$. 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 $2k + 4 \log p$ time steps on the tree or hypercube, and in $6k + 10 \log p$ time steps on the perfect shuffle.

In order to fully illustrate the techniques discussed in Section 1, we now study the implementation of segmented Prefix with the Copy operation in greater detail. As stated in Section 1, processor i initially holds the boolean value a_i and $x_i \in \mathcal{X}$, $0 \leq i < p$. Note that under the Copy operation the only relevant x_i's are those for which the corresponding a_i is **true**.

Clearly, there is no identity element for the Copy operation in $\mathcal{X}$. To remedy this situation, we extend the domain of Copy from $\mathcal{X}$ to $\mathcal{B} \times \mathcal{X}$ and define every pair with first component **false**, say, to be an identity element. In practice, this corresponds to prepending a single bit

b_i to each of the x_i's. 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 $x_0, x_1 \in \mathcal{X}$.

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

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

Dropping the inner parentheses and simplifying, this amounts to

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

Note that the above formulation allows bit pipelining in the sense described by Blelloch [6]. In other words, as each bit of the two operands is received, the next output bit can be computed. This holds not only for the Copy operator, but also for any other single-pass operator, as defined in [6].

Finally, we observe that the data distribution operation defined by Ullman [16] 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 network families.

4 Sorting on a Pipelined Hypercube

In this section, we describe a simplified implementation of the optimal merging algorithm of Varman and Doshi [17], and show how this can be used to develop a pipelined version of the sorting algorithm of Nassimi and Sahni [10] for a *pipelined model* of the hypercube.

The Sort operation is defined as follows. Given n $O(\log p)$-bit values, with $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ at any processor, rearrange the n values so that every value in processor i is less than or equal to every value in processor j whenever $0 \le i < j < p$. 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. For the 1-port model of the hypercube that we have been considering thus far, see [1], [4], [7], [9], [10] and [12]. For examples of results based on other assumptions, we refer the reader to [13], [15], [17] and [18].

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 d-port model, in which a processor can send and/or receive a packet from each of its $\log p$ neighbors in a single time step. This model, which we refer to as the *pipelined hypercube model*, was originally defined by Varman and Doshi [17], and we refer the reader to their paper for both the strict definition 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 [10], and it is easy to show that k such operations can be performed in $k + \log p$ 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 $\Omega(k \log^{1/2} p)$ 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 $O(p \log^{-1/2} p)$ processors with an equal number of 0's and 1's (or one more 0 than 1, say, if $\log p$ is odd), which implies a lower bound of $\Omega(k \log^{1/2} p)$ 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 applies to the pipelined concentration and inverse concentration operations as well.

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 [17], but is somewhat simpler. The optimal merging algorithm of Anderson, Mayr and Warmuth for the EREW PRAM also takes a similar approach [2]. For expository purposes, we make the (avoidable) assumption that all of the $2pk$ input values are distinct. 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, $0 \le i < p$. 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, $0 \le i < p$. 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, $0 \le i < p$, and must be sorted locally.

Our approach is to first merge X' and Y', and then 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', $0 \le j < 2p$. 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}, and the set Y_i corresponding to the smallest y_i that is greater than z_{2j+1}. These observations lead to the following pipelined merging algorithm.

Algorithm Merge

1. Reverse the list Y', that is, route y_i to processor $p-i-1$, $0 \le i < p$. This takes $\log p$ time steps.

2. Merge X' and Y' by simulating a bitonic merge over $2p$ processors. Record the data movements to facilitate the "unmerge" of step 3. This takes $2\log p$ time steps.

3. Route the rank of each value in Z' back to the processor which originally held that value. This can be done in $2\log p$ time steps by following the paths recorded in step 2 in the reverse direction.

4. Route each set X_i to the processor that held x_i after step 2, $0 \le i < p$. The id of that processor can be computed from the rank received by processor i in step 3. The routing can be performed in $2k + 2\log p$ time steps using a pipelined inverse concentration. Route the Y_i's in a similar fashion, for a total cost of $4k + 4\log p$ time steps.

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}, $0 \le i < p$. This can be done in $2k + 4\log p$ time steps with a single application of the Prefix$'$ operation, as described in Section 2.

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 $j_i - 1$, $0 \le i < p$. This can be done with a single application of a "backwards" version of Prefix$'$, and takes $2k + 4\log p$ 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 $y_i > z_{2j+1}$, $0 \le j < p$. 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.

Note that only step 4 uses the power of the pipelined model. The total running time of Merge is $8k + 17\log p$ time steps. Now consider the case in which $2p$ processors are available to perform the merge, where we assume that X_i is initially stored at processor i, Y_i is initially stored at processor $2p - i - 1$, and Z_j is to be output at processor j, $0 \le i < p$, $0 \le j < 2p$. In this case, step 1 is unnecessary, and the cost of each of the steps 2, 3 and 4 is halved, while the cost of the remaining steps is unchanged. Thus, the total cost of Merge with $2p$ processors is $6k + 12\log p$ time steps. Note that for $k = \Omega(\log p)$, 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 $\Omega(\log p)$. 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

$$\sum_{0 \le i < \log p} (6k + 12i) = 6k \log p + O(\log^2 p).$$

As mentioned above, this running time is optimal for $k = \Omega(\log p)$.

We now describe a pipelined version of the multi-way merging procedure of Nassimi and Sahni [10] 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 $O(k + \log p)$ time steps on a hypercube with $p = 2^{2l+m}$ processors. Let the ith input list be denoted X^i, $0 \le i < 2^l$, 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, $0 \le j < 2^m$. 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, $0 \le j < 2^{l+m}$.

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). Note that all of the X^i_j's are stored in row 0. In fact, each processor in row 0 contains exactly one set X^i_j.

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}, $0 \le i < p$, $0 \le j < k$. The output is the k sums $\sum_{0 \le i < p} a_{ij}$, $0 \le j < k$, all of which are output at a single designated processor. Although such operations can be performed using Prefix, other implementations exist which are more efficient by a constant factor. For example, using the multiple spanning binomial tree (MSBT) embedding of Ho and Johnsson [8] it is possible to perform k broadcasts in $k + \log p$ time steps. Similarly, k sums can be performed in $k + \log p$ time steps. Note that although these operations are pipelined, they run on the 1-port model and thus do not require the additional power of the pipelined model.

Algorithm MultiWayMerge

1. Broadcast X^i_j to all of the processors in column $i2^m + j$, $0 \le i < 2^l$, $0 \le j < 2^m$. 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, $0 \le i < 2^l$. In other words, route a copy of X^i_j to each column of the ith row that is congruent to $j \bmod 2^m$. This amounts to performing pipelined broadcasts over subcubes of dimension l, which can be done in $k + l$ time steps using the MSBT embedding.

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 $(j+1)2^m - 1$), $0 \le i, j < 2^l$, $i \ne j$. This takes $8k + 17m$ time steps.

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), $0 \leq i, j < 2^l$, $i \neq j$. 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). In terms of data paths, this is essentially an inverse pipelined broadcast operation over a subcube of dimension l, and it can be performed in $k + l$ time steps using the MSBT embedding.

Only steps 3, 4 and 6 require the power of the pipelined model. 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 on successively larger subcubes, we can obtain a fast sorting algorithm for the case $n < p \log p$. The running time of this algorithm, which we refer to as MultiWayMergeSort, will be shown to be $O(\log^2 p / \log((p \log p)/n)))$, as opposed to $O(\log^2 p / \log(p/n))$ for the sorting algorithm of Nassimi and Sahni. For the interesting case $n = p$, the running time of MultiWayMergeSort is $O(\log^2 p / \log \log p)$, a slight asymptotic improvement over that of Batcher's bitonic sort. It must be emphasized, however, that MultiWayMergeSort only runs on the pipelined model of the hypercube.

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).

Algorithm MultiWayMergeSort

1. Each column of the array contains k values. Route all of these to the top of the column (row 0). As in step 7 of MultiWayMerge, this takes $k + l$ time steps.

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

3. 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, for a total cost of approximately $(14k + 4l + 12m)m/l$ time steps.

4. 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 $k + \log p$ time steps.

Steps 3 and 4 make use of the power of the pipelined model. The total running time of MultiWayMergeSort is minimized (to within a constant factor) by setting $k = \log p$, and for this choice of k the running time is dominated by the cost of step 3. Observing that $l = \log(pk/n)$ and $m = \log p - l \leq \log p$, we find that for $k = \log p$ the algorithm runs in $\frac{47}{2}\log^2 p/\log((p\log p)/n) + O(\log p)$ time steps. For the case $n = p$, we can set $k = \log p/\log\log p$ and reduce the dominant term in the running time to $\frac{19}{2}\log^2 p/\log\log p$, at the expense of increasing the error term to $O((\log p/\log\log p)^2)$.

5 Concluding Remarks

In this paper, we have presented simple and efficient pipelined implementations for the Prefix operation on the complete inorder binary tree, hypercube and shuffle exchange families of networks. This led immediately to an elegant pipelined implementation of Ullman's data distribution primitive. A variant of the Prefix was used to obtain a simplified implementation of Varman and Doshi's optimal merging algorithm for the pipelined model of the hypercube.

In order to better assess the practical speed of the various algorithms presented in this paper, we have computed the coefficient on 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 4 could be improved with only minor modifications to the code.

It should be mentioned that for permutation routing, an important special case of the sorting problem, there is a much simpler $O(\log^2 p/\log\log p)$ time algorithm for the case $n = p$ than MultiWayMergeSort [11]. The idea is to route packets in a greedy fashion over sets of $\log\log p$ dimensions at a time. Each set of routings produces a load balancing problem in which there may be as many as $\log p$ packets at any one processor, and the objective is to redistribute the packets so that there is exactly one at each processor. It is a worthwhile exercise to show how this redistribution can be performed in $O(\log p)$ time on the pipelined hypercube by making use of the pipelined prefix, broadcast and concentration operations discussed in this paper.

References

[1] A. Aggarwal and M.-D. A. Huang. Network complexity of sorting and graph problems and simulating CRCW PRAMs by interconnection networks. In J. H. Reif, editor, *Lecture Notes in Computer Science: VLSI Algorithms and Architectures (AWOC 88)*, vol. 319, pages 339–350. Springer-Verlag, 1988.

[2] 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.

[3] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference,* vol. 32, pages 307–314, 1968.

[4] G. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers*, C–27:84–87, 1978.

[5] 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, 1986.

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

[7] R. E. Cypher and J. L. C. Sanz. Cubesort: An optimal sorting algorithm for feasible parallel computers. In J. H. Reif, editor, *Lecture Notes in Computer Science: VLSI Algorithms and Architectures (AWOC 88),* vol. 319, pages 456–464. Springer-Verlag, 1988.

[8] 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.

[9] S. L. Johnsson. Combining parallel and sequential sorting on a Boolean n-cube. In *Proceedings of the 1984 IEEE International Conference on Parallel Processing*, pages 444–448, 1984.

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

[11] D. Peleg. Personal communication.

[12] C. G. Plaxton. Load balancing, selection and sorting on the hypercube. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 64–73, 1989.

[13] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *JACM*, 34:60–76, 1987.

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

[15] S. R. Seidel and W. L. George. Binsorting on hypercubes with d-port communication. Technical Report CS–TR 99–01, Michigan Technological University, January 1988.

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

[17] 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.

[18] B. Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, pages 292–299, 1986.