Parallel Algorithms for Bucket Sorting and the Data Dependent Prefix Problem

Robert A. Wagner
Yijie Han

Department of Computer Science
Duke University
Durham, NC 27706

Abstract

The data dependent prefix problem is to compute all the n initial products $x_1 \bigcirc x_2 \bigcirc ... \bigcirc x_k$, $1 \leq k \leq n$, where the order is specified by a linked list. A parallel algorithm for the data dependent prefix problem is presented. This algorithm has time complexity $O(\frac{n}{p} + \log n \log \frac{n}{p})$ using p processors on the exclusive-read exclusive-write computation model. A bucket sorting algorithm is also developed to be used as a component of the prefix algorithm. This bucket sorting algorithm sorts n numbers in the range $\{1, 2, ..., m\}$ using p processors in $O(\lceil \frac{\log m}{\log(\frac{n}{p} + \log p)} \rceil (\frac{n}{p} + \log p))$ time.

Index words – Prefix, bucket sorting, parallel algorithms, graph algorithms, optimal algorithms.

1. Introduction

Parallel algorithms have been studied in many different areas. Some serial algorithms are easily parallelized while some other serial algorithms seem difficult to parallelize. The study of parallel algorithms reveals many properties of the problems which were previously unknown. In the serial case, time complexity is usually the most important complexity measure for algorithms. In the parallel case, the processor complexity also comes into play. To measure how much parallelism of the problem has been exploited by an algorithm the parameter of *speedup* is defined as $\frac{T_1}{T_p}$, where T_i is the time complexity of the algorithm using i processors and T_1 is the time complexity of the fastest known sequential algorithm for the same problem. It is not difficult to see $\frac{T_1}{T_p} \leq p$. When $T_p = \Theta(\frac{T_1}{p})$ we say the parallel algorithm achieves linear speedup with p processors. In the parallel environment algorithms with maximum speedup and algorithms with optimal performance (linear speedup) are both sought. As the number of processors grows the utilization of the processors tends to decrease and linear speedup algorithms become difficult to obtain. The contribution of the research presented in this paper is a new parallel algorithm for the data dependent prefix computation. This algorithm achieves linear speedup allowing asymptotically more processors than previously known algorithms. The time complexity of our algorithm improves the best previous result. A bucket sorting algorithm used as a component of the prefix algorithm is also presented. This sorting algorithm shows better performance than known algorithms.

The shared memory computation model is assumed in this paper. In a shared memory model both memory cells and processors are linearly ordered and each memory cell can be addressed by any processor. Three shared memory models are distinguished according to the ways they resolve read or write conflicts[10]. The Concurrent-Read Concurrent-Write model(CRCW), which is the strongest model among these three, allows simultaneous read and write of one memory cell by several processors. The content resulting after a cell is written by more than one processor can be determined in a variety of ways. In the Concurrent-Read Exclusive-Write(CREW) model several

*Work reported herein was partially supported by the National Science Foundation under grant MCS - 8202610.

processors can simultaneously read one memory cell, but simultaneous write is prohibited. The weakest model among three, the Exclusive-Read Exclusive-Write(EREW) model, prohibits both simultaneous read and write. Our algorithms are EREW and CREW algorithms.

Let $\bigcirc$ denote an associative operation. Given x_1, $x_2, ...,$ x_n, the (data independent) prefix problem is to compute $x_1 \bigcirc x_2 \bigcirc ... \bigcirc x_k$, $1 \leq k \leq n$. The data dependent prefix problem is to compute $x_1 \bigcirc x_2 \bigcirc ... \bigcirc x_k$, $1 \leq k \leq n$, for a given linked list containing $x_1, x_2, ..., x_n$, with x_i following x_{i-1}. For both cases, the product computation problem is to compute $x_1 \bigcirc x_2 \bigcirc ... \bigcirc x_n$.

The prefix problem is a fundamental problem in parallel computation. It finds applications in many problems such as processor allocation and reallocation, data alignment, data compaction, sorting and polynomial evaluation. The data dependent prefix problem is especially important since linked lists have been used widely as a data structure in many algorithms and prefix computation is probably the most basic and fundamental computation for linked list manipulation. One example is to use the prefix algorithm to number the data items in a linked list so that each item knows how far it is from the first element of the linked list (compute list rank for each element). This numbering operation is extremely useful when one intends to apply certain parallel operations to the linked list.

Studies have been conducted on the parallel prefix problem[2][5][7][8][9]. These studies assume that the input data is data independent in the sense x_i is in memory cell i. For the data dependent case, where input data forms a linked list and a map specifies which element follows which, optimal algorithms are also pursued. The best previous result is due to Kruskal, Rudolph and Snir[6] which achieves time complexity $O(\frac{\log n}{\log(2n/p)}\frac{n}{p})$[*] using p processors on the EREW model. In this paper we show that time complexity $O(\frac{n}{p} + \log n \log\frac{n}{p})$[**] can be obtained on the EREW model. Our result achieves linear speedup whenever $p = O(\frac{n}{\log n \log\log n})$. This shows improvement over the best previous result[6] which achieves linear speedup only when $p < n^{\epsilon}$, $0 \leq \epsilon < 1$. When our algorithm uses the maximum number of processors for which it can achieve linear speedup its time complexity becomes $O(\log n \log\log n)$, a polynomial in $\log n$. In contrast the best previous algorithm[6] can achieve a time complexity of $O(n^{1-\epsilon})$ using the maximum number of processors for which the algorithm obtains linear speedup. Our algorithm is the first result which shows that linear speedup can be achieved for the data dependent prefix problem with poly-logarithmic time complexity.

Parallel sorting problem has been studied extensively. See [1] for a survey of this area. For our purpose, a sorting algorithm is required for sorting n numbers in the range $\{1, 2, ..., \log n\}$ with p processors in time $O(\frac{n}{p} + \log p)$. Parallel sorting algorithms mentioned in [1] do not meet our requirement. Parallel comparison sorting algorithms are not useful here because they will take at least $\Omega(\frac{n\log n}{p})$ time with p processors. Hirschberg's bucket sorting algorithm [1][4] also implies a time bound of $O(\frac{n\log n}{p})$. Kruskal, Rudolph and Snir claimed a sorting algorithm which sorts n numbers in the range $\{1, 2, ..., n\}$. The time complexity of their algorithm is $O(\frac{\log n}{\log(2n/p)}(\frac{n}{p}+\log p))$ with p processors. Application of their sorting algorithm to our problem would imply a time bound of $O(\frac{\log n}{\log(2n/p)}(\frac{n}{p} + \log p))$ which is also too slow. We have developed our own bucket sorting algorithm. Our algorithm sorts n numbers in the range $\{1, 2, ..., m\}$ with p processors. The time

[*] Logarithms in this paper are to the base 2.

[**] We note here that $O(\frac{n}{p} + \log p) = O(\frac{n}{p} + \log n)$ and $O(\frac{n}{p} + \log p \log\frac{n}{p}) = O(\frac{n}{p} + \log n \log\frac{n}{p})$, for $p \leq n$.

complexity of our sorting algorithm is $O(\lceil \frac{\log m}{\log(\frac{n}{p} + \log p)} \rceil (\frac{n}{p} + \log p))$. Compared to the results of Hirschberg and Kruskal et. al., our result is more flexible in the sense that it adjusts itself better to the range of the numbers and the number of processors used.

2. Basic Fact and Definition

Product circuits, which are directed acyclic graphs, are also widely used as a parallel computation model. Some results may be obtained relatively easily using this model. We allow indegree of any node in the graph to be no more than 2. A node with indegree 2 represents a product of its two inputs. The nodes with indegree 0 are the input nodes. The depth of a circuit is the longest directed path in the graph. Fig. 1 shows two circuits for product computation. For the prefix computation the following fact is important to us.

Fact:

> Any circuit for product computation with the property that every node of the circuit represents a product of $\bigcirc_{k=i}^{j} x_k, 1 \leq i \leq j \leq n$, can be used to construct a circuit for the prefix problem, and the depth of the prefix circuit is within a constant factor of the depth of the original product circuit.

For the product computation, the circuits can be viewed as trees, which we will call product trees, with sons pointing to fathers. For the class of product trees satisfying the condition stated in the above fact we can define, for each non-input node, its left son as the node which feeds $\bigcirc_{k=i}^{m} x_k$ to it and right son which feeds $\bigcirc_{k=m+1}^{j} x_k$ to it.

The fact comes from the following observation. To compute the prefix, each node representing $\bigcirc_{k=1}^{j} x_k, 1 < j \leq n$, passes its product to its father. Each of these fathers then passes the product received to its right son. Each interior node getting a product from its father will transfer it to its left son and then compute a new product by combining the product passed from its father with the product accumulated by its left son during the process of computing the product. This new product will be passed to the interior node's right son. Fig. 2 shows this process.

This fact essentially shows that, for the serial and parallel (data independent) case, the prefix problem is no harder than the product problem. This fact is noted by Schwartz[9] and many other researchers and is used to construct parallel (data independent) prefix algorithms. This fact also applies to the data dependent prefix problem as it was used by Kruskal et. al. to construct their algorithm. Our algorithm also uses this fact, i.e. we are going to construct a parallel product algorithm with the product tree which the algorithm implicitly builds satisfying the condition stated in the fact. Thus our algorithm can be easily transformed to compute the prefix. This is accomplished by, when the product computation finishes, reversing the computation and popping out the product stored during the process of product computation.

Now we give definitions for some concepts related to linked list.

Definition:

A linked list with n elements is a data structure whose n elements are associated with n consecutive memory locations, where these n memory locations are linked one after another by arcs. The last arc in the list points to a special symbol *nil* denoting the end of the list. An arc (a, b) indicates that the element associated with location a is followed by the element associated with location b. This arc is represented by value b in location a. a is the tail of the arc and b is the head of the arc.

3. Previous Results for the Prefix Problem

It is advantageous here to mention some of the previous work on the prefix problem, since our algorithm uses some of the previous known algorithms as components and our work represents a continuation of the effort of these researchers.

The following well-known algorithm can be used to solve the prefix problem.

```
PREFIX_INDEPENDENT ($X[0..n-1]$);
    begin
        for $k := 0$ to $\lceil \log n \rceil - 1$ do
            forall $i$: $2^k \leq i < n$ do
                $X[i] := X[i - 2^k] \bigcirc X[i]$;
    end.
```

Algorithm PREFIX_INDEPENDENT has a corresponding version for the data dependent case, which is expressed by Wyllie[11]:

```
PREFIX_DEPENDENT($X[0..n-1]$,$NEXTX[0..n-1]$,$HEAD$);
    forall $i$: $0 \leq i < n$ do
        begin
            $temp := HEAD$;
            if $NEXTX[i] \neq nil$ then $NEXTX[NEXTX[i]] := i$
            else $HEAD := i$;
            $NEXTX[temp] := nil$;
            for $k := 1$ to $\lceil \log n \rceil$ do
                if $NEXTX[i] \neq nil$ then
                    begin
                        $X[i] := X[i] \bigcirc X[NEXTX[i]]$;
                        $NEXTX[i] := NEXTX[NEXTX[i]]$;
                    end
        end.
```

If n processors are available both algorithms will take $O(\log n)$ time. However, in no situation will they achieve linear speedup. This occurs because the total number of operations N involved in these algorithms is $O(n \log n)$. With p processors the best one can hope for is $O(\frac{n \log n}{p})$.

Kruskal et. al. [6] discussed an improved version of PREFIX_INDEPENDENT which achieves linear speedup. The elements are partitioned into p blocks where block i is $X[i \cdot \frac{n}{p} .. (i+1) \cdot \frac{n}{p} - 1]$, $0 \leq i < p$. Processor P_i is assigned to block i to calculate the prefix (and therefore the product p_i) of block i. Then PREFIX_INDEPENDENT can be used to obtain the prefix p_i^* for the p products. Finally p_i^* is added to each element of block $i+1$ by processor P_i.

However, no technique similar to the one mentioned above is known for algorithm PREFIX_DEPENDENT.

A handy improvement can be made to achieve $O(\frac{n \log p}{p})$ time with p processors. Fig. 3 shows that a linked list can be transformed into a linked tree of p branches, each with length no more than $\frac{n}{p} + 1$. This can be done in $O(\frac{n}{p} \log p)$ time with p processors by repeatedly assigning $NEXTX[i] := NEXTX[NEXTX[i]]$. This is equivalent to executing PREFIX_DEPENDENT for $\lceil \log p \rceil$ loops. Thereafter each processor works on one branch. The timing will be $O(\frac{n \log p}{p})$.

Kruskal, Rudolph and Snir's result shows that with p processors $O(\frac{\log n}{\log(2n/p)} \frac{n}{p})$ time can be achieved for the data dependent prefix computation[6]. The hard part of the problem is to process arcs, in an order that guarantees that the arcs being simultaneously processed are not chained together. By dividing the n elements into $\frac{n}{p}$ blocks, they apply p processors to visit the $\frac{n}{p}$ blocks one at a time. During this phase, only arcs leaving a block are processed. Since no processor will look at the head of the arcs leaving the block which the processors are visiting, it is ensured that arcs simultaneously processed are not chained together. After this all the inter-block arcs are treated so the whole problem is divided into $\frac{n}{p}$ equal size subproblems. By distributing processors evenly to all the subproblems and applying recursion all the arcs will be processed. This is one pass of the so-called Pair-Off. After pair-off each remaining element must have both of its neighbors eliminated, thus the number of remaining elements is at most $\frac{2n}{3}$. Elements are compacted and another pass of pair-off is applied. The number of elements left after each pass will form a geometric series and consequently, time complexity of $O(\frac{\log n}{\log(2n/p)} \frac{n}{p})$ is obtained.

4. A Parallel Bucket Sorting Algorithm

In this section a parallel bucket sorting algorithm is presented. This algorithm sorts n numbers in the range $\{1, 2, ..., m\}$ using p processors. The sorting requires time $O(\lceil \frac{\log m}{\log(\frac{n}{p} + \log p)} \rceil (\frac{n}{p} + \log p))$.

To sort n elements in the range $\{1, 2, ..., m\}$ with p processors, we divide the elements into $\lceil \frac{n}{p} \rceil$ blocks. These blocks will be visited by the p processors sequentially. Suppose element i valued v_i is visited by processor P_k, the element will be dropped into the $((v_i - 1) \cdot p + k)$-th bucket. After all the elements are dropped into buckets, packing the elements will yield the sorted sequence. The details of the algorithm are shown below.

```
BUCKETSORT($A[1..n]$) /* $1 \leq A[i] \leq m$, $1 \leq i \leq n$. */

    processors: $P_i$, $1 \leq i \leq p$;
    array:      $S[1..m \cdot p]$, /* Buckets */
                $C[1..n]$,     /* Counter for each element to
                                  calculate its rank.
                               */
                $G[1..m]$;     /* $G[i]$ is the total number
                                  of occurrences of elements
                                  valued $i$.
                               */

    forall $P_i$: $1 \leq i \leq p$
        begin
```

```
/* Initialization. Takes $O(m)$ time.*/
    $S[1..m \cdot p] := 0$;

/* Throwing elements into buckets.
    Takes $O(\frac{n}{p})$ time.
*/
    for $k := 1$ to $\lceil \frac{n}{p} \rceil$ step 1
        begin
            $C[(k-1) \cdot p + i] := S[(A[(k-1) \cdot p + i] - 1) \cdot p + i]$;
            $S[(A[(k-1) \cdot p + i] - 1) \cdot p + i] :=$
                $S[(A[(k-1) \cdot p + i] - 1) \cdot p + i] + 1$;
        end

/* $G[i]$ is going to be returned to the calling
    procedure. Takes $O(m + \log p)$ time.
*/
    forall $k$: $1 \le k \le m$ do $G[k] := \sum_{i=(k-1) \cdot p+1}^{k \cdot p} S[i]$;
/* Packing. Takes $O(\frac{n}{p} + m + \log p)$ time. */
    PREFIX_INDEPENDENT($S[1..m \cdot p]$);
    for $k := 1$ to $\lceil \frac{n}{p} \rceil$ step 1
        begin
            $C[(k-1) \cdot p + i] := C[(k-1) \cdot p + i]$
                $+S[(A[(k-1) \cdot p + i] - 1) \cdot p + i - 1]$;
        end
end
```

For n elements ranging from 1 to m, BUCKETSORT takes $O(\frac{n}{p} + m + \log p)$ time units with p processors. When $m > \frac{n}{p} + \log p$, we can use the idea of radix sorting. The $\log m$ bits representing the numbers are divided into $\lceil \frac{\log m}{\log(\frac{n}{p} + \log p)} \rceil$ blocks. Starting from the least significant block of bits, BUCKETSORT can be applied. After applying BUCKETSORT $\lceil \frac{\log m}{\log(\frac{n}{p} + \log p)} \rceil$ times, once for each block of bits, the sorting is accomplished. The total time will be $O(\lceil \frac{\log m}{\log(\frac{n}{p} + \log p)} \rceil (\frac{n}{p} + \log p))$ since the m in the BUCKETSORT is $\frac{n}{p} + \log p$ now.

We note here that this sorting algorithm can also be used to pack data. Assigning 1 to each marked element and 2 to each unmarked element, execution of this sorting algorithm will pack the marked elements to the beginning of the array and unmarked elements to the end of the array.

5. The Main Algorithm

Our algorithm for the data dependent prefix problem is presented in this section. This algorithm uses the observation stated before, i.e. an appropriate product computation algorithm can be used to construct an algorithm for prefix computation. Thus we are focusing on constructing a suitable algorithm for product computation. The idea of Kruskal et. al. that appropriate pairs of elements are combined such that one pass of combining (or 'pair-off') will reduce the number of elements by at least one third is followed here. However, instead of using p processors to cut n elements into $\frac{n}{p}$ blocks we observed that the n arcs in the linked list can be divided into $O(\log n)$ groups and all the arcs in one group can be eliminated simultaneously by one 'pair-off' pass. The details of this idea are expressed below.

We divide arcs into groups by the following rule: arc (a, b) is in group $2k - a_k$ where

$$k = \min\{i \mid \text{the } i\text{-th bit of } a\ XOR\ b \text{ is } 1\},$$

XOR is the exclusive-or operation, bits are counted from the least significant bit starting at 1, a_k is the k-th bit of a. We prove the following lemma.

Lemma:

a) Each arc belongs to one and only one group.
b) For any group G, if two different arcs $(a, b) \in G$ and $(c, d) \in G$, then a, b, c, d are all distinct.

Proof:

Part a) is obvious from the way we form groups. Now we prove part b).

Assume G is an arc group indexed by $2i - a_i$. $a \neq c$ since no two arcs can originate from one element, and $b \neq d$ since no two arcs can have the same target. Since both arcs are in G, $a_i = c_i$. Now by the definition of grouping, $b_i = d_i = (a_i\ XOR\ 1)$. Hence a, b, c, d are all distinct.

Q.E.D.

By the above lemma, we know pairs of elements connected by arcs in a group can be combined simultaneously without interfering with each other. Since at most $\lfloor \log n \rfloor + 1$ bits are used to represent the elements in the linked list, at most $2\lfloor \log n \rfloor + 2$ groups are needed. To determine, for arc (a, b), which group it belongs to, we compute

```
$c := a\ XOR\ b$;
$c := (((c-1)\ XOR\ c) + 1)/2$;
```

This will set all the bits of c to zeros except the lowest bit valued 1. Now we use value c to index into a table T to determine which bit of c is 1. Table T can be built by the following simple procedure.

```
TABLE$(n, p)$
    begin
        for $j := 0$ to $\log n$ step 1
            $T[2^j] := j + 1$;
    end.
```

It is possible that several processors attempt to read the same entry of the above table simultaneously. We can use this table on the CREW model anyway. The time for table building will not dominate the overall time complexity. The procedure for determine the arcs' group membership is as follows.

```
MEMBER$(X[1..n]$, $NEXTX[1..n])$
    forall $P_i$: $1 \leq i \leq p$
        begin
            for $k := 1$ to $\frac{n}{p}$ step 1
                begin
                $M[i] := X[(i-1)\frac{n}{p}+k]\ XOR\ NEXTX[(i-1)\frac{n}{p}+k]$;
                $M[i] := (((M[i]-1)\ XOR\ M[i])+1)/2$;
                $A[(i-1)\frac{n}{p}+k] := 2 \cdot T[M[i]]$;
                if $(M[i]\ AND\ X[(i-1)\frac{n}{p}+k]) > 0$ then
                    $A[(i-1)\frac{n}{p}+k] := A[(i-1)\frac{n}{p}+k] - 1$;
            end
        end.
```

After the group of each arc is calculated, the bucket sorting algorithm is used to move arcs so that arcs in the same group are gathered together. Now, one group at a time, all the arcs are visited. This will take no more than $O(\sum_{i=1}^{2\log n+2} \lceil \frac{|G(i)|}{p} \rceil) = O(\frac{n}{p} + \log n)$ time, where $|G(i)|$ is the number of arcs in group i. Specifically, when the head of an arc visited by a processor is already deleted due to the previous processing of that arc's predecessor then the processor does nothing, otherwise it combines the head and the tail of the arc, marks the tail deleted, and updates the arc to point to the successor of its successor. After one pass of this 'pair-off', any remaining element must have both its predecessor and successor marked deleted. So at most $\frac{2n}{3}$ elements can survive. The remaining elements are packed and after $O(\frac{\log n}{p})$ passes, there are no more than p elements left. Finally the remaining elements are combined by algorithm PREFIX_DEPENDENT. The following gives the details of our algorithm for the CREW model.

```
PRODUCT_PREFIX($X[1..n]$, $NEXTX[1..n]$, $HEAD$)
    begin
        TABLE($n, p$);
        COMBINE($X[1..n]$, $NEXTX[1..n]$, $HEAD$);
    end

COMBINE($X[1..n]$, $NEXTX[1..n]$, $HEAD$);
    forall $P_i$: $1 \leq i \leq p$
        begin
            $DELETED[1..n] := false$;
            if $n \leq p$ then
                PREFIX_DEPENDENT($X[1..n]$, $NEXTX[1..n]$, $HEAD$);
            else
                begin
                    /* Calculating group membership. */
                    MEMBER($X[1..n]$, $NEXTX[1..n]$);

                    /* Bring elements of the same group together.*/
                    /* BUCKETSORT returns each element's rank
                        in array $C$.
                    */
                    BUCKETSORT($A[1..n]$);
                    for $k := 1$ to $\frac{n}{p}$ step 1
                        $B[C[(i-1)\frac{n}{p}+k]] := (i-1) \cdot \frac{n}{p} + k$;
                    /* Visiting arcs one group at a time. */
                    /* Array $G$ is returned by BUCKETSORT.
                        $G[k]$ is the total number of elements
                        in group $k$.
                    */
                    $j := 1$;
                    for $k := 1$ to $2\log n + 2$ step 1
                        begin
                            while $G[k] \geq p$ do
                                begin
                                    PAIR-OFF($B[j..j+p]$);
                                    $j := j + p$;
                                    $G[k] := G[k] - p$;
                                end
                            if $G[k] \neq 0$ then PAIR-OFF($B[j..j+G[k]]$);
                            $j := j + G[k]$;
                        end

                    /* Now pack. PACK is a simplified version
                        of BUCKETSORT. PACK returns the size
                        of the packed array.
                    */
                    $n'$:=PACK($A[1..n]$);
                    COMBINE($X[1..n']$, $NEXTX[1..n']$, $HEAD$);
                end

PAIR-OFF($B[a..b]$)
    begin
        if $b - a < p$ then Disable $P_i$: $i > b - a + 1$;
        if NOT $DELETED[B[a+i-1]]$ then
            begin
                $X[B[a+i-1]] := X[B[a+i-1]] \bigcirc$
                            $X[NEXTX[B[a+i-1]]]$;
                $DELETED[NEXTX[B[a+i-1]]] := true$;
                $NEXTX[B[a+i-1]] :=$
                            $NEXTX[NEXTX[B[a+i-1]]]$;
            end
        Enable $P_i$: $1 \leq i \leq p$;
    end
```

Each execution of procedure MEMBER takes $O(\frac{n}{p})$ time and each execution of procedure BUCKETSORT takes $O(\frac{n}{p} + \log n)$ time, for the n numbers range from 1 to $2\log n + 2$. As mentioned before, visiting groups of arcs takes $O(\frac{n}{p} + \log n)$ time, and packing can be easily verified to take $O(\frac{n}{p} + \log p)$ time. Thus one pass of the combining process, i.e. an execution of procedure COMBINE, takes $O(\frac{n}{p} + \log n)$ time. After executing COMBINE no more than $\frac{2n}{3}$ elements are left, and hence the following recursive formula for time complexity T_p can be established.

$$T_p(n) = O(\frac{n}{p} + \log n) + T_p(\frac{2n}{3}).$$

Since $O(\frac{\log n}{p})$ passes of COMBINE are used and the remaining less than p elements can be combined in $O(\log p)$ time units, we have $T_p(n) = O(\frac{n}{p} + \log n \log \frac{n}{p})$. All the operations in the algorithm except the ones for calculating the group membership of the arcs are EREW operations.

For the EREW model, one way to determine the group membership for an arc (a, b) is to continuously bisect the bits of $a\ XOR\ b$ and ask if the lower half of the bits are 0's. In this binary splitting fashion, we can calculate the group membership for all the arcs in time $O(\frac{n}{p} \log\log n)$. The $\log\log n$ factor can be eliminated by using p copies of table T, one copy for each processor. This table TE, which is of size $n \cdot p$, has only $(\log n + 1) \cdot p$ entries which could possibly be indexed into. These entries are $TE[i][j]$, $1 \leq i \leq p$, $j = 2^k$, $0 \leq k \leq \log n$. Table $TE[1..p][1..n]$ can be built by copying relevant entries of table T. Distribute processors such that P_i, $(k-1) \cdot \frac{p}{\log n + 1} + 1 \leq i \leq k \cdot \frac{p}{\log n + 1}$, $1 \leq k \leq \log n + 1$, are assigned to entries $TE[1..p][2^{k-1}]$. These processors will make p copies of the $T[2^{k-1}]$ and put them in $TE[1..p][2^{k-1}]$. The total copying operation will take $O(\log n)$ time. The procedure for determine arcs' group membership is obtained by merely replacing the statement

$$A[(i-1)\frac{n}{p} + k] := 2 \cdot T[M[i]]$$

in procedure MEMBER by

$$A[(i-1)\frac{n}{p} + k] := 2 \cdot TE[i][M[i]].$$

The table storage technique due to Fredman, Komlós and Szemerédi[3] can be used here to reduce the size of the index table to $O(p \cdot \log n)$ while retaining constant time for table lookup, i.e. $O(\log n)$ space is enough for one copy of the index table. Although the sequential time complexity of the algorithm[3] for building a single copy of the index table is $O(\log^4 n)$, $O(\frac{n}{p} + \log n \log \frac{n}{p})$ time is more than enough when p processors are employed to build a single copy of the index table. Thus, we can first create a single copy of the index table with the help of p processors, then duplicate the table to p copies. The time complexity of $O(\frac{n}{p} + \log n \log \frac{n}{p})$ still holds. The algorithm just obtained is an EREW algorithm.

As mentioned before, for the prefix computation we execute our algorithm first and then reverse the computation and pop out products stored at each node of the product tree.

6. Conclusion

Parallel algorithms allowing a maximum number of processors to achieve linear speedup for the data independent prefix problem have already been obtained, even on weaker models such as ultracomputers[9]. This is not the case in the data dependent case. We conclude this paper by raising the question: is $O(n/p + \log n)$ time bound achievable for the data dependent prefix problem on the EREW model?

References

[1]. D. Bitton, D.J. DeWitt, D.K. Hsiao and J. Memon. A taxonomy of parallel sorting. ACM Computing Survey, Vol. 16, No. 3, Sept. 1984, pp. 287-318.

[2]. F. Fich. New bounds for parallel prefix circuits. Proc. of the 15th Annual ACM symposium on Theory of computing, May 1983, pp. 100-109.

[3]. M.L. Fredman, J. Komlós, E. Szemerédi. Storing sparse table with $O(1)$ worst case access time. J. ACM, Vol. 31, No. 3, July 1984, pp. 538-544.

[4]. D.S. Hirschberg. Fast parallel sorting algorithms. Comm. ACM, Vol. 21, No. 8, Aug. 1978, pp. 657-661.

[5]. P.M. Kogge and H.S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Comput. Vol. C-22, Aug. 1973, pp. 786-792.

[6]. C.K. Kruskal, L. Rudolph, M. Snir. The power of parallel prefix. IEEE Tran. Computers, Vol. C-34, No. 10, Oct. 1985, pp. 965-968.

[7]. R.E. Ladner and M.J. Fischer. Parallel prefix computation. J. ACM, Oct. 1980, pp. 831-838.

[8]. J. Reif. Probabilistic parallel prefix computation. Proc. of 1984 International Conf. on Parallel Processing, Aug. 1984, pp. 493-443.

[9]. J.T. Schwartz. Ultracomputers, ACM Transactions on Programming Languages and Systems, Dec. 1980, pp. 484-521.

[10]. M. Snir. On parallel searching. SIAM J. Comput. Vol. 14, No. 3, Aug. 1985, pp. 688-708.

[11]. J.C. Wyllie. The complexity of parallel computation, TR 79-387, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

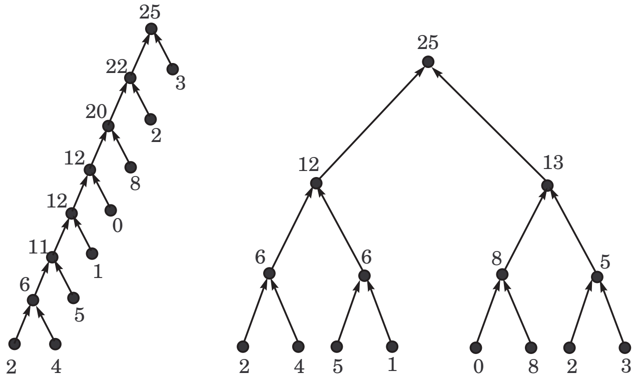

Fig. 1. Two product trees. $\bigcirc$ is addition here.

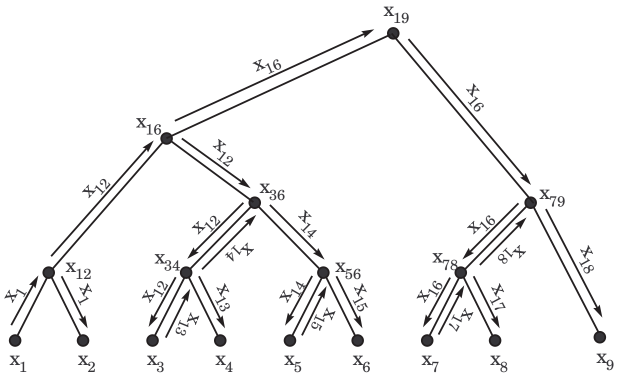

Fig. 2. Use a product tree to construct a prefix algorithm. x_{ij} denotes $\bigcirc_{k=i}^{j} x_k$.

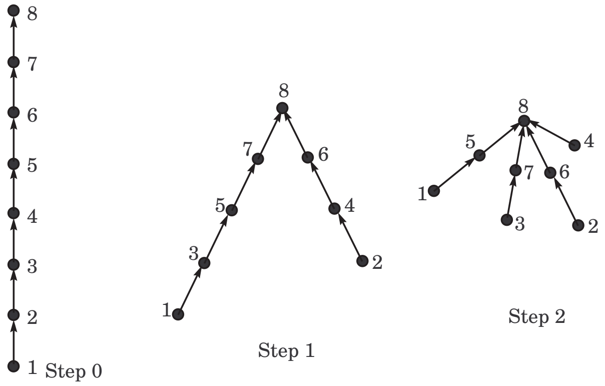

Fig. 3. A linked list of n items can be transformed into a tree of maximum depth $\frac{n}{p}+1$ in $O(\frac{n}{p}\log p)$ time, using p processors.