Chapter 7

List Ranking and Pointer Jumping Algorithms

7.1 Introduction

In Chapters 4 and 5, we covered parallel algorithms on arrays via reduce and parallel-prefix. The advantage of arrays is that any thread can access any element in the array in constant time. In many applications, we have a linked list of threads such that each thread has some data item. The linked list does not provide us *random access* to its elements. Can we still do operations such as computing the maximum in $O(\log n)$ time where the length of the linked list is n? In this chapter, we introduce the technique of *pointer jumping* that can be applied to linked lists with significant reduction in parallel time complexity.

This chapter is organized as follows. Section 7.2 describe the pointer-jumping algorithm. Section 7.3 gives multiple applications of the pointer jumping technique. Section 7.4 gives the LLP algorithm for pointer jumping. We also consider the case when we have a circular linked list. For the circular linked list, there is no notion of list ranking. For a circular linked list, we are interested in coloring the nodes with at most three colors. Section 7.5 discuss a parallel algorithm that takes $O(\log^* n)$ parallel time.

7.2 Pointer Jumping

Suppose that we have a linked list implemented as follows. Each node i in the list has a field $next[i]$ that returns the next node in the linked list. If the element i is the last node in the linked list, then $next[i]$ returns -1. Our goal is to return an array R that returns the distance of any element from the last node in the linked list. In the Pointer-Jumping, we first copy the *next* pointers in an array Q and then use a method called *pointer jumping* to reduce the length of the longest path in the link structure. The pointer jumping is accomplished by setting $Q[i]$ to $Q[Q[i]]$ for every node i. The algorithm assumes that all nodes execute each iteration of the while loop in a synchronous manner. An execution of the algorithm is shown in Fig. 7.1.

The algorithm satisfies the following invariants. The variable $Q[i]$ always points to a node that is reachable from node i (and to -1 if no node is reachable from i). Initially, $Q[i]$ points to the next node in the linked list except for the tail node and thus satisfies the invariant. The variable $R[i]$ is equal to the length of the path from i to $Q[i]$ in the linked list. Initially, $R[i]$ equals 1 for all nodes except the tail node and therefore satisfies the invariant. In any iteration of the while loop, we consider a node i such that $Q[i]$ and $Q[Q[i]]$ are not null (i.e., if $Q[i]$ is viewed as the parent of node i, then node i has a grandparent). By setting $Q[i]$ to $Q[Q[i]]$, we maintain the invariant on Q. Furthermore, by setting $R[i]$ to $R[i] + R[Q[i]]$, we maintain the invariant on R because the length of the path from i to $Q[Q[i]]$ is the sum of the lengths of the paths from i to $Q[i]$ and from $Q[i]$ to $Q[Q[i]]$.

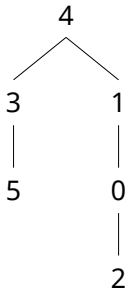
Algorithm Pointer-Jumping: Finding the distance from the tail of the linked list.

```
1  P_i::
2  input: next[i]: pointer to the next node in the linked list;
3  var: Q[i]: pointer to a node;
4       R[i]: int; // distance to the tail of the linked list
5  forall i in parallel do:
6       Q[i] := next[i];
7  forall i in parallel do:
8       if (Q[i] == -1) then R[i] := 0 else R[i] := 1;
9  forall i in parallel do:
10      while (Q[i] ≠ -1) and (Q[Q[i]] ≠ -1) do
11          // evaluate the second conjunct only if Q[i] ≠ -1
12          R[i] := R[i] + R[Q[i]];
13          Q[i] := Q[Q[i]];
```

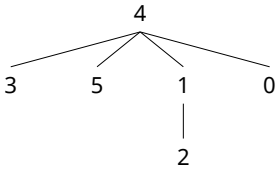
```
i:      0  1  2  3   4   5
next:   3  4  5  1  -1   0

R:      1  1  1  1   0   1
Q:      3  4  5  1  -1   0
links: 2 --> 5 --> 0  --> 3 --> 1 --> 4

iteration 1:
i:      0  1  2  3   4   5
R:      2  1  2  2   0   2
Q:      1  4  0  4  -1   3
links:     2 --> 0 --> 1 --> 4       and 5 --> 3 --> 4

iteration 2:
i:      0  1  2  3   4   5
R:      3  1  4  2   0   4
Q:      4  4  1  4  -1   4
links:     2 --> 1 --> 4      0 --> 4  5 --> 4  3 --> 4

iteration 3:
i:      0  1  2  3   4   5
R:      3  1  5  2   0   4
Q:      4  4  4  4  -1   4
links:     2 --> 4    1 --> 4   0 --> 4  5 --> 4  3 --> 4
```

Figure 7.1: Execution of Pointer Jumping Algorithm

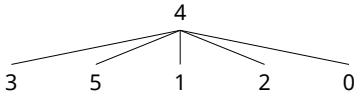
(a)



(b)



(c)



(d)

Figure 7.2: Pointer Jumping (a) Initial Tree (b) Tree after first iteration (c) Tree after second iteration (d) Star tree after the third iteration

When there is no node i such that it has a grandparent, we have that every node except the tail node is pointing to the tail node (a reachable node that is pointing to null). From the invariant on R, we get that $R[i]$ equals the length of the path from i to the tail node.

This algorithm takes $O(\log n)$ time where n is the size of the linked list. It does $O(n \log n)$ work and is therefore not work-optimal. It is possible to reduce the work to $O(n)$ using list contraction. We will not discuss those algorithms and refer the interested reader to [JáJ92].

Although the algorithm was given for the linked list, it is easily verified that the idea of *pointer jumping* is applicable to any rooted tree where a node points to its parent in the tree. In case of a linked list, we had a degenerate tree. The algorithm return the distance of any node from the root.

Let us also verify that we need only CREW PRAM for implementation of this algorithm. In any time step a node i writes only to its own $R[i]$ and $Q[i]$. Hence, CREW PRAM is sufficient to implement the pointer jumping algorithm.

7.3 Applications of Pointer Jumping

We now show that the pointer jumping technique can be applied to many problems.

- *Returning the maximum value in a linked list*: Let us consider the problem mentioned in the introduction of this Chapter. Suppose that we have a linked list of numbers and we would like to compute the maximum of these numbers. Each node i has a natural number $num[i]$ stored at i.

Algorithm Max-Pointer-Jumping: Finding the Maximum in a linked list.

```
1  P_i::
2  input: next[i]: pointer to the next node in the linked list;
3  var: Q[i]: pointer to a node;
4       R[i]: int; // distance to the tail of the linked list
5  forall i in parallel do:
6       Q[i] := next[i];
7       R[i] = num[i];
8  forall i in parallel do:
9       while (Q[i] ≠ −1) and (Q[Q[i]] ≠ −1) do
10           // evaluate the second conjunct only if Q[i] ≠ −1
11           R[i] := max(R[i], R[Q[i]]);
12           Q[i] := Q[Q[i]];
```

The head node now has the maximum and can write that value on a shared variable.

- *Converting a linked list into an array*: In our original problem, we had computed the distance of any node from the tail of the linked list. Let us suppose that we are given a linked list and would like to convert into an array. If the space for the array is allocated, then every thread can compute the index in which the entry in the linked list needs to be written. Observe that once we have an array, we can use parallel-prefix to solve many problems.

  This technique returns the array in the reversed order. We leave it to the reader to compute the array that has values stored in the same order as the linked list.

7.4 LLP Algorithm for List Ranking

In this section we show how List Ranking algorithm can be viewed as LLP Algorithm. We state the problem as follows. We are given a tree rooted at vertex r as input. Each node i other than the root points to another node. The root node r points to null. Our goal is to determine the distance of every node from the root. We view this as the search problem for an integer valued vector G that satisfies a certain feasibility predicate. There is one component in G for every node other than the root. $G[i].dist$ is initialized to 1 as no node can have distance less than 1 from the root. However, this vector G may not be feasible. Associated with every i, we also keep $G[i].next$ which indicates a node that can be reached in $G[i].dist$ hops. Initially, $G[i].next$ is simply the parent of i in the tree. We define $B(G)$ as $\forall i : G[i].next = r$. Thus, any node j is forbidden in G if $G[j].next \neq r$. To advance G, we set $G[j].dist$ to $G[j].dist + G[G[j].next].dist$ and $G[j].next$ to $G[G[j].next].next$. The resulting algorithm LLP-ListRank returns only when for every j, $G[j].next$ equals r. The execution time of the algorithm is crucially dependent on the order in which the forbidden vertices are advanced. If all vertices that are forbidden are advanced in parallel, then the algorithm takes $O(\log n)$ time in the worst case.

Algorithm LLP-ListRank: A Parallel LLP Algorithm for List Ranking

```
1 input: parent: array[1..n] of int
2 var: G: array[1..n − 1] of {dist, next};
3 P_j: Code for thread j
4 init: G[j].dist := 1; G[j].next := parent[j]
5 forbidden: G[j].next ≠ r
6       advance: G[j].dist := G[j].dist + G[G[j].next].dist;
7                G[j].next := G[G[j].next].next;
```

7.5 Vertex Coloring in a Ring

Suppose that we have a circular linked list of n nodes. We would like to color the nodes in the linked list so that adjacent vertices do not have the same color. It is desirable to use as few colors as possible. If there are odd number of nodes in the circle with 3 or more nodes, then we would need three colors. By sequentially traversing the nodes in the cycle, this task can be accomplished in $O(n)$ time. How can we do this in parallel? We first show that given any valid coloring with greater than 6 colors, the nodes can be assigned another set of colors such that the number of colors in the new set is significantly reduced. Assume that initially all nodes have valid coloring. For example, each node could initially have different color by simply using its unique identifier. The following parallel step accomplishes the reduction in the number of colors.

Algorithm Par-VertexColoring: A Parallel Algorithm for Vertex Coloring in a Ring

```
1 input: n nodes arranged in a cycle such that next[i] gives the next node for node i
2       A coloring c of vertices such that for all i: c[i] differs form c[next[i]]
3 output: Another valid coloring d with fewer colors
4 forall nodes i in parallel do
5       let k be the least significant bit such that c[i] and c[next[i]] differ
6       d[i] := 2 * k+ the k^th least significant bit in c[i]
```

Consider a circular linked list $(12, 9, 13, 1, 3)$ where the node with 3 points back to the node with 12. The execution of the algorithm on this set of nodes is shown below.

node	c	k	d
12	1100	0	0
9	1001	2	4
13	1101	2	5
1	0001	1	2
3	0011	0	1

The algorithm gives a valid coloring. Suppose that $d[i] = d[next[i]]$. This means that the corresponding least significant bit k for i and $next[i]$ must be the same. By our algorithm for defining d, this would imply that k^{th} bit for i and $next[i]$ would also be same. This contradicts the definition of k.

The number of color reduces to $O(\log n)$ after every iteration so long as the initial number of colors is greater than 6. Hence in $O(\log^* n)$ iterations, we can reduce the colors to at most 6, where $\log^*(n)$ is the minimum number of times the log function needs to be applied to n to get a number at most 1. Formally, let $\log^{(i)}(n)$ be the log function iterated i times. Then,

$$\log^*(n) = min\{j \mid log^{(j)}(n) \leq 1\}$$

We can now reduce the number of colors to 3 as follows. We first replace all $3's$ in parallel by a number from $0, 1, 2$. Every node can choose a color different from its predecessor and the successor. We then replace all $4's$ and finally all $5's$. These actions correspond to only three parallel steps. The algorithm takes $O(n \log^* n)$ work because n processors take $O(\log^* n)$ time. Therefore the algorithm is not work-optimal but close to work-optimal since $\log^*(n)$ is a very slowly growing function.

7.6 Summary

The following table lists all the algorithms discussed in this chapter.

Problem	**Algorithm**	**Parallel Time**	**Work**
List Ranking	Pointer Jumping	$O(\log n)$	$O(n \log n)$
List Ranking	LLP	$O(\log n)$	$O(n \log n)$
Vertex Coloring	Par-Vertex Coloring	$O(\log^* n)$	$O(n \log^* n)$

7.7 Problems

1. Show that the number of iterations of the list ranking algorithm is $O(\log n)$ if the length of the linked list is n.
2. Given a linked list such that each thread has reference to a node in the linked list of integers, give a parallel algorithm to determine the total number of even numbers in the linked list.
3. Given a linked list such that each thread has reference to a node in the linked list of integers, give a parallel algorithm to determine the total number of times the integer z appears in the linked list.
4. Give a parallel algorithm to compute the suffix sums for all nodes in a linked list.

7.8 Bibliographic Remarks

Reader will find a detailed description of list ranking algorithms in [JáJ92] and [LB00].