Parallel Algorithms for Computing Linked List Prefix

Yijie Han

Department of Computer Science, Duke University, Durham, North Carolina 27706

Received May 20, 1987

Given a linked list \( x_1, x_2, ..., x_n \) with \( x_i \) following \( x_{i-1} \) in the list and an associative operation \( \circ \), the linked list prefix problem is to compute all prefixes \( \bigcirc_{j=1}^{i} x_i, j = 1, 2, ..., n \). In this paper we study the linked list prefix problem on parallel computation models. A deterministic algorithm for computing a linked list prefix on a completely connected parallel computation model is obtained by applying vector balancing techniques. The time complexity of the algorithm is \( O\left(\frac{n}{p} + p \log p\right) \), where \( n \) is the number of elements in the linked list and \( p \) is the number of processors used. Therefore our algorithm is optimal when \( n \geq p^2 \log p \). A PRAM linked list prefix algorithm is also presented. This PRAM algorithm has time complexity \( O\left(\frac{n}{p} + \log p\right) \) with small multiplicative constant. It is optimal when \( n \geq p \log p \).

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 \( \circ \), the linked list prefix problem is to compute all prefixes \( \bigcirc_{j=1}^{i} x_i, j = 1, 2, ..., n \). In this paper we study the linked list prefix problem on two parallel computation models — the direct connection machine model [6] and the PRAM (parallel random access machine) model [2].

In a PRAM [2] model memory cells are shared by processors and each cell can be accessed by any processor. The conflicts generated with simultaneous memory access are resolved by imposing conflict resolving rules. Commonly used rules and the corresponding PRAM models are the following three [14]:

* CRCW (concurrent read concurrent write) model: In each step a memory cell can be read or written by several processors simultaneously. The resulting content of the cell after concurrent write can be decided by assuming that an arbitrary processor will succeed in writing.

* CREW (concurrent read exclusive write) model: In each step a memory cell can be read by several processors simultaneously. But simultaneous writes are prohibited.

* EREW (exclusive read exclusive write) model: Both simultaneous read and simultaneous write are prohibited.

A direct connection machine (DCM) [6] has an equal number of processors and memory modules. Any processor can access any cell in any memory module, but simultaneous access to any memory module is prohibited. This restriction generates the problem of “hot spot” [11] — a memory module to be accessed by several processors simultaneously. This problem is not accounted for on the PRAM. Upfal and Wigderson showed [17] that a DCM can simulate a PRAM with the loss of a factor \( \log n \log \log n \) in time complexity (their version of DCM is stronger than the one defined here). Because their simulation is not optimal, it cannot yield optimal algorithms. The work of Kruskal et al. [6] showed that it is possible to obtain optimal algorithms even for “irregular” memory reference problems such as the linked list prefix problem.

The linked list prefix problem is a fundamental problem which has been studied extensively [5-7, 13, 18, 19]. It is known that many tree problems can be reduced to the linked list prefix problem and...
many graph and other problems require fast and efficient solutions to the linked list prefix problem [5, 8, 15]. Among these problems are packing, processor allocation, expression evaluation, network routing, and code decomposition.

Recently Kruskal et al. showed an algorithm [6] for computing linked list prefix on the DCM. The time complexity of their algorithm is \(O(n/p + p^2)\), where \(n\) is the number of elements of the linked list and \(p\) is the number of processors in the DCM. When \(n \geq p^3\), their algorithm is optimal. In this paper we show an improved result. We present a DCM algorithm with time complexity \(O(n/p + p \log p)\). Therefore our DCM algorithm is optimal when \(n \geq p^2 \log p\). An interesting feature of our algorithm is the application of vector balancing techniques. The vector balancing techniques we have discovered enable us to resolve memory module conflicts in an efficient way. We believe that vector balancing techniques can find applications in many other problems.

On the PRAM model we discovered a matching partition function for a linked list [5]. This function enables us to compute a maximal matching for a linked list efficiently. Cole and Vishkin independently discovered this function but termed it “deterministic coin tossing” [3]. We present here a PRAM CRCW algorithm with time complexity \(O(n/p + \log p)\). The algorithm is optimal when \(n \geq p \log p\). Our algorithm is practical in the sense it has small multiplicative constant (less than 20). Cole and Vishkin claimed time complexity \(O(n/p + \log n)\) for their algorithm [4] which uses unusually complicated techniques and has huge multiplicative constant.

We emphasize our intuitive observation (in Section 4) which led us to the discovery of the matching partition function. This intuitive observation was not independently observed by others.

Our PRAM linked list prefix algorithm finds an important application in the parallel connected component problem [5]. The PRAM connected component algorithm shown in [5] was obtained by applying our edge and graph reduction techniques and the techniques for linked list contraction. The PRAM connected component algorithm [5] is optimal when \(e \geq n \log^* n\), where \(e\) is the number of edges and \(n\) is the number of nodes of the input graph.

A known approach to solving the linked list prefix problem efficiently is linked list contraction [10]. In each stage the linked list is contracted to a shorter list until the whole list is contracted to a single node. To accomplish such a stage, a matching (a matching of a graph is a set of edges such that no two edges in the set are incident on the same node) is found for the list, then for each edge \((v, w)\) in the matching combine \(v\) and \(w\) using the associative operator \(\circ\). After the combining step the edges in the matching are deleted resulting in a contracted list. This combining process is also called pair-off [7] and is illustrated in Fig. 1. After the list is contracted to a single node the computation can be reversed to expand the single node to the original list. The prefix values can be evaluated during the process of contracting and expanding. The principle of this approach is explained in detail in [5, 18]. Readers who are not familiar with this approach are referred to [5, 18].

The rest of the paper is organized as follows. In Section 2 we present two utility algorithms for the DCM. One algorithm is the load balancing algorithm. For a DCM with different number of data items in different memory modules the algorithm moves data items from memory modules to memory modules so that the number of data items in different memory modules differs by no more than one. The time complexity of this load balancing algorithm matches the lower bound. The other algorithm is the data-permuting algorithm. The permutation of data items on the DCM is accomplished by using integer sorting technique which yields linear speedup when \(n \geq p^{1+\epsilon}, \epsilon > 0\).

In Section 3 we present vector balancing techniques and demonstrate how to eliminate hot spots by balancing vectors. We then give our DCM linked list prefix algorithm. In Section 4 the PRAM linked list prefix algorithm is presented.

2. LOAD BALANCING AND DATA PERMUTING ON THE DCM

In this section we show how to balance load and permute data on the DCM. The procedures
presented here could be useful in constructing algorithms for the DCM. They will be used in our DCM linked list prefix algorithm. The time complexity of our load balancing procedure matches the lower bound. We will use an integer sorting algorithm to permute data. We combine Leighton’s column sort scheme [9] with sequential bucket sorting scheme to obtain a DCM integer sorting algorithm to sort \( mp \) integers in \( p \) memory modules. For integers in the range \( \{1, 2, \ldots, n^{O(1)}\} \) this algorithm is optimal when \( m \geq p^\epsilon, \epsilon > 0 \). It is interesting to note that known deterministic PRAM algorithms [18] for integer sorting are optimal when \( n/p = m \geq p^\epsilon, \epsilon > 0 \), where \( n \) is the number of data items, although there is a randomized PRAM algorithm for integer sorting with time complexity \( O(n/p + \log p) \) [12].

Let memory module \( m \) contain \( n_m \) data items. The load balancing problem is to redistribute data items such that the numbers of data items in different memory modules differs by no more than one. Balancing a seriously unbalanced system could be time consuming. Suppose initially all \( n \) data items are in one memory module. It will take about \( n \) operations to distribute these data items to other memory modules because the memory module containing \( n \) data items forms a bottleneck. However, to efficiently utilize a DCM, we intend to distribute tasks to processors so that the initial loads among processors are balanced. As the computation proceeds, the balanced system may become unbalanced. But if sufficiently few computations are performed, the balanced system could hardly become a seriously unbalanced one. Thus the usual task of balancing is to balance a slightly unbalanced system. We present a procedure which balances a system with \( p \) memory modules and a maximum of \( M \) data items in any module in time \( O(M + \log p) \). Our result matches the lower bound because \( \Omega(M) \) is the obvious lower bound and \( \Omega(\log p) \) can be obtained by a simple fan-in argument.

The basic idea of this load balancing algorithm is to find a permutation of memory modules such that offering memory modules will have receiving memory modules nearby. We explain the scheme while presenting our algorithm. Figure 2 demonstrates the idea of procedure BALANCE by an example.

Procedure BALANCE

Step 1: Compute the average number \( a \) of data items for memory modules and broadcast this average number to all modules. Add some null data items if necessary to make \( a \) an integer.

Step 2: For memory module containing \( m \) data items label it with number \( m - a \). The sign of the label indicates whether the memory module is an offering module or a receiving module. The absolute value of the label indicates how many data items the module can offer or receive.

Step 3: Modules labeled with zeros have already attained average number of data items. They can be ignored from now on. Divide modules into two groups with modules labeled with positive numbers in one group and modules with negative numbers in another group.

Step 4: For each group, compute prefix sum of their labels. The prefix values form an ascending (descending) sequence. The prefix value of memory module \( m \) indicates how many data items can be offered/received by the memory modules up to and including memory module \( m \).

Step 5: Merge the two sequences by the absolute value of their prefix values using Batcher’s bitonic merging algorithm [1]. Ties are broken by viewing the negative label as the smaller one. The merge yields a permutation of memory modules.

Step 6: The permutation obtained in step 5 is a sequence consisting of offering module subsequences and receiving module subsequences. A receiving module in a receiving module subsequence, if it is not the leftmost module in this subsequence, receives data from the leftmost offering module in the offering module subsequence immediately to the right of the receiving module subsequence. An offering module in an offering module subsequence, if it is not the leftmost module in this subsequence, sends data to the leftmost receiving module in the receiving module subsequence immediately to the right of the offering module subsequence. In the example of Fig. 2, \( m_2 \) receives data items from \( m_1 \), \( m_4 \) sends data items to \( m_3 \), and both \( m_5 \) and \( m_7 \) receive data items from
Step 7: Transfer data items.

Steps 1 and 7 take \(O(M + \log p)\) time. Step 2 takes constant time. Steps 3 to 6 each take \(O(\log p)\) time. The time complexity of Procedure BALANCE is thus \(O(M + \log p)\).

Now we show how to sort integers in the range \(\{1, 2, \ldots, n^{O(1)}\}\) on the DCM. This is a simple application of Leighton’s column sort [9]. Let each memory module contain \(m\) integers in \(p\) memory modules. Each module is viewed as one column. The sorted sequence is to be in the row-major order; i.e., integers in row \(i\) are no greater than integers in row \(j\) if \(i < j\).

Leighton’s column sort tells us that to sort a sequence of \(n\) elements one needs only to sort \(n^{1/3}\) sequences of \(n^{2/3}\) elements a constant number of times. Recursively applying Leighton’s column sort, we observe that to sort a sequence of \(n\) elements one needs only to sort \(n^{1-\epsilon}\) sequences of \(n^\epsilon\) elements for \(f(\epsilon)\) times, where \(0 < \epsilon \leq 1\) and \(f(\epsilon)\) is a function of \(\epsilon\). When \(\epsilon\) is fixed, \(f(\epsilon)\) is a constant. Therefore we obtain an optimal algorithm for sorting integers on a DCM when \(m > p^2\).

Note that a sequential radix sorting algorithm is used within each memory module.

3. Vector Balancing Techniques and a DCM Linked List Prefix Algorithm

3.1 The Vector Balancing Problem

Given a set \(S\) and \(m\) vectors of \(n\) components: \(e_{ij} \in S, 1 \leq i \leq m, 1 \leq j \leq n. e \in S\) is balanced with respect to the \(m\) vectors if the number of occurrences of \(e\) in each of the \(m\) vectors differs no more than one. The \(m\) vectors are balanced if for any \(e \in S, e\) is balanced with respect to the \(m\) vectors.

The vector balancing problem is to rearrange \(e_{ij}\)’s, under the restriction that \(e_{ij}\) can be moved only to the position of \(e_{kj}\) (i.e., component \(j\) stays as component \(j\)), such that \(m\) vectors are balanced.

We explain in Section 3.2 why the vector balancing problem is related to the resolution of memory access conflicts, i.e., removing hot spots, for a linked list stored in a DCM. This relation motivates us to study the vector balancing problem.

The \(i\)th component \(e_{1i}\) and \(e_{2i}\) of two vectors \(V_1\) and \(V_2\) is called a pair and denoted by \((e_{1i}, e_{2i})\).

A partition \(P = \{P_1, P_2, \ldots, P_s\}\) of all pairs of \(V_1\) and \(V_2\) is a primitive chain partition, where \(P_i\) is denoted by \([a_i, b_i]\) and called a primitive chain, when the following conditions are met:

(i) \(P_i\) is a sequence of pairs:

\[
P_i = \left[ \begin{array}{c} a_i \\ b_i \end{array} \right] = \left( \begin{array}{c} e_{1i} \\ e_{2i} \\ e_{12} \\ e_{22} \\ \cdots \\ e_{1k} \\ e_{2k} \end{array} \right).
\]

(ii) \(a_i = e_{11}, b_i = e_{2i}, e_{2i+p} = e_{1i+p+1}, p = 1, 2, 3, \ldots, k - 1.\)

(iii) \(a_i \neq b_j\) if \(i \neq j.\)

Note that for any \(e, e \neq a_i, b_i, e\), the number of occurrences of \(e\) in \(P_i \cap V_1\) is equal to the number of occurrences of \(e\) in \(P_i \cap V_2\). If we exchange all elements in \(P_i \cap V_1\) with all elements in \(P_i \cap V_2\), we are adding one more occurrence of \(a_i\) to \(V_2\) and \(b_i\) to \(V_1\) and subtracting one occurrence of \(a_i\) from \(V_1\) and \(b_i\) from \(V_2\). This is the reason we use \([a_i, b_i]\) to denote primitive chain \(P_i.\)

Let \(P = \{P_1, P_2, \ldots, P_s\}\), where \(P_i = [a_i, b_i]\), be a primitive chain partition. A partition \(C = \{C_1, C_2, \ldots, C_t\}\) of \(P\) is a chain partition and \(C_i\) is a chain when the following conditions are met:
(i) $C_i$ is a sequence of primitive chains:

$$C_i = \begin{bmatrix} a_{i1} \\ b_{i1} \\ \vdots \\ a_{i\ell} \\ b_{i\ell} \end{bmatrix}.$$

(ii) Either $a_{ip} = a_{ip+1}$ and $b_{ip} = b_{ip+1}$, or $b_{ip} = b_{ip+1}$ and $a_{ip+1} = a_{ip+2}$, $p = 1, 3, 5, 7,$ ....

(iii) The concatenation of $C_i$ and $C_j$, $C_iC_j$ or $C_jC_i$, $i \neq j$, is not a chain.

A simple strategy for constructing a primitive chain is to match elements in $V_1$ to elements in $V_2$ by adding links between matched elements. When no more links can be added, each linked list (or linked cycle) identifies a primitive chain. A chain can be constructed in a similar way by adding links to matched elements in $V_1$ and to those matched elements in $V_2$. Since the strategy is close to trivial, we omit the procedural description of the strategy and the proof of its correctness. An example of constructing primitive chains and chains is shown in Figs. 3a and 3b.

We first present two sequential vector balancing procedures.

**Lemma 1** (Procedure VECTOR1). Two $n$-component vectors can be balanced using one processor in time $O(n)$.

**Proof.** Execute the following procedure:

**VECTOR1**($V_1$, $V_2$)

1. Construct a chain $C$ for vectors $V_1$ and $V_2$.
2. For every other primitive chain $P$ in $C$ exchange elements in $P \cap V_1$ with elements in $P \cap V_2$.
3. Delete $C$ from $V_1$ and $V_2$.
4. Let $V_1'$ and $V_2'$ be the resulting vectors after the deletion in step 3, call **VECTOR1**($V_1'$, $V_2'$).

Note that after step 2, elements in chain $C$ are balanced with respect to $V_1$ and $V_2$ and, therefore, can be deleted in step 3.

The elements of the two vectors can be placed in buckets so that each element can be accessed in constant time by indexing into the buckets. Thus a chain can be constructed in time proportional to the number of elements in it. Therefore the time complexity of balancing two vectors is $O(n)$.■

An example of balancing two vectors is shown in Fig. 3.

**Lemma 2** (Procedure VECTOR2). $m = 2^k$ $n$-component vectors can be balanced using one processor in $O(mn \log m)$ time.

**Proof.** First balance two big vectors $V_1$, $V_2$ of $n 2^{k-1}$ components. Each big vector is formed by concatenating $2^{k-1}$ vectors. After $V_1$ and $V_2$ are balanced. Recursively balance the vectors within $V_1$ and $V_2$. The time complexity is $O(mn \log m)$.■

It would be nice to have a sequential algorithm with time complexity $O(mn)$. Also it would be nice to have procedure VECTOR2 work for any size $m$ with time complexity no greater than $O(mn \log m)$, although this will not affect our applications except for a constant factor. Because in our applications of procedure VECTOR2, we can add null vectors to make the number $m$ to be a power of 2.

Now we consider balancing vectors on the DCM.

**Lemma 3** (Procedure VECTOR3). Two vectors of $k \log k$ components can be balanced on a DCM with $\log k$ processors in time $O(k)$ if vector elements are integers in the range $\{1, 2, ..., k^{O(1)}\}$.

**Proof.** Let $V_1$ and $V_2$ be two vectors we wish to balance. Refer to the example shown in Fig. 3. **VECTOR3** first matches elements in $V_1$ with elements in $V_2$ to form primitive chains, as shown in Fig. 3a. Links between matching elements are added by employing the DCM integer sorting algorithm. Next, chains are formed by chaining primitive chains together as shown in Fig. 3b. This can also be done by sorting. Finally every other primitive chain in each chain is identified and elements in $V_1$
are exchanged with those in $V_2$, as illustrated in Fig.3c. The linked list prefix algorithm of Kruskal et al. [6] is used to identify every other primitive chain.

$O(k)$ time is needed for sorting and the linked list prefix algorithm of Kruskal et al. achieves time complexity $O(k)$ because $n = k \log k > p^3 = \log^3 k$. Therefore the whole procedure takes time $O(k)$. ■

Note that the time complexity of VECTOR3 is dominated by the linked list prefix algorithm of Kruskal et al. [6]. If two vectors of $n$ components are to be balanced on a DCM with $p$ processors, optimal time $O(n/p)$ is achieved by VECTOR3 if $n \geq p^3$. Condition $n \geq p^3$ can be improved by using a better DCM linked list prefix algorithm. Although a modified version of our DCM linked list prefix algorithm can be used to improve condition $n \geq p^3$ to $n \geq p^2 \log p$ for VECTOR3, it will not be discussed here.

Let $i_0, i_1, \ldots, i_{p-1}$ be $p$ integers, where $p$ is a power of 2. Let us “balance” these integers using the following procedure. In step $j$ compute $[(i_l + i_{l+2})/2]$ and $[(i_l + i_{l+2})/2]$, and assign one of them to $i_l$, the other to $i_{l+2}$, where $l = 0, 1, \ldots, p-1$ and $\oplus$ is the bit-wise exclusive-or operation. This procedure uses $\log p$ steps. The communication pattern of this procedure is exactly that depicted by the butterfly network [16].

**Lemma 4.** After the above procedure is executed, the difference of the largest and the smallest integer among $p$ integers is at most $\log p$.

**Proof.** Suppose the largest integer and the smallest integer in the sequence $a_0, a_1, \ldots, a_m$ are $A_1$, $A_2$; those in $b_0, b_1, \ldots, b_m$ are $B_1, B_2$. Consider one “balance” operation performed on these two sequences; i.e., compute $[(a_i + b_i)/2]$ and $[(a_i + b_i)/2]$ and assign one of them to $a_i$ and the other to $b_i$. The smallest integer in the new sequence is $[(A_1 + B_1)/2]$ and the largest is $[(A_2 + B_2)/2]$. Their difference is no greater than $|(A_1 - A_2) + (B_1 - B_2)|/2 + 1$. Note that when $A_1 - A_2 = B_1 - B_2$ the new difference of the largest integer and the smallest integer is at most the old difference plus 1. Thus the lemma follows. ■

Let $p$ $p$-component vectors contain $p$ $1$’s, $2$’s, $\ldots$, $p$’s. If these vectors are balanced, then each balanced vector is a permutation of $1, 2, \ldots, p$. We shall call these vectors partially balanced if no vector contains more than constant number of $i$’s, $i = 1, 2, \ldots, p$.

**Lemma 5 (Procedure VECTOR4).** $p$ $p$-component vectors containing $p$ $1$’s, $2$’s, $\ldots$, $p$’s can be partially balanced on a $p$ processor DCM in time $O(p \log p)$, assuming both $p$ and $\log p$ are powers of 2.

**Proof.** Step 1: We first concatenate $\log p$ vectors to form a big vector. There is a total of $p/\log p$ big vectors. These big vectors are partially balanced using Lemma 4; i.e., in stage $i$ big vectors $V$ and $V \oplus 2^i$ are balanced using procedure VECTOR3, $i = 0, 1, \ldots, \log p - \log \log p - 1$. By Lemma 4, after the partial balancing process the difference of the number of an integer $j$ in any two big vectors is at most $\log p$. Let $M_j = \max\{\text{the number of } j\text{'s in big vector } V | V = 0, 1, 2, \ldots, p/\log p - 1\}$ and $m_j = \min\{\text{the number of } j\text{'s in big vector } V | V = 0, 1, 2, \ldots, p/\log p - 1\}$. We have $M_j - m_j \leq \log p$ and $m_j \leq \log p$. Thus any big vector can have at most $2 \log p$ $j$’s.

Step 2: $\log p$ vectors in each big vector are balanced using the scheme of VECTOR2; i.e., we form two vectors $v_1, v_2$ each containing $\log p/2$ vectors, balance $v_1$ and $v_2$ using procedure VECTOR3, then recursively balance vectors in $v_1$ and $v_2$.

After the above balancing steps, each of the $p$ vectors contains no more than $2$ $i$’s, $i = 0, 1, \ldots, p-1$.

Step 1 has $\log p - \log \log p$ stages each of which takes $O(p)$ time. Step 2 has $\log p$ stages each of which takes $O(p)$ time. Therefore the time complexity of VECTOR4 is $O(p \log p)$. ■

3.2. A DCM Linked List Prefix Algorithm

As we have mentioned in Section 1, previous methods used for obtaining optimal algorithms for the linked list prefix problem contract the list by pairing-off adjacent elements of the list [7, 10]. In
this subsection a DCM algorithm ListContract1 is given. For a linked list of length \( n \), ListContract1 finds a matching which contains at least \( n/4 \) pointers of the list. The head and tail of the pointers in the matching set are paired-off. Therefore ListContract1 contracts a linked list from length \( n \) to length at most \( 3n/4 \).

If we take a look at the pointers of a linked list stored in a DCM, they are pointing “irregularly” to each other. If each of the \( p \) processors examines the first pointer in the \( p \)th memory module, it is likely that many of these pointers are pointing to the same memory module. Thus the memory module with many pointers pointing to it becomes a hot spot. Before we start processing the linked list we ought to organize the process or to rearrange the linked list so that in each step of the processing no hot spot will be formed.

Each pointer of the linked list can be represented by two pairs: the module link \( < a, b > \) and the position number \( < c, d > \). The module link \( < a, b > \) specifies that the tail of the pointer is in memory module \( a \) and the head is in memory module \( b \). The position number \( < c, d > \) specifies the address of the pointer’s head and tail within memory modules. Consider the module links of all the pointers processed in one computation step. If no memory module appears more than once in these links, then no hot spot is formed for this computation step. To organize the computation without forming hot spots we need only to deal with the module links of the pointers. Henceforth when we mention pointers we implicitly mean the module links of the pointers.

Let \( L \) be an \( mp \) element linked list stored in \( p \) memory modules with \( m \) pointers in each module. If we view the pointers as \( m \) vectors of \( p \) components, then if \( m \) vectors are balanced (by the heads of the pointers), the heads of the pointers in each vector are permutations of \( \{1, 2, \ldots, p\} \).

Thus if we first balance the \( m \) vectors we can then process the linked list without forming hot spots.

Note that VECTOR4 is not an optimal algorithm (for an optimal algorithm would have time complexity \( O(p) \)). In order to obtain an optimal algorithm for the linked list prefix problem, we group pointers. Pointers originating from one memory module and pointing to the same memory module can be put into one group. This grouping idea is used in [6]. Now each element in a vector is not a single pointer, but a group of pointers. After the vectors are partially balanced, we process groups of pointers, one vector of these groups at a time. This method leads to an optimal algorithm for the linked list prefix problem.

Now let us consider the processing of a partially balanced vector. The \( i \)th component of the vector has the form \( < i, k > \), where \( k \) is the memory module this pointer (or this group of pointers) points to. Since the vectors are partially balanced (by the heads of the pointers), the \( k \)th memory module can appear no more than twice as heads in the pointers of the vector. If \( k \)th memory module appears more than once, we delete one \( k \) so that each memory module appears at most once. To ensure that an element will not be paired with both its predecessor and successor, we choose pointers from the partially balanced vector so that no two pointers chosen are incident on the same memory module. It can be easily verified that we can choose at least one-half of the pointers remaining after pointer deletion. Thus from each vector we can choose at least one-fourth of total pointers for processing without memory module access conflicts.

Below we give the details of our DCM algorithm for contracting a linked list. Procedure ListContract1 contracts a linked list of length \( mp \) to a list of length \( 3mp/4 \), \( m \) is the number of elements in a memory module. Procedure ListContract1 assumes that \( m \geq p \log p \).

Procedure ListContract1:

Step 1: For pointers in each memory module form groups. Each group has exactly \( \tau = m/p \geq \log p \) pointers and pointers in one group are all pointing to the same memory module. Thus there are at most \( m/\tau = p \) groups in each module. Add null groups if the number of groups in a memory module is less than \( p \). A group in memory module \( i \) with pointers pointing to module \( j \) is denoted

\[<i, k>\]

This statement is exact for a linked cycle of \( mp \) element and it can be accommodated for a linked list.
by a pair \(<i,j>\). \(i\) is the tail of the group and \(j\) is the head. A null group in memory module \(i\) is denoted by \(<i,nil>\).

Step 2: The groups we obtained form \(p\) vectors, the \(i\)th component of a vector is a group in memory module \(i\). Transpose the vectors in the array of memory to store one vector into each memory module, execute:

\[
\text{for } i := 0 \text{ to } p - 1 \text{ step 1} \\
\text{for all processors } k: 0 \leq k < p \\
\text{\hspace{1em}begin} \\
\hspace{2em}processor k \text{ gets } 1 \text{ group from memory} \\
\hspace{2em}module (k + i) \mod p \text{ and stores it in} \\
\hspace{2em}memory module k. \\
\text{\hspace{1em}end}
\]

Step 3: In step 2 every memory module has one vector. The \(i\)th component of the vector has the form \(<i,k>\), where \(k\) can be either nil or any integer from 0 to \(p - 1\). Execute procedure VECTOR4 to partially balance vectors by the heads of their components. After balancing the number of occurrences of \(k\), \(0 \leq k < p\) in the heads of any vector is no more than 2.

Step 4: In the case \(0 \leq k < p\) occurs twice in the heads of a vector change one of them to nil so that \(k\) has no more than one occurrence in any vector. Note that the number of groups which are not changed to a null group consists of at least one-half of the number of original nonnull groups. Also for nonnull groups in each vector, check if they form a module chain \(<i_1,i_2>,<i_2,i_3>,<i_3,i_4>,<i_4,i_5>,\ldots\). For groups in a module chain, null at most one-half of them to break the chain. Now the number of nonnull groups is at least one-fourth of the number of the original nonnull groups.

Step 5: Store groups back to the memory modules where they were taken. This step is an inverse of step 2 except the vectors are now partially balanced. After this step vectors are stored row by row in memory modules with one vector occupying one row, if we view each memory module as a column.

Step 6: Processing one vector at a time. For each pointer contained in a group of the vector combine the head and the tail of the pointer if its tail was not touched by previous processing. Mark the head and tail of the pointer as being processed and update the pointer to point to the next element.

Step 7: Step 6 has processed at least one-fourth of the pointers included in groups. This step processes pointers which are not included in any group:

\[
\text{for } i := 0 \text{ to } p - 1 \text{ step 1} \\
\text{for all processors } k: 0 \leq k < p \\
\text{\hspace{1em}begin} \\
\hspace{2em}Deactivate one-half of the processors to break processor chains \(<k,(k+i) \mod p>,< (k+i) \mod p,(k+2i) \mod p>,\ldots\). If processor \(k\) is not deactivated, it processes at most \(\tau\) pointers with tails in memory module \((k+i) \mod p\) and heads in memory module \(k\). The tail and head of such a pointer are paired-off using \(\bigcirc\) operation. The elements are marked as being processed and the pointers are updated. \\
\text{\hspace{1em}end}
\]

Step 8: Pack the remaining elements. This is done by the load balancing algorithm (procedure BALANCE). After packing, the pointers of the remaining elements must be modified. This can be done using the data permutation algorithm.

**THEOREM 1.** A linked list of \(n\) elements can be contracted to a single node on a \(p\) processor
DCM in time $O(n/p + p \log p)$.

**Proof.** When $n \geq p^2 \log p$, ListContract1 can be invoked to contract the linked list. As we explained before, one execution of ListContract1 reduces the length of the linked list by a fraction of $\frac{3}{4}$. Let us analyze the time complexity of ListContract1. Step 1 takes $O(m)$ time. Step 2 takes $O(p)$ time. Step 3 takes $O(p \log p)$ time. Steps 4 and 5 take $O(p)$ time. Step 6 takes $O(m)$ time. Step 7 takes $O(p \log p)$ time. Step 8 takes $O(m + \log p)$ time since $m \geq p \log p$. Thus the overall time complexity is $O(m + p \log p) = O(n/p + p \log p)$.

When $m < p \log p$, one can add null entries to satisfy the condition $m \geq p \log p$ and then execute ListContract1.

To contract the list to a single node we first execute ListContract1 $O(\log n)$ times to contract the list of length $n$ to length $p \log p$. Then a sequential algorithm can be used to contract the list to a single node. We use BALANCE to pack the contracted list to fewer memory modules and use fewer processors to satisfy the condition $m \geq p \log p$. Since each execution will reduce the length of the list by a constant factor, it follows that both the length of the linked list and the number of processors used form geometric serieses and the time complexity for contracting the list to a single node is still $O(n/p + p \log p)$ which is optimal when $n \geq p^2 \log p$. ■

4. A PRAM LINKED LIST PREFIX ALGORITHM

In this section we present a PRAM linked list prefix algorithm. This algorithm has time complexity $O(n/p + p \log p)$ and is optimal when $n \geq p \log p$.

As we have mentioned, a matching is needed in order to contract a linked list. A matching is a maximal matching if adding one more edge to the matching results in a nonmatching set. Below we give fast and efficient algorithms for finding a maximal matching for a linked list.

We assume that the input linked list of length $n$ is stored in an array addressed from 0 to $n - 1$, as shown in Fig. 4. A pointer stored in address $NEXT[a]$ and pointing to address $b$ is denoted by $<a, b>$. A pointer $<a, b>$ is called a forward pointer if $b > a$, otherwise it is called a backward pointer. 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 $pre(v)$. By drawing a bisecting line $c$ of the array as shown in Fig. 5, we observe that forward pointers crossing line $c$ constitute a matching, because no two pointers can originate from the same address or point to the same address and one pointer’s head cannot be another pointer’s tail since both cross line $c$. Therefore two matching sets can be associated with line $c$, one containing forward pointers crossing line $c$ and the other containing backward pointers crossing line $c$. Line $c$ divides the array into two subarrays. We now draw bisecting lines $c_1$, $c_2$ for the two subarrays. Forward pointers crossing either $c_1$ or $c_2$ but not $c$ constitute a matching. Continuing to divide subarrays and forming matching sets in this way, we see that $2 \log n$ matching sets can be formed. We also observe that for pointer $<a, b>$, $\max\{i\}$ the $i$th bit of $a \oplus b$ is 1 characterizes the bisecting line with which pointer $<a, b>$ should be associated, where $\oplus$ is the bit-wise exclusive-or operation and bits are counted from the least significant bit starting from 0. We define function $f$ to characterize our intuitive observation:

$$f(a, b) = \text{def } 2k + a_k, \quad \text{where } k = \max\{i\}\text{ the }i\text{th bit of }a \oplus b\text{ is }1\text{ and }a_k\text{ is the }k\text{th bit of }a.$$  

**Lemma 6.** Function $f$ partitions the pointers of an $n$-element linked list into $2\lceil \log n \rceil$ matching sets.

**Proof.** The proof is by our intuitive observation stated above and by noting that $a_k$ indicates whether $<a, b>$ is a forward pointer or a backward pointer. ■

Function $f$ is called a matching partition function [5].

If concurrent read is allowed, then an index table can be set up for processors to find value $k$. Value $a_k$ can be found by comparing $a$ and $b$. On the EREW model function $f$ is modified to

---

2 Results presented in this section are part of the author’s Ph.D. thesis work [5].
\( f(a, b) = 2k + a_k \), where \( k = \min\{i \mid \text{the } i\text{th bit of } a \oplus b \text{ is } 1 \} \) and \( a_k \) is the \( k \)th bit of \( a \) and the table technique [5, 18] can be used to satisfy \(^3\):

(a) The space for storing these tables is \( O(p \log n) \), where \( p \) is the number of processors used.

(b) The function value for up to \( p \) pointers can be evaluated in parallel in constant time.

Note that the intuitive observation stated above was not observed by Cole and Vishkin [3] and no scheme is specified in [3] to satisfy both conditions (a) and (b).

When function \( f \) is applied to a linked list, it assigns an integer in the range \( \{0, 1, 2, \ldots, 2\lfloor \log n \rfloor - 1\} \) to each pointer (and therefore to each node except the last one) of the list. Since \( f \) is a matching partition function, no two adjacent nodes are assigned the same number. Therefore we can identify turn points — nodes which are assigned local minimum values, as shown in Fig. 6. The linked list can be cut at these turn points to obtain many shorter lists. The facts that integers in the range \( \{0, 1, 2, \ldots, 2\lfloor \log n \rfloor - 1\} \) are assigned to the nodes and no two adjacent nodes are assigned the same number ensure that the distance between two neighboring turn points is at most \( 4\lceil \log n \rceil \). Thus each list obtained after cutting the original list has length no greater than \( 4\lceil \log n \rceil \). We can view the number assigned to each node as the “new address” of the node and apply function \( f \) to the lists again; now integers in the range \( \{0, 1, 2, \ldots, 2\lceil \log(4\lceil \log n \rceil) \rceil \} \) are assigned to the nodes and each list will be cut into shorter lists with length no longer than \( 4\lceil \log(4\lceil \log n \rceil) \rceil \). Let \( \log(1) n = \log n, \log(2) n = \log(\log(1) n) \) and \( G(n) = \min\{i \mid \log(i) n \leq 2\} \), we have:

**THEOREM 2** (Procedure MATCH1). A maximal matching of a linked list can be found in \( O(nG(n)/p + G(n)) \) time using \( p \) processors.

**Proof.** For each node \( v \) compute \( \text{pre}(v) \) and initialize \( l(v) \) to the address of \( v \). Then execute the following steps \( G(n) \) times.

For all nodes do in parallel

1. **Step 1:** Update \( l(v) \). If \( v \) is not the last node in the list then set \( l(v) := f(l(v), l(\text{suc}(v))). \)

2. **Step 2:** If \( v \) is the last node in the list, set \( l(v) := l(\text{pre}(v)) \).

3. **Step 3:** If \( l(\text{pre}(v)) > l(v) \) and \( l(v) < l(\text{suc}(v)) \), then delete either \( < \text{pre}(v), v > \) or \( < v, \text{suc}(v) > \) on the basis of the following criterion. If \( \text{suc}(\text{pre}(v)) \neq \text{nil} \) then delete \( < v, \text{suc}(v) > \) (making \( \text{suc}(\text{pre}(v)) := \text{nil} \)), else delete \( < \text{pre}(v), v > \) (making \( \text{suc}(\text{pre}(v)) := \text{nil} \)). This step cuts the linked list into many shorter lists.

After \( i \) executions of the above steps, the length of each list is no longer than \( c\log(i) n \), where \( c \) is a constant. Thus when the above steps finish each list is of constant length. We can then cut these constant length lists to lists of length 1 (simply allocate one processor to each list and walk down the list to cut it). The time complexity of our algorithm is \( O(nG(n)/p + G(n)) \). Note also that no three consecutive pointers will all be deleted; therefore when the lists are cut to length no more than 1, a maximal matching is found. The process of cutting the linked list can also be viewed as a process of obtaining coarser and coarser matching partitions. When MATCH1 finishes, a partition of three matching sets is obtained.

Although the above algorithm is fast in the sense that if sufficient processors are available only time \( O(G(n)) \) is needed for finding a maximal matching, it is not optimal because the number of operations taken by the algorithm is greater than \( O(n) \). Below we give an optimal algorithm for finding a maximal matching for a linked list. Our algorithm uses an integer sorting algorithm which is an adaptation of Reif’s partial sum algorithm [12].

**LEMMA 7.** Let \( A[1..n] \) be an array of integers of magnitude \( n^{O(1)} \). The partial sum of \( A \), \( AP[i] = \sum_{j=1}^{i} A[j] \), \( i = 1, 2, \ldots, n \), can be computed in \( O(n/p + \log n/\log^3 n) \) time using \( p \) processors.

\(^3\)While the method of evaluating \( f \) using the most significant nonzero bit of \( a \oplus b \) is related to our intuitive observation, the method of using the least significant bit [5, 18] is convenient for evaluating \( f \) on the EREW model and it does not seem to be related to any intuition.
on the CRCW model.

Proof. Assume \( p \geq n^{1/2} \), otherwise simulate \( n^{1/2} \) processors using \( p \) processors. First compute \( B[1..p] \), where \( B[i] = \sum_{j=1}^{\lfloor n/p \rfloor} A[n(i-1)/p + j] \). Then compute \( BP[1..p] \), where \( BP[i] = \sum_{j=1}^{i} B[j] \). \( AP[1..n] \) can now be computed. \( O((\log p)/(\log^3 p)) \) time is needed \([12]\) to compute \( BP[1..p] \). The remaining steps take \( O(n/p) \) time.

**Lemma 8.** \( n \) integers in the range \( \{1, 2, ..., (\log n)^{O(1)}\} \) can be sorted in \( O(n/p + (\log n)/(\log^3 n)) \) time using \( p \) processors on the CRCW model.

Proof. Assume \( p \geq n^{1/2} \), otherwise simulate \( n^{1/2} \) processors using \( p \) processors. Also assume that \( n \) integers \( A[1..n] \) are in the range \( \{1, 2, ..., \log n/\log^3 n\} \). Initialize buckets \( B[1..n] \) to \( \phi \). Denote \( A[i-1]/p + j \) by \( A_i[j] \), \( 1 \leq i \leq p, 1 \leq j \leq n/p \). \( A_i[j] \) is dropped into bucket \( B[(A_i[j] - 1)p + i] \) by processor \( i \). Then elements in \( B \) are packed, resulting in a sorted sequence. Element dropping takes \( O(n/p) \) time. Packing takes \( O(n/p + (\log n)/(\log^3 n)) \) time, by means of the partial sum algorithm of Lemma 7.

By using the idea of radix sorting we can extend our algorithm to sort integers in the range \( \{1, 2, ..., (\log n)^{O(1)}\} \).

**Theorem 3 (Procedure MATCH2).** A maximal matching of a linked list can be found in \( O(n/p + \log n/\log^3 n) \) using \( p \) processors on the CRCW model.

Proof. Assume that the linked list is represented by array \( NEXT[1..n] \), where \( NEXT[i] \) represents pointer \( <i, NEXT[i]> \). We execute the following algorithm.

**MATCH2( )**

Step 1: Divide pointers into \( 4[\log(4[\log n])] \) matching sets using matching partition function \( f \) twice (i.e., execute the three steps in MATCH1 twice) and label each pointer with its set number.

Step 2: Sort pointers by their set numbers.

Step 3:

begin
  \( M := \phi; \)
  for all nodes \( v \) do in parallel
  \( DONE[v] := false; \)
  for \( k := 0 \) to \( 4[\log(4[\log n])] - 1 \) do
    for all pointers \( <v, NEXT[v]> \) in matching set \( k \) do in parallel
    begin
      if \( DONE[v] = false \) AND \( DONE[NEXT[v]] = false \) then
        begin
          \( DONE[v] := true; \)
          \( DONE[NEXT[v]] := true; \)
          \( M := M \cup \{<v, NEXT[v]>\}; \)
        end
    end
  end
end

When the algorithm finishes, pointers in \( M \) constitute a matching. If a node is isolated (neither its incoming pointer nor its outgoing pointer is in \( M \)), then both its predecessor and successor are not isolated. Thus a maximal matching is obtained.

Step one takes \( O(n/p) \) time. By Lemma 8 sorting in step 2 takes time \( O(n/p + \log n/\log^3 n) \). Let \( N(i) \) be the number of pointers in matching set \( i \). Step 3 of MATCH2( ) takes \( O(\sum_{i=0}^{4[\log(4[\log n])]} - 1 \lfloor N(i)/p \rfloor) = O(n/p + \log \log n) \) time.

Our linked linked list contraction algorithm has three stages which are coded using three sub-
routines.

Subroutine A

Step 1: Find a maximal matching \( M \) of the linked list using MATCH2 which has time complexity \( O(n/p + \log n/\log(3)n) \).

Step 2: For all pointers \(<i, \text{NEXT}[i]> \) in \( M \) do:
\[
X[i] := X[i] \odot X[\text{NEXT}[i]]; \quad /* \text{Combine the head and the tail} */
\]
\[
\text{NEXT}[i] := \text{NEXT}[\text{NEXT}[i]]; \quad /* \text{Update the pointer} */
\]

Step 3: Pack the contracted list.

Since a maximal matching of a linked list of \( n \) pointers has at least \( n/3 \) pointers in it, if follows that subroutine A contracts a linked list by a fraction of \( 2/3 \). Subroutine A takes \( O(n/p + \log n/\log(3)n) \) time and is optimal when \( p \leq n \log(3)n/\log n \). Note that in step 3 we use Lemma 7 for packing.

Subroutine B

Step 1: Find a maximal matching \( M \) of the linked list using MATCH1 which has time complexity \( O(nG(n)/p + G(n)) \).

Step 2: For all pointers \(<i, \text{NEXT}[i]> \) in \( M \) do:
\[
X[i] := X[i] \odot X[\text{NEXT}[i]]; \quad /* \text{Combine the head and the tail} */
\]
\[
\text{NEXT}[i] := \text{NEXT}[\text{NEXT}[i]]; \quad /* \text{Update the pointer} */
\]

Subroutine B contracts a linked list by a fraction of \( 2/3 \) in time \( O(nG(n)/p + G(n)) \). It is not optimal, but it has better performance when \( p \) is close to \( n \). Note that subroutine B does not pack the contracted list. In our situation packing would take at least \( \log n/\log(3)n \) time which is worse than the minimum time possible \( O(G(n)) \) subroutine B takes.

Subroutine C is Wyllie’s linked list prefix algorithm [19] which has time complexity \( O(n \log n/p + G(n)) \). When \( p = n \) this is the the best algorithm known for computing the linked list prefix, or in other words, for contracting a linked list to a single node.

Now we show our PRAM algorithm. The important thing to remember is that while the length of the linked list is shrinking, the number of processors does not change. Thus after the list has shrunk to a certain length we can use subroutines B and C while retaining the efficiency of the algorithm.

ListContract2( )

Step 1: For \( i := 1 \) to \( 4 \log(3)n \) do subroutine A; /* Now the contracted list has length \( n/(\log^2 n)^2 \). */

Step 2a: For \( i := 1 \) to \( \log(2)n \) do subroutine B; /* Now the contracted list has length \( n/\log n \). */

2b: Pack the list;

Step 3: Do subroutine C. /* Now the list has been contracted to a single node. */

THEOREM 4. ListContract2() contracts a linked list of length \( n \) to a single node in \( O(n/p + \log n) \) time\(^4\) using \( p \) processors.

Proof. This algorithm has three steps. In step 1 \( 4 \log(3)n \) executions of subroutine A are performed. Two executions of subroutine A reduce the length of the list by at least one-half (by a fraction of \( 1/2 \)). Therefore \( 4 \log(3)n \) executions of subroutine A reduce the list to length \( n/(\log^2 n)^2 \).

Since each execution of subroutine A takes \( O(n/p + \log n/\log(3)n) \) time for a list of length \( n \), the timing for step 1 is \( O(\sum_{i=0}^{4 \log(3)n} ((\frac{1}{2})^i n/p + \log n/\log(3)n)) = O(n/p + \log n) \). Note that since the list is packed after every contraction step, the length of the list forms a geometric series. The factor

\(^4\) \( O(n/p + \log n) = O(n/p + \log p) \).
accounts for this geometric series.

In step 2 \(2 \log^{(2)} n\) executions of subroutine B are performed followed by a packing operation. This step reduces the length of the list to \(n/(\frac{3}{2})^{2 \log^{(2)} n} < n/\log n\). Since the input list to this step has length \(n/(\frac{3}{2})^{2 \log^{(2)} n}\), each execution of subroutine B takes \(O(nG(n)/p(\log^{(2)} n)^2 + G(n))\) time. The \(2 \log^{(2)} n\) executions of subroutine B, i.e., step 2a, take \(O(\sum_{i=1}^{2 \log^{(2)} n} (nG(n)/p(\log^{(2)} n)^2 + G(n))) \leq O(n/p + \log n)\) time. Because the list is not packed in step 2a, each execution of subroutine B is performed on an array of length \(n/(\frac{3}{2})^{2 \log^{(2)} n}\), while the list stored in the array is shrinking. After step 2a, step 2b packs the list. Step 2b takes \(O(n/p + \log n/\log^{(3)} n)\) time.

After step 2 the list has been contracted to length \(n/\log n\). Step 3 takes \(O(n/p + \log n)\) time because the length of the list is \(n/\log n\).

We can also exploit the techniques presented here on the EREW model to achieve time complexity \(O(n \log^{(k)} n/p + k \log n)\), where \(k\) is an arbitrarily large positive integer. In particular, when \(k = G(n)\) time complexity \(O(n/p + G(n) \log n)\) can be achieved [5].

5. Conclusions

The linked list prefix problem is a fundamental problem which deserves in-depth study. Presented in this paper are optimal algorithms for the linked list prefix problem. The DCM algorithm is optimal when \(n \geq p^2 \log p\). The PRAM algorithm is optimal when \(n \geq p \log p\). While the problem of achieving optimal timing on the DCM for \(n = o(p^2 \log p)\) is still open, it seems that our vector balancing techniques could hardly be exploited further to improve the condition \(n \geq p^2 \log p\). On the PRAM model, \(\Omega(n/p + \log n)\) is the lower bound for general situations. However, it is not known whether a sublogarithmic PRAM algorithm exists for certain associative operations. Note that Reif first showed that a sublogarithmic algorithm does exist for the partial sum problem [12], but no similar result is known for the linked list partial sum problem.

Acknowledgments

Many thanks to Professor John Reif for encouraging me to publish my research results. The help provided by Amr Farmy in preparing this document is also appreciated.

References


YIJIE HAN was born in Hangzhou, China, in 1959. He received a B.S. in engineering from the University of Science and Technology of China, Anhui, in 1982, an M.A. from Duke University in 1984, and a Ph.D. from Duke University in 1987. He joined the Department of Computer Science at the University of Kentucky as an assistant professor in 1987. He has discovered efficient parallel algorithms for the linked list prefix problem and the connected component problem. His current interests lie in the areas of parallel algorithms and VLSI algorithms.
Fig. 1. Pair-off.

\[
\begin{array}{c}
\text{Number of data items} \\
5 & 11 & 4 & 2 & 7 & 3 & 13 & 3 \\
\text{Average} \\
6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
\text{Label} \\
-1 & 5 & -2 & -4 & 1 & -3 & 7 & -3 \\
\end{array}
\]

Grouping

\[
\begin{array}{c}
\text{Label} \\
\mathbf{v}_0 & \mathbf{v}_2 & \mathbf{v}_3 & \mathbf{v}_5 & \mathbf{v}_7 & \mathbf{v}_1 & \mathbf{v}_4 & \mathbf{v}_6 \\
-1 & -3 & -7 & -10 & -13 & 5 & 6 & 13 \\
\end{array}
\]

Merging

\[
\begin{array}{c}
\text{Label} \\
-1 & -3 & 5 & 6 & -7 & -10 & -13 & 13 \\
\end{array}
\]

Fig. 2. An example of execution of procedure BALANCE.
Component:

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
2 & 3 & 2 & 5 & 2 & 4 & 4 & 2 & 6 \\
1 & 2 & 1 & 6 & 3 & 1 & 3 & 1 & 5 \\
\end{array}
\]

(a). Construct primitive chains.

\[
V
\]

V\text{\_1}:

V\text{\_2}:

(b). Construct chains by chaining primitive chains.

Components:

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
2 & 3 & 2 & 5 & 2 & 4 & 4 & 2 & 6 \\
1 & 2 & 1 & 6 & 3 & 1 & 3 & 1 & 5 \\
\end{array}
\]

(c). Exchange elements in \(V\_1\) with those in \(V\_2\) for every other primitive chain resulting in balanced vectors.

Fig. 3. An example of balancing two vectors.

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
x\_0 & x\_2 & x\_4 & x\_1 & x\_5 & x\_3 & x\_6 \\
\hline
3 & 5 & 4 & 1 & 6 & 2 & \text{nil} \\
\end{array}
\]

Fig. 4. A linked list.
Fig. 5. The intuitive observation of bisecting.

Fig. 6. Identifying turn points of a linked list.