Matching Partition a Linked List and Its Optimization

Yijie Han

Department of Computer Science
University of Kentucky
Lexington, KY 40506

ABSTRACT

We show the curve $O(\frac{n\log i}{p}+\log^{(i)} n+\log i)$ for the time complexity of computing a maximal matching for a linked list, where n is the size of the input list, p is the number of processors used in the algorithm and i is an adjustable parameter. For all constructible i the time complexity represented by the curve can be realized. Our algorithm is optimal using up to $O(\frac{n}{\log^{(i)} n})$ processors with an arbitrarily large constant i. This algorithm can be used to compute a maximal independent set or a 3 coloring for a linked list.

1. Introduction

A matching set of a graph is a set of edges such that no two edges in the matching set are incident on the same vertex. The matching set is maximal if adding one more edge to the matching set makes it a nonmatching set. In a sense, to find a maximal matching set for a linked list in parallel is to break the parallel symmetrical situation of the linked list.

The parallel computation model used in this paper is the well known Parallel Random Access Machine[2] (PRAM). We say that a parallel algorithm is optimal if $p\cdot T_p = O(T_1)$, where p is the number of processors used in the parallel algorithm, T_p is the time complexity of the parallel algorithm and T_1 is the time complexity of the best known sequential algorithm. In this paper we shall use n to denote the size of the input, *i.e.*, the number of nodes in the linked list, and p to denote the number of processors used in an algorithm.

Although many previous linked list prefix algorithms [9,11,13,16] can be used to compute a maximal matching set for a linked list, they are either randomized algorithms or algorithms with time complexity not less than $O(\log n)$. A previous algorithm discovered independently by Han[6] and Cole and Vishkin[3] for computing a maximal matching set uses $p=n$ processors and achieves time complexity $O(G(n))$, where $G(n)=\min\{k|\log^{(k)} n<1\}$, $\log^{(k)} n$ is defined as: $\log^{(1)} n = \log n$, $\log^{(k)} n=\log(\log^{(k-1)} n)$, and as usual, the base of a logarithm is 2 if not specified. This algorithm is not optimal. An EREW[14] PRAM algorithm with time complexity $O(\frac{n}{p}+\log n)$ was also independently discovered by Han[6] and Cole and Vishkin[3]. This algorithm is optimal using up to $O(\frac{n}{\log n})$ processors. The main obstacle prohibiting this optimal algorithm using more than $O(\frac{n}{\log n})$ processors is a global sorting step which sorts integers of small magnitude. By using Reif's sublogarithmic partial sum algorithm[12], Han[6] showed that on the CRCW(concurrent read concurrent write)[14] PRAM the sorting can be done in $O(\frac{n}{p}+\frac{\log n}{\log^{(3)} n})$ steps, thus obtaining a CRCW maximal matching algorithm with time complexity $O(\frac{n}{p}+\frac{\log n}{\log^{(3)} n})$ which is optimal using up to $O(\frac{n\log^{(3)} n}{\log n})$ processors. Cole and Vishkin showed[4] that the partial sum can be computed in time $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n})$, thus yielding a CRCW maximal matching algorithm with time complexity $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n})$ which is optimal using up to $O(\frac{n\log^{(2)} n}{\log n})$ processors.

Recently Han[7] and Beame observed a faster algorithm for computing a maximal matching. Beame's observation was stated in the journal paper of Goldberg *et al.*[5]. This algorithm uses a table lookup technique and has time complexity $O(\frac{n\log G(n)}{p} + \log G(n))$. Han's version[7] gives a fast scheme for constructing the lookup table. Note that this algorithm is not optimal.

In this paper we present a new algorithm for computing a maximal matching set for a linked list. This algorithm is designed by applying an optimal processor scheduling technique to compute a maximal matching set from the matching partition obtained by the known matching partition algorithms. Combined with previous results we demonstrate the curve $O(\frac{n\log i}{p} + \log^{(i)} n + \log i)$ for the time complexity of computing a maximal matching for a linked list, where n is the size of the input list, p is the number of processors used in the algorithm and i is an adjustable parameter. We say that $i = i(n,p)$ is constructible if the time complexity of computing function $i(n,p)$ does not dominate the time complexity of the whole algorithm. Here, $i(n,p)$ is constructible if it can be computed in $O(\frac{n\log i}{p} + \log^{(i)} n + \log i)$ time. For all constructible i the time complexity represented by the curve can be realized. In particular, when i is a constant an algorithm with time complexity $O(\frac{n}{p} + \log^{(i)} n)$ can be obtained which is optimal using up to $O(\frac{n}{\log^{(i)} n})$ processors.

2. Matching Partition a Linked List

	0	1	2	3	4	5	6
X	x_0	x_2	x_4	x_1	x_5	x_3	x_6
NEXT	3	5	4	1	6	2	*nil*

Fig. 1. A linked list.

Let us assume that a linked list of n elements (nodes) is stored in an array $X[0..n-1]$ and $NEXT[0..n-1]$ is the array of pointers with $NEXT[i]$ pointing to the next element to $X[i]$, as shown in Fig. 1. For a node v in the linked list, we also denote the node following v in the list by $suc(v)$ and the node preceding v by $pre(v)$. We use $< a, b >$ to denote a pointer valued b in location $NEXT[a]$. b is the head of the pointer and a is the tail. A pointer $< a, b >$ is a forward pointer if $b > a$, otherwise the pointer is a backward pointer. By drawing a line c bisecting the array containing the linked list as shown in Fig. 2, we observe that forward pointers crossing line c have disjoint heads and tails. This is because no two pointers can have the same head or the same tail and the head of one pointer can not be the tail of the other pointer because both pointers are forward pointers crossing line c. We associate with bisecting line c two matching sets of pointers, one consisting of forward pointers crossing c, the other of backward pointers crossing c. The linked list array is divided into two sub-arrays by line c. We can draw bisecting lines c_1, c_2 for the two sub-arrays. Forward pointers crossing either c_1 or c_2 but not c have disjoint heads and tails. Continuing in this fashion it is not difficult to see that the pointers of the linked list can first be partitioned into two sets, a set of forward pointers and a set of backward pointers, and then each set can further be partitioned into $\lceil \log n \rceil$ matching sets, with pointers in one set having disjoint heads and tails. A close examination of the pointers crossing bisecting lines reveals that the function $g(< a, b >) = \max\{i|$ the i-th bit of $a\ XOR\ b$ is $1\}$, where XOR is the bit-wise exclusive-or operation and bits are counted from the least significant bit starting with 0, characterizes the set of pointers crossing bisecting lines (both forward and backward pointers). Function g can be modified to distinguish between forward and backward pointers. We define function $f(< a, b >) = 2k + a_k$, where $k = \max\{i|$ the i-th bit of $a\ XOR\ b$ is $1\}$ and a_k is the k-th bit of a. Note that a_k denotes whether $< a, b >$ is a forward pointer or a backward pointer.

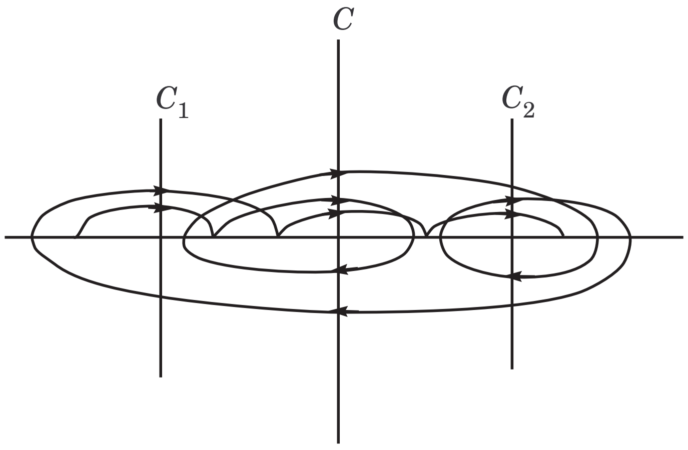

Fig. 2. The intuitive observation of bisecting.

Lemma 1[6]: Function f partitions the n pointers of a linked list into $2\lceil \log n \rceil$ matching sets. □

We say that function $m(a, b)$ is a matching partition function if $m(a, b) \neq m(b, c)$ whenever $a \neq b$ or $b \neq c$. Function f defined above is a matching partition function which partitions the n pointers of a linked list into $2\lceil \log n \rceil$ matching sets[6].

In [6,15] we also used the least significant bit[1] instead of the most significant bit for defining function f. In doing so, we gain the advantage for computing function f at the expense of losing intuition. Function f can be computed using a table lookup technique[6,15]. We shall deal with the details of the parallel computation of the matching partition functions in the appendix of this paper. Henceforce we shall assume that $f(a, b)$ can be computed in $O(1)$ time by a single processor.

Because on a linked list $b = suc(a)$, we also write $f(< a, b >)$ as $f(a, suc(a))$. If a is the last element in the list, we can define $f(a, suc(a)) = f(a, b)$, where b is (the address of) the first element of the linked list. After we assign the value of $f(a, suc(a))$ to node a and view it as the "new address" of the node, function f can be re-applied to obtain a coarser partition[6]. Define $f^{(2)}(a_1, a_2) = f(a_1, a_2), f^{(k)}(a_1, a_2, ..., a_k) = f(f^{(k-1)}(a_1, a_2, ..., a_{k-1}), f^{(k-1)}(a_2, a_3, ..., a_k))$. Function $f^{(k)}$ represents repeated applications of function f to the linked list.

Lemma 2[6]: Function $f^{(k)}$ partitions the pointers of a linked list into $2 \log^{(k-1)} n(1+o(1))$ matching

[1]Lemmas 1 to 4 are from [6]. The scheme of using the least significant 1-bit to obtaining a matching partition function and the results in Lemmas 2 to 4 were independently discovered in [3]. The intuitive observation presented here was observed exclusively by the author[6].

sets. □

We shall extend the definition of matching partition functions. We say that function $m^{(k)}$ is a matching partition function if $m^{(k)}(a_1, a_2, ..., a_k) \neq m^{(k)}(a_2, a_3, ..., a_{k+1})$ whenever there exist i and j such that $a_i \neq a_j$. Function $f^{(k)}$ is a matching partition function which partitions the n pointers of a linked list into $2\log^{(k-1)} n(1 + o(1))$ matching sets[6].

Remark: Recent studies[8] show that there exists a matching partition function $m^{(k)}(a_1, a_2, ..., a_k)$ which partitions the n pointers of a linked list into $\log^{(k)} n(1 + o(1))$ matching sets. Such a function may be more difficult to compute than the matching partition functions provided in Lemmas 1 and 2. However, it can be shown that no matching partition function $m^{(k)}(a_1, a_2, ..., a_k)$ can partition the pointers into less than $\log^{(k-1)} n$ matching sets[8,10].

When k reaches $G(n)$, $2\log^{(k-1)} n(1 + o(1))$ becomes a constant. Consequently, we obtained a maximal matching. The algorithm[6] for achieving this is shown below.

Algorithm Match1:

```
Step 1. For node $v$, $1 \leq v \leq n$, do $label[v]$ := address of $v$.

Step 2.
    for $i := 1$ to $G(n)$ do
        for node $v$, $1 \leq v \leq n$, do in parallel
                /*Compute $f^{(i+1)}$. */
            $label[v] := f(< label[v], label[suc(v)] >)$;

Step 3. If $label[pre(v)] > label[v]$ and $label[v] < label[suc(v)]$ then delete pointer $< v, suc(v) >$.
```

Comment: After step 3 the linked list is cut into many sublist each of them has constant number of nodes.

```
Step 4. Walk down each sublist to add every other pointer of the sublist to the matching set.
```

The matching set found by Algorithm Match1 is maximal because at least one of any three consecutive pointers of the linked list is in the matching. The time complexity of Algorithm Match1 is $O(\frac{nG(n)}{p} + G(n))$.

Lemma 3[6]: The n pointers of a linked list can be partitioned into $O(\log^{(i)} n)$ matching sets in $O(\frac{in}{p})$ time, where i is a constructible parameter. A maximal matching can be computed in $O(\frac{nG(n)}{p} + G(n))$ time. □

To obtain an optimal algorithm for computing a maximal matching set, we designed the following algorithm[6].

Algorithm Match2:

```
Step 1. Partition $n$ pointers into at most $\log^{(2)} n$ matching sets.

Step 2. Sort pointers by their set numbers so that pointers in one set are sequentially located.
Note that sorting is performed on integers in the range $\{0, 1, ..., \log^{(2)} n - 1\}$.

Step 3.
    begin
```

```
        $S := \phi$;
        for all nodes $i$ do in parallel:
            $DONE[i] := false$;

        for $k := 0$ to $\log^{(2)} n - 1$ do
            for all pointers $< a, b >$ in matching set $k$ do in parallel
                begin
                    if $DONE[a] = false$ AND $DONE[b] = false$ then
                        begin
                            $DONE[a] := true$;
                            $DONE[b] := true$;
                            $S := S \cup \{< a, b >\}$;
                        end
                end
    end
```

Lemma 4[6]: A maximal matching for a linked list can be computed in $O(\frac{n}{p} + \log n)$ time. □

The time complexity of Step 2 in Match2 dominates the whole algorithm. As shown in [6], by adapting Reif's CRCW partial sum algorithm to sort integers in the range $\{0, 1, ..., \log^{(2)} n - 1\}$, the time complexity of the sorting step can be improved to $O(\frac{n}{p} + \frac{\log n}{\log^{(3)} n})$. Cole and Vishkin[4] showed that partial sum can be done in time $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n})$, thus yielding a better algorithm for computing maximal matching.

The idea in Match2 is to obtain a matching partition, then combine partitioned sets into a maximal matching. The sorting step is used to schedule processors to process partitioned sets one by one. In the next section we show that this global sorting scheme is inefficient. We note that the inefficiency due to the global sorting and packing steps for computing linked list prefix on the EREW model was exposed[3,6]. Anderson and Miller[1] presented an elegant load balancing scheme to circumvent the repetitive global sorting and packing operations in the linked list prefix algorithm. The inefficiency due to the sorting step in Match2 seems to be more subtle. The fact that known algorithms for computing maximal matching are good enough for the design of a linked list prefix algorithm with timing $O(\frac{n}{p} + \log n)$ on the CRCW model[6] may have left the impression that the global sorting step in Match2 is efficient.

Recently Han[7] and Beame observed a faster algorithm for computing a maximal matching set. This algorithm has time complexity $O(\frac{n \log G(n)}{p} + \log G(n))$ and is shown below.

Algorithm Match3

Step 1. For node v, $1 \leq v \leq n$, do $label[v] :=$ address of v.

```
Step 2.
    for $i := 1$ to $k$ do
        for node $v$, $1 \leq v \leq n$, do in parallel
            $label[v] := f(< label[v], label[suc(v)] >)$;
```

Comment: k is an adjustable parameter. What step 2 accomplishes is number crunching. After step 2 $label[v]$ can be represented using no more than $\log^{(k)} n$ bits.

Step 3.

```
for $i := 1$ to $\log G(n)$ do
    for node $v$, $1 \leq v \leq n$, do in parallel
        begin
            $label[v] := label[v]label[NEXT[v]]$;
            $NEXT[v] := NEXT[NEXT[v]]$;
        end
```

Comment: $label[v]label[NEXT[v]]$ is the concatenation of the bits of $label[v]$ and the bits of $label[NEXT[v]]$. After step 3 $label[v]$ has $G(n)\log^{(k)} n$ bits.

Step 4. For node v, $1 \leq v \leq n$, do $label[v] := T[label[v]]$.

Comment: Step 4 does table lookup. After step 4 each $label[v]$ is a constant not related to n. Besides, $label[v] \neq label[suc(v)]$.

Step 5. Step 3 of Match1.

Step 6. Step 4 of Match1.

A copy of the lookup table T is of size $2^{G(n)\log^{(k)} n}$. The largest k we can use for this algorithm is $O(\log G(n))$. As shown in [7], a copy of table T can be constructed in constant time using n processors on the CRCW model when k is greater than 4. Note that table T contains the function values of a matching partition function. We shall deal with the details of the parallel computation of the matching partition functions in the appendix of this paper.

Lemma 5: The n pointers of a linked list can be partitioned into $O(\log^{(i)} n)$ matching sets in $O(\frac{n \log i}{p} + \log i)$ time, where i is a constructible parameter. A maximal matching can be computed in $O(\frac{n \log G(n)}{p} + \log G(n))$ time. □

Note that in order to prove the first statement in Lemma 5 we execute $O(\log i)$ iterations of the loop in step 3 of Match3 and initialize the contents of the lookup table with the function values of an appropriate matching partition function.

3. Optimization in Computing a Maximal Matching

In order to study the processor scheduling on the linked list, we view the linked list of n nodes being stored in a two dimensional array of x rows and y columns. y processors are used, each one works on one column. A pointer $< a, b >$ is called an intra-row pointer if both a and b are on the same row, otherwise the pointer is called an inter-row pointer. Note that by rearranging the pointers in this array we may convert certain inter-row pointers to intra-row pointers and certain intra-row pointers to inter-row pointers. We have the following lemma.

Lemma 6(Algorithm WalkDown1): If all the pointers of a linked list are inter-row pointers, then a maximal matching can be computed in $O(x)$ steps with y processors.

Proof: We let y processors walk down the columns of the array, starting at row 0 and proceeding to row $x-1$. In a step, let $< a, b >$ be the pointer to be processed by processor p. Processor p checks to see whether and what numbers pointers $< pre(a), a >$ and $< b, suc(b) >$ are labeled, and label pointer $< a, b >$ with a number in $\{0, 1, 2\}$ such that the label is different from the labels assigned to pointers $< pre(a), a >$ and $< b, suc(b) >$. Because all pointers are inter-row pointers, when processor p is working on pointer $< a, b >$, no processor is working on either of the pointers

$< pre(a), a >$, $< b, suc(b) >$. Thus the labelling of pointer $< a, b >$ by processor p can be done in constant time.

After the labelling, we have obtained a partition of 3 matching sets. Executing steps 3 and 4 of Match1 yields a maximal matching set. □

For any input linked list, we can always use Lemma 6 to partition the inter-row pointers into 3 matching sets. If we have a scheme to partition the intra-row pointers into 3 matching sets, we would have a scheme of partitioning the pointers of the whole list into 3 matching sets. Note that minor adjustment is needed in combining the partitions for the inter-row and intra-row pointers.

We now outline a processor scheduling scheme. Suppose that each element of a two dimensional array of x rows and y columns is labeled with an integer from $\{0, 1, ..., x-1\}$. Let each column of the array be sorted in the ascending order by these labels. Allocate one processor to each column and each processor walks down the column in the following manner:

Algorithm WalkDown2:

Let processor p walk down the array $A[0..x-1]$ containing the sorted labels.

```
$count$ := 0;
$index$ := 0; /*Initialize $count$ and $index$ which
            are local variables of processor $p$. */
for $i$ := 0 to $2x - 2$ /*Step $i$*/
   if $index \le x - 1$ then
      begin
         if $A[index] = count$ then
            begin
               $A[index] := MARKED$;
                          /*Mark the array element*/
               $index := index + 1$; /*Walk to next row*/
            end
         else /*Idling in the same row
               but increment the count*/
            $count := count + 1$;
      end
```

We say that a processor is in row r at step k if $A[r]$ is marked at the k-th iteration (starting with 0-th iteration) of the loop indexed by i.

Lemma 7: In an execution of WalkDown2, processor p is in row r at step k iff $A[r] = k - r$ before it is marked.

Proof: Each iteration of the loop indexed by i increments either variable *count* or variable *index* when *index* is not greater than $x - 1$. It takes $k - r$ iterations to raise the value of *count* to $k - r$ and another r iterations to raise the value of *index* to r. Therefore $A[r]$ is marked at step k. □

Corollary 1: After the execution of WalkDown2, every element of A is marked.

Proof: Since $A[x-1] \le x - 1$, processor p is in row $x - 1$ at step $\le 2x - 2$. □

When WalkDown2 is executed in parallel, processors need to be synchronized at the beginning of each step. We assume that this is implied by the algorithm.

Corollary 2: At step k, $0 \le k \le 2x - 2$, all processors in the same row have the same $A[index]$ value before $A[index]$ is marked.

Proof: All processors in row r have $A[index] = k - r$ at step k. □

Our main algorithm follows.

Algorithm Match4:

Step 1. Compute a partition of $\log^{(i)} n$ matching sets for the input linked list, where i is an adjustable parameter.

Step 2. View the array containing the input linked list as a two dimensional array with $x = \log^{(i)} n$ rows and $y = \frac{n}{x}$ columns. Allocate one processor to each column. Each processor then sorts pointers in its column by the pointers' matching set numbers (numbers in $\{0, 1, ..., \log^{(i)} n - 1\}$). The sorting can be done by using a sequential integer sorting scheme.

Step 3. Call WalkDown1 to partition the inter-row pointers into 3 matching sets.

Step 4. Use WalkDown2 to partition the intra-row pointers into 3 matching sets.

Step 5. Execute steps 3 and 4 of Match1 to obtain a maximal matching.

Step 4, the key step of the algorithm, may need further explanation. Note that in step 4 only intra-row pointers are treated. Corollary 2 says that all processor in the same row at a step have the same $A[index]$ value. This implies that pointers in the same row which are processed in a step by processors in that row at that step are in the same matching set, therefore no pair of these pointers is incident on the same node. This ensures that processors process these pointers can independently choose matching set numbers from $\{0, 1, 2\}$ to label these pointers. When WalkDown2 is used to partition the intra-row pointers, we need only replace the statement $A[index] := MARKED$ with the statement "process the pointer at that memory cell". Here "process the pointer" indicates the same operation performed on a pointer in WalkDown1, *i.e.*, label it with a number in $\{0, 1, 2\}$ such that the labeling differs from the labeling of the predecessor pointer and the successor pointer. A trace of step 4 would show that intra-row pointers are processed in a pipelined fashion.

Note that step 1 of Match4 takes $O(\frac{n \log i}{p} + \log i)$ steps and the sequential sorting step can be done in time $O(x)$, where x is the number of rows. Thus we obtain:

Theorem 1: A maximal matching set for a linked list can be computed in optimal time using up to $O(\frac{n}{\log^{(i)} n})$ processors, where i is an arbitrarily large constant. $\Box$

Combined with previous lemmas, we have:

Theorem 2: A maximal matching set for a linked list can be computed in time $O(\frac{n \log i}{p} + \log^{(i)} n + \log i)$ for any constructible i. $\Box$

4. Conclusions

The processor scheduling technique presented in this paper is powerful enough to yield an optimal algorithm with timing $O(t)$ for computing a maximal matching set for a linked list provided that the pointers of the list has already been partitioned into $O(t)$ matching sets. Since the best known algorithm with timing $O(\frac{n \log i}{p} + \log i)$ can partition a list into at least $\log^{(i)} n$ matching sets, the application of our scheduling technique stops here for obtaining optimal algorithms. It is not known whether a faster optimal algorithm exists for partitioning the pointers of a linked list. In particular, we do not know whether it is possible to partition pointers into $G(n)$ matching sets in $O(G(n))$ time using $\frac{n}{G(n)}$ processors.

References

[1]. R. J. Anderson, G. L. Miller. Deterministic parallel list ranking, Lecture Notes in Computer Science 319, VLSI Algorithms and Architectures(John Reif ed.), 3rd Aegean Workshop on Computing, 81-90(June-July, 1988).

[2]. R. A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parallel models of computation, Proc. 14th ACM Symposium on Theory of Computing, San Fransisco, 338-344(April, 1982).

[3]. R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms, Proc. 18th ACM Symp. on Theory of Computing, 206-219(1986).

[4]. R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems, 27th Symp. on Foundations of Comput. Sci., IEEE, 478-491(1986).

[5]. A. V. Goldberg, S. A. Plotkin, G. E. Shannon. Parallel symmetry-breaking in sparse graphs, SIAM J. on Discrete Math., Vol 1, No. 4, 447-471(Nov., 1988).

[6]. Y. Han. Designing fast and efficient parallel algorithms. Ph.D. dissertation. Dept. Computer Sci., Duke Univ., 1987.

[7]. Y. Han. An optimal linked list prefix algorithm on a local memory computer. Proc. of the Computer Science Conf. (CSC'89), 278-286(Feb., 1989).

[8]. Y. Han. On the chromatic number of shuffle graphs, Technical report, TR 140-89, Dept. of Computer Sci., University of Kentucky, 1989.

[9]. C. P. Kruskal, L. Rudolph, M. Snir. The power of parallel prefix, IEEE Trans. Comput., Vol. C-34, No. 10, 965-968(Oct., 1985).

[10]. N. Linial. Distributive graph algorithms — global solutions from local data, Proc. 1987 IEEE Annual Symposium on Foundations of Computer Science, 331-336(1987).

[11]. G. L. Miller, J. H. Reif. Parallel tree contraction and its application, 26th Symp. on Foundations of Computer Sci., IEEE 478-489(1985).

[12]. J. H. Reif. An optimal parallel algorithm for integer sorting, 26th Symp. on Foundations of Computer Sci., IEEE, 291-298(1985).

[13]. J. H. Reif. Probabilistic parallel prefix computation, Proc. of 1984 International Conf. on Parallel Processing, 493-443(Aug., 1984).

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

[15]. R. A. Wagner and Y. Han. Parallel algorithms for bucket sorting and the data dependent prefix problem, Proceedings 1986 International Conf. on Parallel Processing, 924-930.

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

Appendix

We discuss the details of the parallel evaluation of certain functions in this appendix. Note that these function values can be computed in the preprocessing stage and their time complexity can be excluded from the time complexity of our algorithms shown in the paper. However, there is always the desire that the computation of these functions be included in the algorithm and their time complexity be counted in the time complexity of the whole algorithm. We show how to evaluate these functions on the EREW model. In some cases we need the concurrent read feature to fan out the function values to all processors.

Two schemes have been proposed for computing the matching partition function $f_1^{(2)}(a,b) = 2k + a_k$, $k = \min\{i |$ the i-th bit of $a \; XOR \; b$ is $1\}$. The key step for evaluating $f_1^{(2)}$ is the operation of converting a unary number to a binary number. One may suggest building such an instruction into the computer[5]. This scheme leaves the question of whether each processor in a parallel computer should have the power of performing a number conversion in one step. An alternative is to use a lookup table to convert unary numbers to binary numbers[6,15]. This scheme is illustrated below:

```
$c := a\ XOR\ b$;
$c := c\ XOR\ (c-1)$;
  /*Converting the unary number in $c$ to
   a binary number yields $k$.*/
$c := (c+1)/2$;
$k := T[c]$;
```

It is straightforward to compute $f_1^{(2)}$ once the value of k is obtained. Note that the table T has only $\lceil \log n \rceil$ entries which are useful. To run our algorithms on the EREW model we need p copies of the table, one for each processor. It is impossible to create that many copies of T in $O(G(n))$ time. Therefore, to run Match1, Match3 and Match4 on the EREW model without building the number conversion instructions into the processors we need p copies of T to be set up in the preprocessing stage. Because p copies of table T can be created using $O(p \log n)$ space and $O(\frac{n}{p} + \log n)$ time on the EREW model[6,15], Match2 can be executed on the EREW model without any precomputation. To compute $f^{(2)}$ we can use a bit reversal permutation table to reverse the bits of a number so that the most significant bit becomes the least significant bit.

With the assumption that the conversion from unary to binary can be done in $O(1)$ step (either by assuming that such an instruction is built into each processor or by applying a table lookup technique), functions $\log^{(i)} n$, $G(n)$ and $\log G(n)$ can be evaluated. Here, the evaluation of function H should be interpreted as finding a number $m = \theta(H)$. Functions $\log^{(i)} n$ and $G(n)$ are used in Match1, Match3 and Match4. They can be evaluated as follows:

To evaluate $\log n$, let the binary representation of n be $a_k...a_2a_1$, compute $n' = a_1a_2...a_k$, which is the bit reversal permutation of n by using a table lookup technique. The following instructions evaluate $\log n$.

```
$n' := n'\ XOR\ (n'-1)$;
$n' := convert(n')$; /*Converting the unary number in $n'$
                       to a binary number.*/
$logn := k - n'$;
```

To evaluate $\log^{(i)} n$, we execute this procedure i times. Note that this is a sequential procedure. To evaluate $G(n)$ using one processor, we iterate the above procedure until the input n being log-ed into a constant, and then count how many iterations has been executed. This sequential procedure takes $O(G(n))$ time.

Function $\log G(n)$ is used in Match3. To evaluate $\log G(n)$ in $O(\log G(n))$ time we use a parallel procedure. We use array $N[1..n]$ and n processors. Processor i checks to see whether i is a power of 2. If i is a power of 2, processor i set $N[i] := \log i$, otherwise processor i set $N[i] := nil$. Processor 1 sets $N[1] := 1$. This creates many linked lists in array N. We call the one containing $N[1]$ the main list. We can evaluate $G(n)$ by computing the length of the main list. This can be done in $O(\log G(n))$ time by using the pointer jumping operation $N[i] := N[N[i]]$. The number of executions of the statement $N[i] := N[N[i]]$ needed to transform last pointer in the main list to point to 1 is an evaluation of $\log G(n)$. Therefore both $G(n)$ and $\log G(n)$ can be evaluated on the EREW model in $O(\log G(n))$ time using n processors.

We turn to the computation of $f^{(i)}(a_1, a_2, ..., a_i)$. It is now obvious that it can be done in $O(i)$ steps. To satisfy Lemma 5 and Theorem 2 we require that it be computed in $O(\log i)$ steps if this time complexity is not to be excluded from the time complexity of the whole algorithm. A scheme is given in [7] for constructing a table for a matching partition function with i arguments.

Construct a graph G as in [10] with each vertex of the graph denoted by an i-tuple $(a_1, a_2, ..., a_i)$, $a_j \in \{0, 1, ..., n-1\}$, $1 \le j \le i$. Vertices $(a_1, a_2, ..., a_i)$ and $(b_1, b_2, ..., b_i)$ are connected by an undirected edge iff $a_j = b_{j+1}$, $1 \le j < i$. A valid vertex coloring of G using $2\log^{(i-1)} n(1 + o(1))$ colors gives a table for a matching partition function. Such a valid coloring can be found in $O(1)$ time on a CRCW model using exponential number of processors[7]. Because of step 2 of Match3, we can reduce the size of the table and the number of processors used for constructing the table. As shown in [7], the adjustable parameter k can be adjusted so that the number of processors needed for constructing the table is less than n.

We now give a scheme to construct the table on the EREW model. To compute $f^{(i)}(a_1, a_2, ..., a_i)$, we use a table containing $\frac{i(i+1)}{2}$ cells. These cells are labeled with $a_p a_{p+1} ... a_{p+q}$, $1 \le p \le i$, $0 \le q \le i - p$. Cell a_p contains a_p. Cell $a_p a_{p+1} ... a_{p+q}$ is supposed to contain $f^{(q+1)}(a_p a_{p+1} ... a_{p+q})$. Now we guess these values and place them into cells and then verify them. A processor verifies the value of cell $a_p a_{p+1} ... a_{p+q}$ by computing function value $f^{(2)}$ using the values in cells $a_p a_{p+1} ... a_{p+q-1}$ and $a_{p+1} a_{p+2} ... a_{p+q}$. If the value in cell $a_p a_{p+1} ... a_{p+q}$ is equal to the function value the processor writes "Yes" into the cell, otherwise it writes "No". The function value we guessed for $f^{(i)}(a_1, a_2, ..., a_i)$ is correct if all the cells contain "Yes". This can be checked in $O(\log i)$ time using a binary tree to fan in all the cell values. In order not to miss the correct value for $f^{(i)}(a_1, a_2, ..., a_i)$, we shall enumerate all possible situations. To set up the table we compute the $f^{(i)}$ value for all possible arguments in parallel. This would require exponential number of processors if $a_j \in \{0, 1, ..., n\}$. Again step 2 of Match3 can be used to reduce the size of the table. Consequently the number of processors needed can be reduced to less than n. Note that because there is only one correct guess for $f^{(i)}(a_1, a_2, ..., a_i)$ no concurrent read or write is needed in setting up a copy of the table.