An Optimal Linked List Prefix Algorithm on a Local Memory Computer

Yijie Han

Abstract — We present a deterministic parallel algorithm for the linked list prefix problem. It computes linked list prefix for an input list of n elements in time $O(n/p + \log n)$ on a local memory PRAM model using p processors and p shared memory cells.

Index terms — Graph algorithms, linked list, local memory computer, memory sharing, optimal algorithms, parallel algorithms, prefix.

1. Introduction

Given a linked list $x_1, x_2, ..., x_n$ with x_i following x_{i-1} in the list and an associative operation $\bigcirc$, the linked list prefix problem is to compute all prefixes $\bigcirc_{i=1}^{k} x_i$, $k = 1, 2, ..., n$.

In this paper, we study the linked list prefix problem on a local memory PRAM model. A similar model called the Direct Connection Machine (DCM) model was also defined by Kruskal *et al.* [9].

A local memory PRAM with p processors is a PRAM [3] with only p shared memory cells. In addition, each processor has its local memory module which can only be accessed by the processor. Therefore, the local memory PRAM is a weaker model than the PRAM model. The prohibition on the simultaneous access to the same memory module can generate a "hot spot"[13] — a memory module to be accessed by several processors simultaneously. This hot spot problem has not been taken into account on the PRAM modal.

According to the conflict resolving rules for the concurrent access to the p shared cells, local memory PRAM's can be classified into CRCW, CREW, and EREW local memory PRAM's [16].

Our linked list prefix algorithm is designed on the CRCW local memory PRAM.

We shall adopt a convenient treatment of memory modules on a local memory PRAM. We view them as a two-dimensional array [1], [8], [9]. Each column of the array is a memory module. A memory cell is addressed by an ordered pair $< i, j >$ with j indicating the memory module and i indicating the address of the cell within the memory module. A processor can access any memory cell in any memory module. Simultaneous access to the same memory module is valid only when processors are reading or writing the same cell in the memory module being accessed. In the case of simultaneous write, we adopt the rule which says an arbitrary processor succeeds in writing. This treatment is valid when we ignore the multiplicative constant in the time complexity of our algorithm.

Sorting on a local memory PRAM is to sort data items into row-major ordering, while packing is meant to rearrange data items such that "null" items will follow "nonnull" items.

The linked list prefix problem is a fundamental problem which has been studied extensively[1], [2], [7]-[10], [15], [17]. Many tree problems can be reduced to the linked list prefix problem and many graph and other problems require fast and efficient algorithms for computing linked list prefix. Among these problems are packing, processor allocation, expression evaluation, network routing and code decomposition. Known PRAM algorithms [2], [7], [8] achieve time complexity $O(n/p + \log n)$[1]. Anderson and Miller also obtained a randomized local memory PRAM algorithm with time complexity $O(n/p + \log n)$ [1]. On the other hand, Kruskal *et al.* [9] and Han [8] showed optimal

Manuscript received March 7, 1988; revised July 10, 1990. A preliminary version of this paper was presented at the 1989 ACM Computer Science Conference, Lousville, KY, February 1989.

The author is with the Department of Computer Science, University of Kentucky, Lexington, KY 40506.

[1]We use n to denote the size of a problem and p to denote the number of processors used in an algorithm. Note also that $O(n/p + \log n) = O(n/p + \log p)$.

deterministic local memory PRAM algorithms. The algorithm of Kruskal *et al.* [9] is optimal when $n \geq p^3$.

Our previous algorithm [8] was obtained through the resolution of vector balancing. It has time complexity $O(n/p + p \log p)$ and is optimal when $n \geq p^2 \log p$. In this paper, we present a local memory PRAM linked list prefix algorithm which computes linked list prefix for a list of n elements in time $O(n/p + \log n)$. The current algorithm is obtained by applying the matching partition function [7], [8] to obtain partitioned matching sets and maximal matching set for the linked list.

The rest of the paper is organized as follows. In section 2 known techniques for computing the linked list prefix on the PRAM model are reviewed. In section 3 we present algorithms for integer sorting and dynamic data permutation on the local memory PRAM model. These algorithms will be used in our linked list prefix algorithm. The linked list prefix algorithm is presented in section 4 and conclusions are drawn in section 5.

2. Preliminary

A known approach for computing linked list prefix in parallel is linked list contraction[12]. This approach relies on an efficient parallel scheme for obtaining a matching set for the linked list. In graph terminology, a matching set is a set of edges such that no edges in the matching set are incident on the same vertex. A matching is maximal if adding one more edge to the matching makes it a nonmatching set. When a matching set is found for the linked list, two elements of every pointer in the matching set can be combined (paired-off [10]) by the associative operation $\bigcirc$. This is illustrated in Fig. 1. After the combining process the linked list is contracted to a shorter list. The contraction process is repeated until the linked list is shrunk to a single node. This linked list contraction process can be utilized to evaluate linked list prefix[7], [10], [12], [17]. The details are explained in [7] and [17].

In order to obtain an efficient parallel algorithm, we need an efficient scheme to obtain a maximal matching set for the linked list. The following intuitive observation[7] gives us a function which partitions the pointers of a linked list into $2\lceil \log n \rceil$ matching sets.

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. 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 subarrays by line c. We can draw bisecting lines c_1, c_2 for the two subarrays. 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 $g(< a, b >) = \max\{i|$ the i-th bit of a XOR b is $1\}$, where XOR is the bitwise 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.

Lemma 1[7]: Function f is a matching partition function which partitions the pointers of a linked list into $2\lceil \log n \rceil$ matching sets. □

In [7] and [17] we also used the least significant bit [2] instead of the most significant bit for defining function f. In doing so, we gain the advantage of computing function f at the expense of losing intuition. Interested readers are referred to [7] and [17] for a scheme of computing function f.

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[7]. Define $f^{(2)}(x, y) = f(x, y)$, $f^{(k)}(x_1, x_2, \cdots, x_k) = f(f^{(k-1)}(x_1, x_2, \cdots, x_{k-1}), f^{(k-1)}(x_2, x_3, \cdots, x_k))$. Function $f^{(k)}(x, suc(x), suc(suc(x)), \cdots)$ represents repeated application of function f to the linked list.

Lemma 2[7]: Function $f^{(k)}$ is a matching partition function which partitions the pointers of a linked list into $c \log^{(k-1)} n$ matching sets, where c is a constant. □

When k reaches $G(n)$, $f^{(k)}$ becomes a constant. Consequently, we obtained a maximal matching. The PRAM algorithm [7] 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

/* $G(n) = \min\{i | \log^{(i)} n < 2\}$ */

$label[v] := f(< label[v], label[suc(v)] >)$

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 becuase at least one of any three consecutive pointers of the linked list is in the matching. The time complexity of Algorithm Match1 is $O(G(n))$ with n processors.

Lemma 3[7]: A maximal matching can be computed in $O(nG(n)/p + G(n))$ time on a PRAM model. □

After $f^{(k)}$ is computed for a linked list, each node a is assigned a number from $\{0, 1, ..., c \log^{(k-1)} n\}$ to denote to which matching set pointer $< a, suc(a) >$ belongs. Because $f^{(k)}$ is a matching partition function, two neighboring nodes in the list cannot be assigned the same number. The numbers associated with the nodes of the linked list consists of ascending and descending subsequences as shown in Fig. 4. The linked list can be cut at those nodes which have been assigned local minimum values. Each cut list has between 2 and $2c \log^{(k-1)} n$ elements.

We can compute function $f^{(k)}$ on a local memory PRAM by using a dynamic data permutation algorithm. A dynamic permutation is a permutation effected at run time, while a static permutation is a known permutation before program execution. Assume that $f^{(i-1)}$ is computed for every node of the linked list. By invoking a dynamic permutation, value $f^{(i-1)}(suc(a), suc(suc(a)), \cdots)$ is routed to node a and $f^{(i)}(a, suc(a), \cdots) = f(f^{(i-1)}(a, suc(a), \cdots), f^{(i-1)}(suc(a), suc(suc(a)), \cdots))$ can be evaluated.

A static permutation can be done in $O(n/p)$ time on a local memory PRAM because the permutation pattern can be decided before program execution. A dynamic permutation may require more

[2]Lemmas 1-3 are from [7]. The scheme of using the least significant 1-bit to obtaining matching partition and the results in Lemmas 2 and 3 were independently discovered in [4]. The intuitive observation presented here was observed exclusively by the author[7].

time. Let $perm(n,p)$ be the time complexity for performing a dynamic permutation of n items on a local memory PRAM with p processors. We have:

Lemma 4: Function $f^{(k)}$ can be evaluated on a local memory PRAM in time $O(perm(n,p)\cdot k)$. A maximal matching set can be computed on a local memory PRAM in $O(perm(n,p)\cdot G(n))$ time. □

3. Integer Sorting and Dynamic Permutation

We first consider sorting n integers in the range $\{1,2,...,n/p\}$ on a local memory PRAM with p processors. We use a principle revealed by Leighton's column sort[11]. The principle says that n elements are sorted by sorting $n^{1/3}$ sequences of $n^{2/3}$ elements a constant number of passes and performing a static permutation after each pass. Applying this principle recursively it is not difficult to see that n elements are sorted by sorting $n^{1/2}$ sequences of $n^{1/2}$ elements a constant number of passes and executing a static permutation after each pass. When $n/p \geq p$, time $O(n/p)$ can be easily achieved for our sorting problem[8]. Therefore we only consider the case when $n/p < p$. In this case, we only need to sort each row a constant number of passes and execute a static permutation after each pass of sorting.

In each pass of sorting p^2/n processors are assigned to each row of the memory modules. Each processor sets up an array of n/p entries, the ith entry is used as a counter counting how many i's. These entries are set to 0's at the beginning. There are three stages. In the first stage, one processor takes care of n/p data items in the row it is assigned to, i.e., it counts how many i's in those n/p data items and records the count in the i-th entry of the table. In the second stage, processors working in one row combine their tables in a binary tree fashion. The resulting table at the root of the binary tree tells how many i's within the row. In the third stage, the information at the root of the binary tree is passed down along the tree. As a result every data item will get the information how many data items in its row are ranked lower than itself. This effectively sorts one row.

Several issues have to be dealt with. Careful processor scheduling can achieve exclusive memory module access. In the second stage, tables should be split up among processors so that processor utilization can be maximized. At the bottom level of the tree, each processor has a table of size n/p, at the second level each processor only has a half of a table, and so on. When these issues have been taken care of, the time complexity of the sorting algorithm is $O(n/p+\log n)$. The sorting scheme is illustrated in Fig. 5.

To sort integers in the range $\{1,2,...,m\}$, the idea of radix sorting can be used. Consequently the time complexity becomes $O((\frac{n}{p}+\log n)\frac{\log m}{\log(n/p)})$.

Because of the simultaneous read/write feature on the local memory PRAM model, we are able to do better on sorting. Simultaneous read/write corresponds to circuits with fan-in greater than constant. This will give us the chance of outperforming the additive factor of $\log n$. Reif[14] first designed a sublogarithmic PRAM algorithm for computing partial sum. His algorithm was used by the author to sort integers on PRAM in sublogarithmic time [7]. Cole and Vishkin [5] obtained an improved partial sum algorithm which has time complexity $O(\frac{n}{p}+\frac{\log n}{\log^{(2)} n})$. We now use this partial sum algorithm to sort integers. In the tree we built for integer sorting, the bottom $\log(n/p)$ levels will still be binary. On higher levels we use sublogarithmic partial sum algorithm. This will enable us to sort integers in the range $\{1,2,...,n/p\}$ in time $O(\frac{n}{p}+\frac{\log n}{\log^{(2)} n})$ on a p processor local memory PRAM. By applying radix sorting, we then extend our algorithm to sort n integers of magnitude $(\log n)^{O(1)}$ in time $O(\frac{n}{p}+\frac{\log n}{\log^{(2)} n})$ on the local memory PRAM.

Now consider the dynamic permutation problem. Without loss of generality we assume p is a power of 2. Consider the problem of performing dynamic permutation for pm data items on a p-processor local memory PRAM where each memory module has m data items stored in cells addressed from 0 to $m-1$. First we sort these data items by the most significant $\lceil \log m \rceil$ bits. The sorting can be done in time $O(\frac{n}{p}+\frac{\log n}{\log^{(2)} n})$ because $m = n/p$. After sorting each data item has been

routed to the destination row in the memory modules. We then perform a permutation for each row to route data items to their final destinations. We have:

Theorem 1: The time complexity for dynamic permutation is $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n})$ on a p-processor local memory PRAM. □

Corollary: A linked list of n elements can be converted to a doubly-linked list in time $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n})$ on a p-processor local memory PRAM. □

4. An Algorithm for Computing Linked List Prefix

By Lemma 3, we immediately obtain a local memory PRAM algorithm for contracting a linked list. L_Pair1 contracts a linked list of n elements to a list of $2n/3$ elements in $O(\frac{nG(n)}{p} + \frac{\log n}{\log^{(2)} n})$ time.

L_Pair1:

Step 1: Compute a maximal matching using the scheme of Algorithm Match1.

Step 2: For each pointer in M combine the two elements of the pointer using operator $\bigcirc$.

Step 3: Pack the contracted linked list to the top of the memory modules.

Step 1 requires $O(\frac{nG(n)}{p} + \frac{\log n}{\log^{(2)} n})$ time. The permutation specified by the linked list needs to be executed $G(n)$ times. The first time it is executed it is a dynamic permutation requiring $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n})$ time. After the first execution it effectively becomes a static permutation and requires only $O(n/p)$ time for each execution. Step 2 is accomplished by permuting data elements to route together two elements of each pointer in M and then performing $\bigcirc$ operation. Packing can be done by sorting on integers with value of 0's and 1's (assigning 0's to null elements and 1's to nonnull elements). The time complexity of L_Pair1 is $O(\frac{nG(n)}{p} + \frac{\log n}{\log^{(2)} n})$.

L_Pair1 is not an optimal algorithm because it uses more than $O(n)$ operations. We design L_Pair2 which is optimal when $n > \frac{p \log n \log^{(k)} n}{\log^{(2)} n}$, where k is an arbitrarily large constant. The basic idea of L_Pair2 is to obtain many cut lists of length no longer than $\log^{(k)} n$. Then allocate processors to walk down these lists sequentially. In one stage, processors walk down one link. In $\log^{(k)} n$ stages, every cut list is exhausted. As processors walk down the links, $\bigcirc$ operation is performed. This effectively contracts the linked list. We can use Lemma 2 to obtain cut lists of length $\log^{(k)} n$ and it only takes $O(k \cdot perm(n,p))$ time. Since the number of pointers being cut is at most $\frac{n}{2}$, after processor walking down the cut lists, the contracted list has at most $n/2$ elements.

L_Pair2:

Step 1: Compute $f^{(k+1)}$ for the input linked list, where k is an arbitrarily large constant. Label node a with $l(a) = f^{(k+1)}(a, suc(a), \cdots)$.

Step 2: If $l(pre(a)) > l(a)$ and $l(suc(a)) > l(a)$, a is at local minimum. In this case delete $< a, suc(a) >$. Each cut list has at most $2c \log^{(k)} n$ elements. Now relabel every node. For node a, if $l(pre(a)) > l(a)$ then reassign $l(a)$ with $2c \log^{(k)} n - l(a)$, otherwise no reassignment will be made. After the relabeling the labels of each cut list is an ascending sequence of numbers in $\{0, 1, ..., 2c \log^{(k)} n\}$, where c is a constant.

Step 3: Sort elements by triple $< l(a), l(pre(a)), l(suc(a)) >$, where $l(pre(a))$ is -1 if $pre(a)$ is null, $l(suc(a))$ is $2c \log^{(k)} n + 1$ if $suc(a)$ is null. Sorting is necessary in order to allocate processors to the list properly for the walk-down.

Step 4:

```
for a := 0 to c log^(k) n − 1 step 1 do
    begin
        for b := a + 1 to c log^(k) n do
```

begin
/* Working on pointers from elements with
triple $< a, l, b >$, $l = -1, 0, ..., a-1$, to
elements with triple $< b, a, m >$, $m = b+$
$1, \cdots, 2c \log^{(k)} n + 1$. */

Permute elements with triple $< a, l, b >$ to
elements with triple $< b, a, m >$ according to
the pointers connecting these elements.

Perform $\bigcirc$ operation to contract lists.
end
end

L_Pair2 contracts a linked list of n elements to a list of at most $n/2$ elements. Step 4 accomplishes the task of walking down the cut lists. Since elements are presorted by triples, permutation can be done in $O(\frac{n_{ab}}{p} + \frac{\log n}{\log^{(2)} n})$ time, where n_{ab} is the number of elements with triple $< a, l, b >$ or $< b, a, m >$.

Step 1 of L_Pair2 takes $O(k \cdot perm(n, p))$ time. Step 2 requires a constant number of permutations and therefore takes $O(perm(n, p))$ time. Step 3 takes $O(\frac{n}{p} + \frac{\log n}{\log^{(2)} n})$ time because sorting is performed on integers of magnitude $(\log n)^{O(1)}$. Note that an element with triple $< p, q, r >$ participates at most twice in the iterations of step 4, once in the iteration with $a = q$ and $b = p$, once in the iteration with $a = p$ and $b = r$. Therefore $\sum_{a,b} n_{ab} \leq 2n$. The time complexity of step 4 is $O(\sum_{a,b}(\frac{n_{ab}}{p} + \frac{\log n}{\log^{(2)} n})) = O(\frac{n}{p} + \frac{\log n \cdot (\log^{(k)} n)^2}{\log^{(2)} n})$. Because k can be arbitrarily large, the time complexity of L_Pair2 can be simplified as $O(\frac{n}{p} + \frac{\log n \log^{(k)} n}{\log^{(2)} n})$.

We shall use Wyllie's pointer jumping algorithm[18] (L_Pair3) which runs on an n-processor local memory PRAM and contracts a linked list of n elements to a single node in $O(\log n)$ time.

We now construct a local memory PRAM algorithm which contracts an input linked list to a single node.

L_Contract

Step 1:

for $j := 1$ to $\log^{(3)} n$ step 1 do
begin
Set i to 4 and call L_Pair2( );
Pack the contracted list;
end

Step 2:

for $j := 1$ to $\log \frac{n}{p}$ step 1 do
begin
Call L_Pair1( );
Pack the contracted list;
end

Step 3: Call L_Pair3( );

L_Contract has three steps. Note that the number of processors does not change while the linked list is being contracted. For different values of n/p L_Contract uses different subroutines to achieve maximum efficiency. We now prove the following theorem.

Theorem 2: L_Contract contracts a linked list of n elements to a single node in time $O(n/p + \log n)$

on a local memory PRAM.

Proof: After step 1 the input linked list has been contracted to a list of $n/\log^{(2)} n$ elements. After step 2 the list has been contracted to a list of p elements. Step 3 contracts the list to a single node.

Step 1 takes $O(\sum_{j=1}^{\log^{(3)} n}(\frac{n}{2^j p} + \frac{\log n(\log^{(4)} n)^2}{\log^{(2)} n})) = O(n/p + \log n)$ time. After step 1 the contracted list has $n/\log^{(2)} n$ elements. Step 2 takes $O(\sum_{j=1}^{\log \frac{n}{p}}(\frac{nG(n)}{2^j p \log^{(2)} n} + \frac{\log n}{\log^{(2)} n})) = O(n/p + \frac{\log n}{\log^{(2)} n} \log \frac{n}{p}) \leq O(n/p + \log n)$ time. After step 2 the list has no more than p elements. Step 3 takes $O(\log p) \leq O(\log n)$ time to contract this list to a single node. □

The linked list prefix can be computed by performing $\bigcirc$ operations in the process of contracting and expanding the linked list. This is a known technique [12]. A detailed explanation of this technique for evaluating linked list prefix is given in [7] and [17].

Our algorithm can also be made to run on the EREW local memory PRAM model. The time complexity would become $O(n/p + \log n \log(n/p))$.

5. Conclusions

We have presented a local memory PRAM algorithm for computing the linked list prefix in $O(n/p + \log n)$ time. Can the computation model be further weakened while still maintaining the same time complexity? We make some comments on this question.

It is not difficult to see that a p-processor local memory PRAM with $o(p)$ shared cells can not compute linked list prefix in $O(n/p)$ time. Divide memory modules into two groups A and B. Place x_{2k} in group A and x_{2k-1} in group B, $k = 1, 2, \cdots$. The computation of linked list prefix requires the communication of $n/2$ data items between groups A and B while the interface between two groups has only $o(p)$ cells. Therefore, timing $O(n/p)$ cannot be achieved.

Another way to weaken the local memory PRAM model is to limit the degree of each processor. The degree of a processor is the number of processors it is directly connected to. The degree of a parallel computer is the maximum of the degrees of its processors. A p-processor local memory PRAM with p shared cells corresponds to a parallel machine of p processors with an interconnection network connecting every pair of processors. The degree of each processor in this machine is p. However, reducing the degree of a computer could significantly weaken its communication power[6]. There are n input data items and n output data items, a permutation from the input data items to the output data items is required for computing the linked list prefix problem. The results of Gottlieb and Kruskal[6] show that with a computer of degree k $\Omega(\frac{n \log_k p}{p})$ is a lower bound for static permutation. When the computer has degree $k = 2^{o(\log p)}$ timing $O(n/p)$ cannot be achieved.

Finally, the memory sharing scheme of the p memory cells could possibly be weakened for computing the linked list prefix. We are not clear if it is possible to achieve time $O(n/p + \log n)$ without concurrent access to the p shared cells.

Acknowledgment

I wish to thank anonymous referees for pointing out an error in the original manuscript.

References

[1] R. J. Anderson, G. L. Miller, "Optimal parallel algorithms for the list ranking problem," U.S.C. Tech Rep. 1986.

[2] — , "Deterministic parallel list ranking," in *Lecture Notes in Computer Science 319, VLSI Algorithms and Architectures*, J. Reif, Ed., 3rd Aegean Workshop on Computing, 81-90, June-July, 1988.

[3] R. A. Borodin and J. E. Hopcroft, "Routing, merging and sorting on parallel models of computation," in *Proc. 14th ACM Symp. Theory Comput.*, San Francisco, CA. Apr. 1982, pp. 338-344.

[4] R. Cole and U. Vishkin. "Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms," in *Proc. 18th ACM Symp. on Theory Comput.*, 1986, pp. 206-219.

[5] — , "Approximate and exact parallel scheduling with applications to list, tree and graph problems," in *Proc. 27th Symp. Foundations Comput. Sci.*, IEEE, 1986, pp. 478-491.

[6] A. Gottlieb, C. P. Kruskal, "Complexity results for permuting data and other computations on parallel processors," *J. ACM*, vol. 31, no. 2, pp. 193-209, Apr. 1984.

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

[8] — , "Parallel algorithms for computing linked list prefix," *J. Parallel Distributed Comput.*, vol. 6, pp. 537-557, 1989.

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

[10] C. P. Kruskal, L. Rudolph, M. Snir, "The power of parallel prefix," *IEEE Trans. Comput.*, vol. C-34, no. 10, pp. 965-968, Oct. 1985.

[11] T. Leighton, "Tight bounds on the complexity of parallel sorting," *IEEE Trans. Comput.*, vol. C-34, 344-354, 1985.

[12] G. L. Miller, J. H. Reif, "Parallel tree contraction and its application," in *Proc. 26th Symp. on Foundations Comput. Sci.*, IEEE, 1985, 478-489.

[13] G. F. Pfister and V. A. Norton. "Hot Spot contention and combining in multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-34, no. 10, pp. 934-948, Oct. 1985.

[14] J. H. Reif, "An optimal parallel algorithm for integer sorting," in *Proc. 26th Symp. Foundations Comput. Sci.*, IEEE, 1985, pp. 291-298.

[15] — , "Probabilistic parallel prefix computation," in *Proc. of 1984 Int. Conf. Parallel Processing*, Aug. 1984.

[16] M. Snir, "On parallel searching," *SIAM J. Comput.*, vol. 14, no. 3, pp. 688-708, Aug. 1985.

[17] R. A. Wagner and Y. Han, "Parallel algorithms for bucket sorting and the data dependent prefix problem," in *Proc. 1986 Int. Conf. Parallel Processing*, pp. 924-930.

[18] J. C. Wyllie, "The complexity of parallel computation," TR 79-387, Dep. Comput. Sci., Cornell University, Ithaca, NY, 1979.

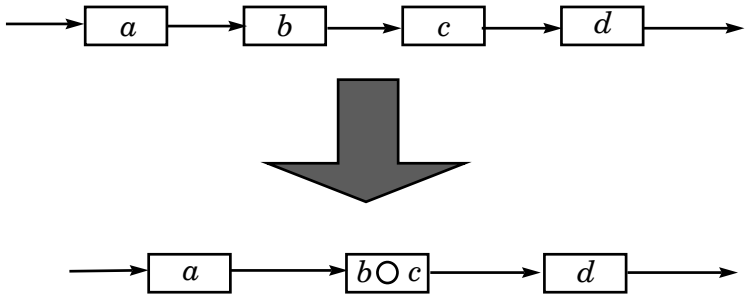

Fig. 1. Paioff.

	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.

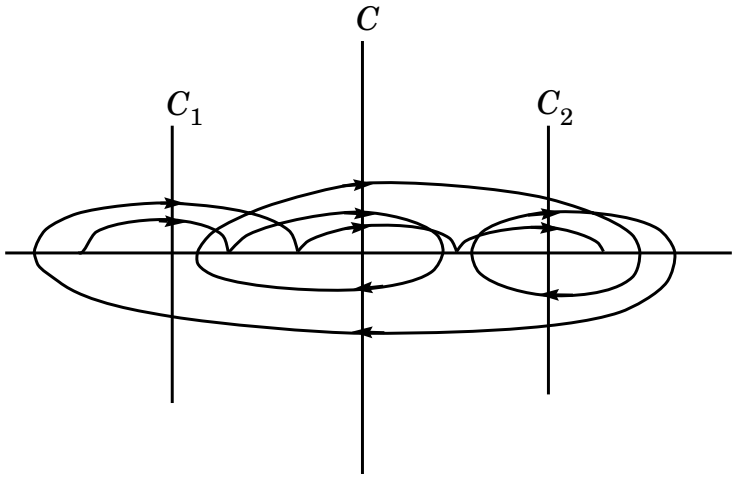

Fig. 3. The intuitive observation of bisecting.

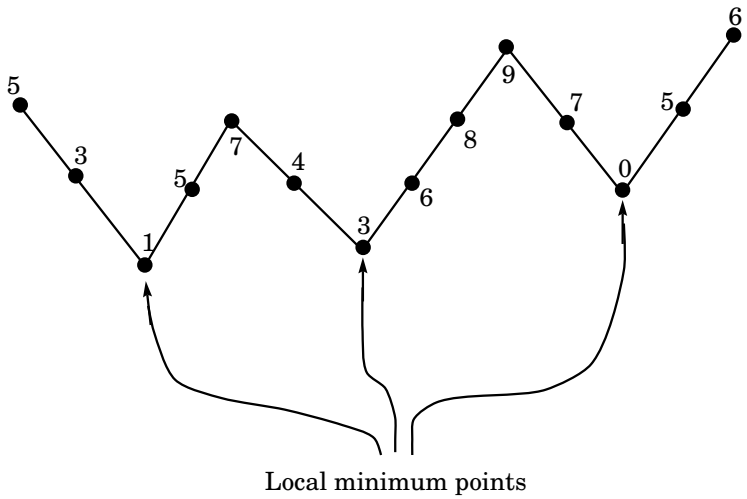

Fig. 4. Cutting a linked list.

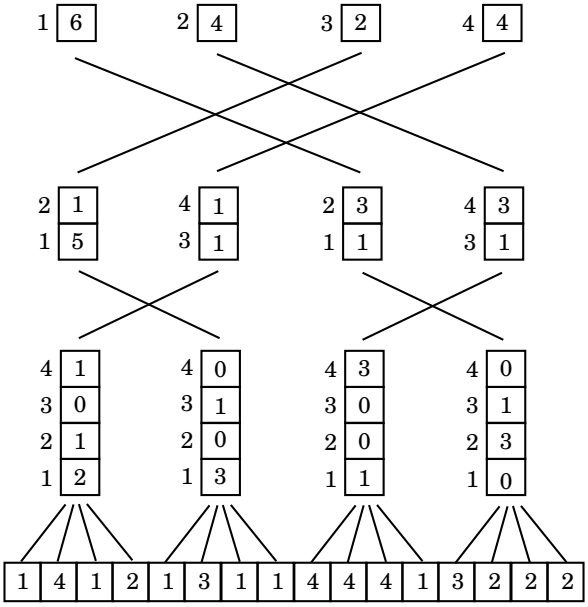

Fig. 5. A sorting scheme.