Parallel Algorithms for Linked List and Beyond

Yijie Han

Department of Computer Science
University of Kentucky
Lexington, KY 40506, USA

Abstract

Linked list problems are fundamental problems in the design of parallel algorithms. We present design techniques for linked list algorithms and applications of these algorithms.

1. Introduction

Linked list is a basic data structure in the design of computer algorithms. The conceptual simplicity of a linked list often leads algorithm designers to believe that it is a trivial structure to be treated. A text in computer algorithms will not treat problems such as finding a maximal independent set on a linked list as a formal topic for the design of a sequential algorithm because the problem is trivial in the sequential case. However, much research interest arouse in the past few years on the design of parallel algorithms for linked list and many efficient parallel algorithms emerged by using efficient linked list algorithms as a critical subroutine. In this paper we summarize these efforts by outlining the main difficulties once presented in the design of efficient parallel algorithms for linked list, by providing the main thread leading to overcoming these difficulties and by demonstrating the power of linked list algorithms through the applications of these algorithms.

The first step in the design of a parallel algorithm is to choose a model. A PRAM (Parallel Random Access Machine)[BH] is usually preferred because the model allows almost uninhibited communication between processors and memory cells. This enables an algorithm designer to focus his attention on the intrinsic parallelism of the problem instead of the architecture of the model. A PRAM has a set of processors and a set of memory cells. Both processors and memory cells are linearly indexed. In a step, a processor in a PRAM can read or write any memory cell. Conflicts arising from simultaneous read or write to the same memory cell by several processors are resolved by imposing additional rules to the PRAM[Sn]. In an EREW (Exclusive Read Exclusive Write) PRAM, simultaneous access to the same memory cell is prohibited. In a CREW (Concurrent Read Exclusive Write) PRAM simultaneous read to the same memory cell is allowed and all processors reading the same cell simultaneously will obtain the value from the cell. Simultaneous write to the same cell is prohibited on the CREW PRAM. In a CRCW (Concurrent Read Concurrent Write) PRAM both simultaneous read and simultaneous write are allowed. The resulting value in a memory cell after simultaneous write to the cell can be determined according to certain stipulations. In a COMMON CRCW PRAM, simultaneous write is allowed only in the situation that all processors write the same value to the same cell and the resulting value of the cell is this common value. In an ARBITRARY CRCW PRAM, an arbitrary processor among the processors attempting to simultaneous write the same cell succeeds in writing the cell. There are other conflict resolving rules for the CRCW PRAM. Chlebus *et al.* [CDHR] is a paper studying these variations of CRCW PRAMs.

CRCW PRAM represents the strongest version of the PRAM model while the EREW PRAM is the weakest version among three. In certain situations the PRAM models can be further weakened by limiting the number of shared memory cells it can use. A PRAM with p processors allowed only p shared memory cells is equivalent to the local memory PRAM[AM1] or the DCM (Direct Connection Machine) model[KMR].

Two fundamental problems related to linked list are the maximal matching problem and the prefix problem.

A matching set is a set of edges (pointers) with no two pointers incident with the same node. A matching is maximal if it is not a subset of any other matching set. For the input of a linked list the maximal matching problem asks the output of a maximal matching set of the list. For a linked list the maximal matching problem is equivalent to the maximal independent set problem (where a maximal independent set of nodes is to be computed) in the sense that a PRAM algorithm for computing a maximal matching for a linked list can be used for computing a maximal independent set with the same time complexity and vice versa.

For the data items $a_0, a_1, ..., a_{n-1}$ on the input linked list and an associative operation $\bigcirc$, the linked list prefix problem asks the output of prefixes b_0, b_1, ..., b_{n-1}, where $b_j = \bigcirc_{i=0}^{j} a_i$.

The input linked list is assumed to be stored in two arrays $X[0..n-1]$ and $NEXT[0..n-1]$. The data items associated with the nodes of the linked list are stored in array X and pointers of the linked list is stored in array $NEXT$.

Both problems are trivial in the sequential case. They are not trivial in the parallel case where many parallel algorithm design techniques have been invented and tested on the two problems.

2. Linked List Algorithms

2.1. Pointer Jumping

A technique due to Wyllie[W] for handling linked list is the technique of pointer jumping (also called recursive doubling). The application of this technique in solving the prefix problem can be described by the following procedure.

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

The effect of pointer jumping is shown in Fig. 1.

This technique is powerful in that it solves the prefix problem in $O(\log n)$ time when n processors are available. However it loses efficiency compared to a sequential algorithm

with time complexity $O(n)$ because it uses a total of $O(n \log n)$ operations or instructions.

Substantial research efforts have been put on cutting the factor of $\log n$ in order to obtain a parallel algorithm using optimal number $O(n)$ operations (algorithms using optimal number of operations are called optimal algorithms). All these efforts follow the same approach, namely contracting the linked list by computing a large matching set for the list and pairing off nodes in the list[MR].

2.2. Computing a Matching Set Using Randomization

Miller and Reif used randomization to obtain a large matching set. This is done by using a random number generator at each node of the list. Each random number generator independently generates a 0 or a 1 with equal probability. If the head of a pointer gets a 1 and the tail of the pointer gets a 0, the head and the tail of the pointer can be paired. For other possible random values ((0, 0), (0, 1), (1, 1) at the head and the tail) the head and the tail will not be paired. It can be shown[MR] that at least $(1-\epsilon)n/8$ nodes will be in the matching set with probability of failure less than $e^{-3\epsilon^2 n/2^7}$. Due to the power of randomization such a matching set can be obtained in constant time.

2.3. The Matching Partition Function

A function for partition the pointers of a linked list into $O(\log n)$ matching sets can be derived from the following intuitive observation[H1].

For a linked list of n elements stored in an array $X[0..n-1]$, $NEXT[0..n-1]$ is the array of pointers with $NEXT[i]$ pointing to the next element to $X[i]$, as shown in Fig. 2. 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. 3, 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 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 $g(< a,\ b >) = \max\{i \mid$ 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 $k = \max\{i \mid$ 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.

Theorem 1[H1]: Function f is a matching partition function which partitions the pointers of a linked list into $2\lceil \log n \rceil$ matching sets. □

It is possible to have a matching partition function which partitions the pointers of a linked list into less than $2\lceil \log n \rceil$ matching sets. Let $g(n) = \binom{n}{\lfloor n/2 \rfloor}$. We have

Theorem 2[H5]: There is a matching partition function f which partitions $g(n)$ pointers of a linked list into n matching sets.

Proof: We show, by induction, the existence of a f which partitions $\binom{n}{i}$ pointers of a linked list into n matching sets such that possible outgoing pointers of a node are in no more than i matching sets.

To establish the base of the induction we show the existence of a f which partitions $\binom{n}{1}$ pointers of a linked list into n matching sets such that possible outgoing pointers of a node are in one matching set. We achieve this by putting the outgoing pointer of node v in matching set v, $0 \leq v \leq n-1$. We have used n matching sets and no matter which node is the head of the pointer with its tail at node v, the pointer is always in one set, *i.e.*, set v.

To obtain f for a list with $\binom{n}{i}$ nodes, we first bisect the list into two parts as shown in Fig. 4, the first part L_1 contains $\binom{n-1}{i}$ nodes and the second part L_2 contains $\binom{n-1}{i-1}$ nodes. We obtain two matching partition functions f_1 and f_2 for L_1 and L_2, respectively, by induction hypothesis. We combine f_1 and f_2. The remaining pointers need to be put into matching sets are those pointers with their heads in L_1 and tails in L_2 (or heads in L_2 and tails in L_1). By induction hypothesis, outgoing pointers in L_1 can be assigned to at most i matching sets and outgoing pointers in L_2 can be assigned to at most $i-1$ matching sets. Therefore for a pointer p with head h in L_2 and tail t in L_1, there is a matching set which can be assigned to possible outgoing pointers at h but is never assigned to possible outgoing pointers at node t, we assign this matching set to pointer p. For any pointer with head in L_1 and tail in L_2 we put it into a new matching set which is a set not used in L_1 and L_2. By now we have obtained f which partitions $\binom{n}{i}$ pointers of a linked list into n matching sets such that possible outgoing pointers of a node are in no more than i matching sets.

Setting $i = \lfloor n/2 \rfloor$ proves the theorem. □

By Stirling's approximation we see the existence of a matching partition function f which partitions n pointers of a linked list into $(1 + o(1)) \log n$ matching sets.

Theorem 2 is the best we could possibly do, as shown in the following theorem.

Theorem 3 [H5]: For $g(n)$ pointers in a linked list n is the minimum on the number of matching sets obtainable by any matching partition function $f(x, y)$.

Proof: Given a matching partition function f, let $S(a) = \{f(a, b) \mid 0 \leq b < g(n)\}$ and $P = \{S(a) \mid 0 \leq a < g(n)\}$. For two different nodes a and b, $S(a) \not\subseteq S(b)$, for otherwise there exists a node c such that pointers $< a, b >$ and $< b, c >$ will be put into the same matching set. Therefore P is a Sperner's family[Sp] and the correctness of the theorem follows since a Sperner's family of $g(n)$ sets has at least n elements. □

The matching partition function can be generalized as follows. Define $\log^{(1)} n = \log n$, $\log^{(k)} n = \log(\log^{(k-1)} n)$, $G(n) = \min\{k \mid \log^{(k)} n < 1\}$. Also define $f^{(2)}(a_1, a_2) = f(a_1, a_2)$, $f^{(k)}(a_1, a_2, ..., a_k) = f(f^{(k-1)}(a_1, a_2, ..., a_{k-1}), f^{(k-1)}(a_2, a_3, ..., a_k))$. $f^{(k)}$ represents repeated applications of f to the linked list.

Theorem 4 [H5]: There is a matching partition function $f^{(k)}$ which partitions n pointers of a linked list into $\log^{(k-1)} n(1 + o(1))$ matching sets.

Proof: By Theorem 2 and definition of $f^{(k)}$. □

On the other hand, we have:

Theorem 5[H5]: A lower bound on the number of matching sets any $f^{(k)}$ can obtain is $\log^{(k-1)} n$. □

Lower bounds similar to that shown in Theorems 3 and 5 were obtained by Linial[L]. His bounds are off by a multiplicative factor from optimal. Upper and lower bounds obtained in [H5] are exact for $f^{(2)}$ and tight up to minor terms for $f^{(k)}$ for $k > 2$, as shown in Theorems 2 to 5.

Since $f^{(G(n))}$ gives constant number of matching sets, if node a knows up to $G(n)$ nodes following it in the linked list it can label itself with a matching set number chosen from a constant number of matching sets. This yields a parallel algorithm for maximal matching with time complexity $O(\frac{n \log G(n)}{p} + \log G(n))$, in this paper p is the number of processors used in an algorithm. This algorithm is due independently to Han[H4] and Beame. Han's version provides a scheme for constructing a lookup table for $f^{(G(n))}$ [H4].

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

2.4. Processor Scheduling

A processor scheduling scheme was provided by Han[H3] which yields processor efficient algorithm for finding a maximal matching set for a linked list. .

To obtain an optimal algorithm with time complexity $O(x)$ using $y = n/x$ processors, we may view the linked list as being stored in a two dimensional array with x rows and y columns. We can thus assign one processor to each column.

It is easy to see[H3] that if all pointers are inter-row pointers, *i.e.*, pointers with their heads and tails on different rows of the two dimensional array, then the pointers can be partitioned into three matching sets by letting the processors walking down the rows of the array. The difficult situation we have to handle is the situation where there are many intra-row pointers, *i.e.*, pointers with their heads and tails on the same row of the two dimensional array.

Assume that all pointers are intra-row pointers and that the pointers are already being partitioned into x matching sets. Further assume that all pointers in a column are in the same matching set. The processor scheduling works as follows. If pointers in column c are in matching set m, the processor assigned to column c will idle for m steps before it walks down column c. This processor scheduling guarantees that no processors will contend on any pointer.

The last paragraph presents the intuition behind Han's algorithm[H3]. To remove the assumptions we observe that inter-row pointers can be handled almost trivially[H3], that the partition of pointers into x matching sets, when $x = \log^{(i)} n$ for any constant i, can be done in $O(in/p)$ time using the matching partition function $f^{(i+1)}$ described in the previous section, and that the pointers can be sorted to ensure that all pointers in the same column belong to the same matching set.

Theorem 7[H3]: A maximal matching for a linked list can be computed in $O(\frac{n \log i}{p} + \log^{(i)} n + \log i)$ time, where i is an adjustable parameter. □

For the linked list prefix problem Wyllie's pointer jumping technique[W] does not yield an optimal algorithm. When the technique of contraction[MR] is applied one can first find a matching set of size n/c for a constant c. For each pointer in the matching set the head and the tail of the pointer can be combined by the $\bigcirc$ operation. This "pair-off" operation contracts the linked list to a list with only $(1 - 1/2c)n$ nodes. $O(\log n)$ stages of this contraction process reduce the input linked list to a single node. After the linked list is contracted to a single node this contraction process can be reversed to expand the single node to a linked list. This contraction and expansion processes can be used to

evaluate linked list prefix[MR]. A detailed explanation of these processes can be found in [H1][WH].

A key problem in achieving time complexity $O(\log n)$ using optimal $n/\log n$ processors for computing linked list prefix is how to balance load among processors. Because in the i-th stage we have a list of n/c_1^i nodes for a constant $c_1 > 1$, the number of operations needed for all stages form a geometric series. Therefore the sum of the number of operations is $O(n)$, assuming the load (the number of nodes assigned to each processor) at each stage can be balanced among $n/\log n$ processors.

$O(\log^{(2)} n)$ balance operations are needed for that many stages of contraction of the linked list in order to reduce the size of the input list from n to $n/\log n$, at that point Wyllie's pointer jumping algorithm[W] can be invoked to finish the rest of the contraction process. A deterministic global balance step requires $O(\log n)$ time on the EREW model, thus any algorithm using more than constant number of global balance steps will fail to achieve time $O(\log n)$. This barrier of global balancing can be overcome by the following techniques.

2.5. Randomized Load Balancing

Suppose there are k 1's in an array of n cells. These k 1's can be packed into $2a$ cells by randomly assign k 1's to $2k$ cells. Reif[Reif], Miller and Reif[MR] showed that elaborations of this idea can be used to balance load among processors. Such a load balance step can be done in sublogarithmic time if k processors are available. The following theorem is from [MR], proving it requires probabilistic analysis of random strings.

Theorem 8[MR]: There exist a PRAM algorithm using $O(\log^{(2)} n)$ time and $O(n/\log^{(2)} n)$ processors which for at least $1 - 1/n$ of strings with b zeros discards at least $b/2$ zeros. □.

Load balancing algorithm obtained from Theorem 8 is powerful enough to yield an optimal algorithm for linked list prefix with time complexity $O(\log n)$ [MR].

2.6. Global Load Balancing

Although global load balancing can not succeed on the EREW model as we explained above, it works on the CRCW model for achieving optimal linked list prefix algorithm with time complexity $O(\log n)$. The idea originates from Reif's sublogarithmic partial sum algorithm [Reif] with time complexity $O(\log n/\log^{(3)} n)$. Han showed[H1][H2] that Reif's algorithm can be used for globally balancing the load with the same time complexity and that such a global balancing algorithm is sufficient for obtaining a linked list prefix algorithm with time complexity $O(\log n)$ using $n/\log n$ processors.

Parberry[P] and Cole and Vishkin[CV] showed $O(\log n/\log^{(2)} n)$ parallel summation algorithms. Their algorithms will enable faster load balancing.

The above two load balancing techniques are not powerful enough to yield a deterministic EREW linked list prefix algorithm with time complexity $O(\log n)$ using $O(n/\log n)$ processors.

2.7. Dynamic Load Balancing

We can formulate the load balancing problem as follows to understand the key to the solution of the problem. The following formulation is an attempt to offer an intuitive explanation of Anderson and Miller's technique[AM1][AM2].

Initially each processor has a queue of $\log n$ data item[AM1][AM2]. In one step each

processor works on the data item at the top of the queue. The data items with processors working on are called the active items. We will use an oracle which provides a pairing of the active items in constant time. Such an oracle is a resemblance and conceptual simplification of the matching algorithms we have discussed above. When two data items a, b are to be paired off as indicated by the oracle, the new data item $c = a \bigcirc b$ can be stored at either the cell where a was stored or the cell where b was stored.

Anderson and Miller's technique points out that c should be stored in the queue with less items, *i.e.*, c should be stored in the cell occupied by a if the queue where a was stored is shorter than the queue where b was stored, otherwise c should be stored at the cell occupied by b. By assigning weight 2^i to item at the i-th position from the end of the queue, one can conclude that in one step the total weight of the data items is reduced by 3/4. Thus after $O(\log n)$ steps the total weight will be reduced to a constant which implies that the linked list has been contracted into a single node.

The above scheme needs to be modified in order to replace the assumed oracle by deterministic matching partition algorithms to obtain an $O(\log n)$ time and $O(n/\log n)$ processor EREW PRAM algorithm for linked list prefix[AM1][AM2].

We note that the crucial information used in Anderson and Miller's technique is the information provided by the pointers which dictates as to where the nodes resulting from pairing-off should be stored. The load balancing could be difficult if the information provided by the pointers is not available. For example, we may assume that the oracle, when invoked, will randomly eliminate half of the active nodes. Such an oracle does not provide the information implied by the pointers. Consequently, Anderson and Miller's technique can not be used for balancing load for such an oracle to achieve time $O(\log n)$ using $O(n/\log n)$ processors. In fact any scheme which balances for the modified oracle must be stronger than Anderson and Miller's technique[AM1][AM2]

From the view of parallel algorithms, computing a maximal independent set of nodes for a linked list is equivalent to computing a maximal matching set of pointers for the list. Treat the problem as computing a maximal matching set has several advantages: it has the intuition of bisecting pointers for the matching partition function $f^{(2)}$ and it has the advantage of discriminating between inter-row and intra-row pointers, and in the case of Anderson and Miller's technique of dynamic load balancing pointers provide vital information as to where load should be moved. These advantages are important in helping us understand the insights behind these algorithms.

2.8. Further Development

Recent results reveals further development of linked list prefix algorithms on weak models. Kruskal *et al.* [KMR], Anderson and Miller[AM1], Han[H4] designed parallel algorithms for linked list which uses $o(n)$ shared memory cells. In particular, it is known[H4] that there exists a deterministic PRAM algorithm for computing linked list prefix with time complexity $O(n/p + \log n)$ using p shared memory cells. Ryu and JáJá [RJ]] developed an optimal linked list prefix algorithm on the hypercube parallel computer.

3. Applications to Tree Problems

There are great many algorithms designed by applying linked list algorithms. In this section we apply linked list algorithms to some tree problems.

3.1. Maximal Independent Set

We first consider the problem of computing a maximal independent set on a rooted

tree. Computing a maximal independent set for a rooted tree is equivalent to computing a maximal matching set for the corresponding hypertree $(V, \mathcal{E})$, where V is the set of nodes and $\mathcal{E}$ is a set of pairs (v, W) called pointers, $v \in V$, $W \subseteq V$. Equivalent in the sense that they are reducible to each other on a PRAM in constant time. The problem of computing a maximal matching set for a rooted tree is also intimately related to the problem of computing a maximal independent set.

Computing a maximal independent set for a rooted tree has been investigated by Goldberg *et al.*[GPS] and by Jung and Mehlhorn[JM]. Algorithms given in [GPS] are not optimal. Their technique[GPS] enumerates the independent sets, thus at one step only processors assigned to one independent set are working while other processors are idling. Jung and Mehlhorn gave an optimal algorithm using up to $O(n/\log n)$ processors. We show how to achieve the curve $O(\frac{n \log i}{p} + \log^{(i)} n + \log i)$ for computing a maximal independent set for a rooted tree.

Each node v in the tree can be labeled with number $l(v) = f^{(i)}(v, p(v), p(p(v)), ...)$, where $p(v)$ is the parent of v. This is essentially the same as we did for a linked list. These labels are called l labels. Each node can also be labeled with number $r(v) = row(v)$, where $row(v)$ is the row where node v is stored when the tree is viewed as being stored in a two dimensional array. After the nodes in the tree are labeled turn points[H1][H2][H4] can be identified. For two pointers $< v_1, v_2 >$, $< v_2, v_3 >$, v_2 is called a turn point for v_1 if either of the following is true.

(1). $l(v_1) > l(v_2)$ and $l(v_2) < l(v_3)$.

(2). $r(v_1) > r(v_2)$ and $r(v_2) < r(v_3)$.

(3). $r(v_1) \neq r(v_2)$ and $r(v_2) = r(v_3)$.

In order to obtain a maximal independent set we execute the following steps for an input rooted tree.

(1). Label nodes v with $l(v) = f^{(i)}(v, p(v), p(p(v)), ...)$. They are called the l labels. l is bounded by $m = \log^{(i-1)} n(1 + o(1))$.

(2). Delete v if $p(v)$ is a turn point for v by its l label.

(3). Relabel v. If $l(p(v)) > l(v)$ then $l(v) = 2m - l(v)$. After relabeling the l labels for the nodes in a tree are strictly increasing from root to leaves.

(4). View the tree as being stored in a two dimensional array with x rows and $y = n/x$ columns, where x is bounded by $2m$. Sort the nodes by their l labels such that all nodes in one column are labeled by the same number. Use extra y columns if needed.

(5). Label nodes v with $r(v) = row(v)$.

(6). Delete v if $p(v)$ is a turn point for v by its r label.

(7). Delete root of the input tree.

What we intend to do is to create a forest by deleting a set of vertices such that the resulting forest has the following property. A root-to-leaf path $root = a_0, a_1, ..., a_t = leaf$ in a tree is stored in the form that the subpath $a_0, ..., a_i$ is stored ascending the rows, *i.e.*, $row(a_k) < row(a_{k+1})$, subpath $a_i, ..., a_j$ is descending the rows, *i.e.*, $row(a_k) > row(a_{k+1})$, and subpath $a_j, ..., a_t$ is stored in the same row. We let processors first walk up the rows, then walk down the rows. We then use processor scheduling (Section 2.4) to handle the subpath stored in the same row. During this process we may have to added some of the deleted leaves back to the maximal independent set.

Theorem 9: A maximal independent set of a rooted tree can be computed in $O(\frac{n \log i}{p} +$

$\log^{(i)} n + \log i)$ time on a PRAM. □.

Pick i to be an arbitrarily large constant, our algorithm is optimal using up to $O(n/\log^{(i)} n)$ processors.

3.2. Optimal Communication Bandwidth

We now outline how to solve certain tree problems on a PRAM in time $O(n/p+\log n)$ using p shared memory cells. The number of shared memory cells used in a PRAM algorithm is regarded as the communication bandwidth of the algorithm. We show the design of parallel algorithms for some tree problems using optimal communication bandwidth.

We assume that the input tree is stored in a $n/p \times p$ array. Column k of the array is in the local memory of processor k.

It is known[KB][KRS][TV] that many tree problems can be reduced to the linked list prefix problem. We review these reduction techniques[KB][KRS][TV] and show how the reductions are carried on a PRAM with p shared memory cells without losing efficiency (after ignoring multiplicative constant). For these problems the input trees are assumed to be double linked between parents and children.

(1). *Number of descendants in a tree.*

Problem: Label the nodes of a tree with the number of their descendants.

Solution: Convert the tree to a linked list(Fig. 5), label each link in the list with 1. Solve the linked list prefix sum problem. Let $< v, w >$ be the link connecting node v to node w, $n(< v, w >)$ be the result of the prefix sum on link $< v, w >$, and $v_l(v_r)$ be the leftmost(rightmost) son of v, then $\frac{n(<v_r, v>)-n(<v, v_l>)+1}{2}$ is the number of descendants of node v. Linked list prefix sum can be computed in $O(n/p + \log n)$ time [H4]. For the internal nodes, v_l and v_r can be computed in $O(n/p+\log n)$ time by using a straightforward array prefix algorithm.

Time Complexity: $O(n/p + \log n)$ on PRAM using p shared memory cells.

(2). *Preorder traversal labeling of a tree.*

Problem: Label the nodes of a tree with the number of preorder traversal labeling.

Solution: For each leaf v, compute $right(v)$. $right(v)$ is defined as follows. If v is the rightmost leave of the tree, then $right(v) = nil$. If v has an immediate right sibling w then $right(v) = w$, else $right(v) = right(parent(v))$. The definition of $right(v)$ is shown in Fig. 6.

$right(v)$ can be computed using linked list prefix algorithm[H4]. Therefore, we obtain $right(v)$ for all leaves in time $O(n/p + \log n)$.

Now we construct $next(v)$. For each internal node v, $next(v)$ is the the leftmost child of v. For a leaf v, $next(v)$ is $right(v)$. These pointers of $next(v)$ form a linked list. Label every node with 1 and then do a linked list prefix sum[H4]. The resulting numbering is preorder traversal numbering.

Time Complexity: $O(n/p + \log n)$ on PRAM using p shared cells.

Inorder traversal labeling and postorder traversal labeling and many other tree problems can be solved using the same approach.

Certain tree problems are solved more naturally by the technique of tree contraction[MR]. In the rest of this section we show how to adapt tree contraction techniques[MR] to PRAM using only p shared memory cells.

A typical problem in this category is the parallel tree expression evaluation prob-

lem[MR]. An expression tree is a tree with leaves labeled with operands and each internal node labeled with constant number of operators and operands. Most expressions involve only unary and binary operators. Corresponding expression trees are binary trees with chains in the tree representing a chain of unary operators.

When a binary operator is associative, we could extend it to be an n-ary operator. Nodes in an expression trees correspond to such operators may have several children.

Parallel tree contraction technique also applies to tree problems with input trees having no links from parents to children. Such input trees are difficult to be processed directly by a linked list prefix algorithm due to the difficulty of constructing lists. To design deterministic algorithms for certain tree problems with such input trees we can apply tree contraction techniques[MR].

Our tree contraction algorithm has two stages. The first stage contracts the input tree from size n to size p. The second stage contracts the size p tree to its root. Notice that in the second stage the number of tree nodes is no more than the number of processors.

We can use a deterministic PRAM tree algorithm for the second stage of our tree contraction algorithm. The PRAM tree contraction algorithm of Miller and Reif[MR] can be used here which takes $O(\log p) \leq O(\log n)$ time since the tree has p nodes and the machine has p processors. In the following we only consider the first stage of our tree contraction algorithm.

In order to contract trees properly a node in a tree needs the information when it will become a chain node. Such information is readily obtained if the node has only constant number of children, for its children can inform the node in constant time. When there is a node the number of its children is not a constant, the following strategies can be used.

(a). *The input tree is double linked.*

After each step of RAKE and COMPRESS[MR], perform a packing operation on the number of children for each parent node. The result of this operation tells, for each parent node, how many children have not been contracted to a single node yet. Thus it takes $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n})$ time to inform the parents. Since the RAKE and COMPRESS operation will reduce the number of nodes in a tree by a constant fraction, $O(\log(n/p))$ executions of RAKE and COMPRESS suffice for the first stage of our algorithm. The total time needed for informing the parents in the first stage of our algorithm is $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n} \log(n/p)) \leq$ $O(n/p + \log n)$ because the number of nodes in the tree being contracted form a geometric series.

(b). *The input tree has no links from parents to children.*

In this case we route the children to their parents and by using concurrent write the parents are informed whether there are children left. Here the routing operation is not a permutation, therefore our permutation algorithm[H4] can not be directly used. Instead we first sort the nodes by the first $\lceil \log(n/p) \rceil$ bits of the addresses in the parent pointer. In doing this we have routed the children into blocks. Here, a block is a continuous address locations containing children whose parents were in the same row before sorting. A block could occupy several rows in the memory and one row could contain several blocks as shown in Fig. 7. If we impose the condition that one row contain one block only, then the number of rows needed is bounded by $\sum_i^{n/p} \lceil \frac{B_i}{p} \rceil \leq 2n/p$, where B_i is the number of elements in block i. After sorting we then use concurrent write to inform the parents. The concurrent write is performed one row at a time. Since elements in a row are in the same block, each child sends information via concurrent write to its parent.

Because sorting takes only $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n})$ time. We obtain the same time complexity as that in (a).

The first stage of our tree contraction algorithm has two steps. In the first step the input tree of size n is contracted to a tree of size $n/\log^{(2)} n$. In the second step the tree of size $n/\log^{(2)} n$ is contracted to a tree of size p. We present the details of these two steps below.

Step 1:

Step 1 is a loop with $\log^{(3)} n/\log c$ iterations. Each iteration reduces the tree from size s to s/c, where $c > 1$ is a constant.

We use schemes outlined above to inform each tree node whether it has children left. A node with no child is a leaf. These leaves then delete themselves, therefore realizing the RAKE operation. These operations take $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n})$ time for a tree of size n.

A node with one child left is a chain node. After identifying chains in the tree, we compute $f^{(4)}$ for chain nodes and label each chain node v with $l(v) = f^{(4)}(v, ...)$. The computation yields a partition of $c_1 \log^{(3)} n$ matching sets (labeled from 0 to $c_1 \log^{(3)} n - 1$), where c_1 is a constant. We then pack these chain nodes and sort them by 3-tuples $< l(v), l(child(v)), l(parent(v)) >$, where $l(child(v)) = -1$ if v has no child as a chain node and $l(parent(v)) = c_1 \log^{(3)} n$ if v has no parent as a chain node (Fig. 8). After sorting we then let processors walk up these chains. To walk up from nodes with $l(v) = i$ to their parent nodes with $l(v) = j$, $j > i$, we pack source nodes with 3-tuples $< i, s, j >$, $s = -1, 0, 1, ..., i - 1$ together and destination nodes with 3-tuple $< j, i, t >$, $t = j + 1, j+2, ...$ together. A permutation then routes source nodes to their parents (destination nodes). Let n_{ij} be the number of nodes with 3-tuple $< i, s, j >$ or $< j, i, t >$. The time needed for walking up from $l(v) = i$ to $l(v) = j$ takes $O(\frac{n_{ij}}{p} + \frac{\log n}{\log^{(2)} n})$ time. Sum for all possible values of ij the time becomes $O(\sum_{ij} \frac{n_{ij}}{p} + \frac{\log n (\log^{(3)} n)^2}{\log^{(2)} n})$. Because a node with 3-tuple $< i, j, k >$ is counted at most twice, once in n_{ik}, once in n_{ji}, we know $\sum_{ij} n_{ij} \leq 2n$. So the timing for one iteration of the loop of step 1 is $O(\frac{n}{p} + \frac{\log n (\log^{(3)} n)^2}{\log^{(2)} n})$, if the size of the tree is n.

Since each iteration of the loop of step 1 reduces the size of the tree to a constant fraction, therefore the time complexity of step 1 is $O(\sum_{i=1}^{O(\log^{(3)} n)} (\frac{n}{c^i p} + \frac{\log n (\log^{(3)} n)^2}{\log^{(2)} n})) \leq O(n/p + \log n)$.

Step 2:

Step 2 is a loop with $\frac{\log(n/p)}{\log c}$ iterations. Each iteration reduces the tree from size s to s/c, where $c > 1$ is a constant.

We use the same scheme to inform each tree node whether it has children left. Then RAKE operation is performed in the same way as that in step 1.

Maximal matchings are found for the chain nodes. This requires $O(\frac{sG(s)}{p} + \frac{\log s}{\log^{(2)} s})$ for chains of size s. However, since step 1 has reduced the tree to size $n/\log^{(2)} n$, this can not lead our algorithm to a nonoptimal one. A permutation is then needed to compress the chains and a pack operation packs the tree so that less time would needed for the later iterations.

The time complexity of step 2 is $O(\frac{sG(s)}{p} + \frac{\log s}{\log^{(2)} s} \log(n/p))$. Notice that, since the

size of the contracted trees form a geometric series, the term $sG(s)/p$ remains unchanged. Because $s \leq n/\log^{(2)} n$, the time complexity is $O(n/p + \log n)$.

After these two steps the size of the tree has been reduced to p. This accomplishes the first stage of our algorithm.

Theorem 10: Parallel tree contraction can be done in time $O(n/p + \log n)$ on PRAM using p shared memory cells. □

The essence of the parallel tree algorithms presented in this paper is the technique of list and tree contractions. Depending on the operations defined on the elements associated with list or tree nodes, many different list and tree functions can be computed efficiently on PRAM using p shared cells. It is not difficult to construct a function which requires only $o(p)$ shared cells to achieve time $O(n/p+\log n)$. We show that $\Omega(p)$ shared cells are required for our tree algorithms to achieve time $O(n/p+\log n)$ by demonstrating that there exists a tree function which requires that many shared cells. Our proof requires that each shared cell has only $O(\log n)$ bits.

Theorem 11: There is a tree function which requires $\Omega(p)$ shared cells to be computed in time $O(n/p + \log n)$.

Proof: We use the well known technique of crossing sequence[U] to prove the lower bound. In t steps the sequence appeared in the p shared cells has $O(tp \log n)$ bits.

Suppose we are to compute the number of descendants for all tree nodes. The input tree is of the following form. The root has $n/4$ children and they are in the local memory of $n/4p$ processors. These $n/4$ tree nodes are said to be in group A. Each of the $n/4$ nodes has exactly one child. The rest $n/2 - 1$ tree nodes are grandchildren of nodes in group A and they are leaves. The nodes in group A compute the number of descendants of theirs by receiving information from shared cells. However, there are $\binom{n/2-1}{n/4}$ ways of attaching these leaves to their parents. Thus $\Theta(n \log n)$ bits are needed to distinguish between different ways of attachment. If $t = o(n/p)$ then the crossing sequence has $o(n \log n)$ bits which is not enough to distinguish between different ways of attachment. □

4. Conclusions

We have presented basic techniques for the design of linked list prefix algorithms and applications of these algorithms. Due to page limit we are unable to explore many important applications and ramifications. We believe that both the techniques and the algorithms for linked list are fundamental and they will be indispensable building blocks for the design of fast and efficient parallel algorithms for a large variety of problems.

References

[AM1]. R. J. Anderson, G. L. Miller. Optimal parallel algorithms for the list ranking problem, USC-Tech. Rept. 1986.

[AM2]. R. J. Anderson, G. L. Miller. Deterministic parallel list ranking. LNCS 319 (J. H. Reif ed.), 81-90.

[BH]. R. A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parallel models of computation. Proc. 14th ACM Symposium on Theory of Computing, 206-219(1986).

[CDHR]. B. S. Chlebus, K. Diks, T. Hagerup, T. Radzik. New simulations between CRCW PRAMs. LNCS 380, 95-104.

[CV]. R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems, Proc. 27th Symp. on Foundations of Comput.

Sci., IEEE, 478-491(1986).

[GPS]. 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).

[H1]. Y. Han. Designing fast and efficient parallel algorithms. Ph.D. Thesis, Duke University, Durham, NC 27706, 1987.

[H2]. Y. Han. Parallel algorithms for computing linked list prefix. J. of Parallel and Distributed Computing 6, 537-557(1989).

[H3]. Y. Han. Matching partition a linked list and its optimization. Proc. ACM Symp. on Parallel Algorithms and Architectures, Sante Fe, New Mexico, 246-253(1989).

[H4]. Y. Han. An optimal linked list prefix algorithm on a local memory computer. Proc. 1989 ACM Computer Science Conf., Lousville, Kentucky, 278-286(1989).

[H5]. Y. Han. On the chromatic number of Shuffle graphs. TR. No. 140-89, Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA.

[JM]. H. Jung, K. Mehlhorn. Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees, Information Processing Letters, Vol. 27, No. 5, 227-236(1988).

[KB]. N. C. Kalra, P. C. P. Bhatt. Parallel algorithms for tree traversals, Parallel Computing, Vol. 2, No. 2, June 1985.

[KMR]. C. P. Kruskal, T. Madej, L. Rudolph. Parallel prefix on fully connected direct connection machine. Proc. 1986 Int. Conf. on Parallel Processing, 278-284.

[KRS]. C. P. Kruskal, L. Rudolph and M. Snir. Efficient parallel algorithms for graph problems. Proc. 1986 International Conf. on Parallel Processing, 869-876.

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

[MR]. G. L. Miller, J. H. Reif. Parallel tree contraction and its application. Proc. 1985 IEEE Foundations of Computer Science, 478-489.

[P]. I. Parberry. On the time required to sum n semigroup elements on a parallel machine with simultaneous writes. LNCS, 227, 296-304.

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

[RJ]. K. W. Ryu, J. JáJá. List ranking on the hypercube. Proc. 1989 Int. Conf. on Parallel Processing. St. Charles, Illinois, (Aug. 1989).

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

[Sp]. E. Sperner. Ein satz über untermenger endlichen menge, Math. Z. 27(1928), 544-548.

[TV]. R. E. Tarjan, U. Vishkin. Finding biconnected components and computing tree functions in logarithmic time, 25th Symp. on Foundations of Computer Sci., IEEE, 12-20.

[U]. J. D. Ullman. Computational aspects of VLSI, Computer Science Press, 1984.

[WH]. R. A. Wagner, Y. Han. Parallel algorithms for bucket sorting the data dependent prefix problem. Proc. 1986 Int. Conf. on Parallel Processing, 924-930.

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

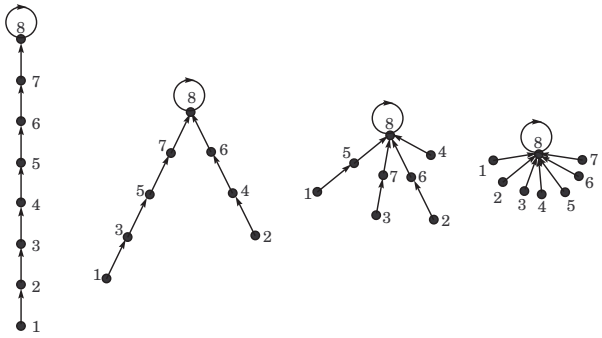

Fig. 1. The effect of pointer jumping

	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. 2. A linked list

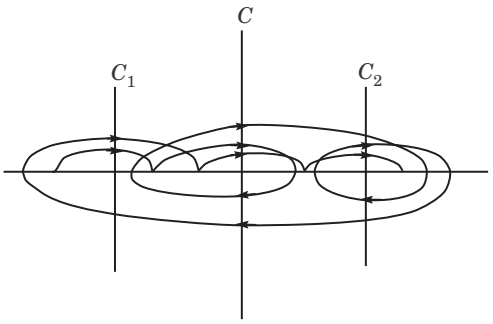

Fig. 3. The intuitive observation of bisecting

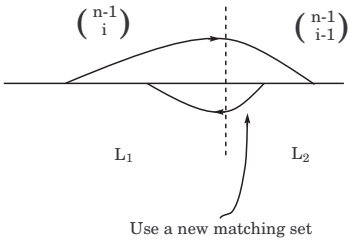

Fig. 4.

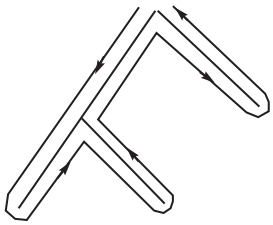

Fig. 5. Unfold a tree

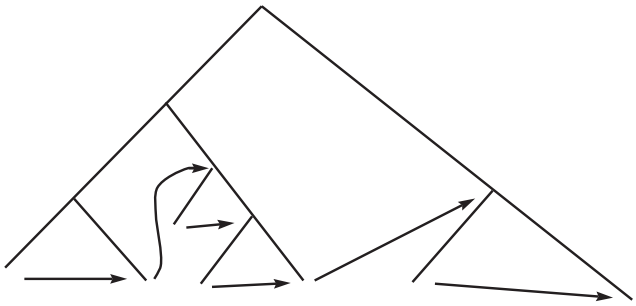

Fig. 6. Pointers of $right(v)$

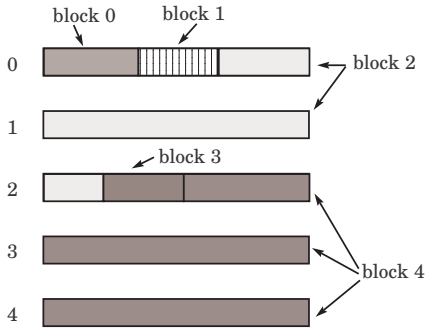

Fig. 7.

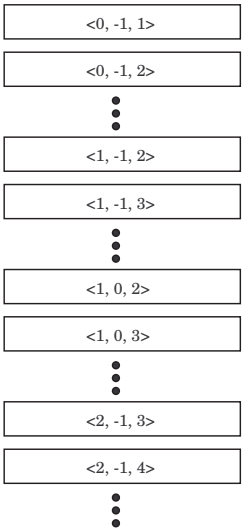

Fig. 8.