EE 382V: Parallel Algorithms **Summer 2017**

Lecture 7: June 16

Lecturer: Vijay Garg *Scribe: Ari Bruck*

7.1 Introduction

During this lecture, we continued our discussion of parallel implementation of algorithms. We were exposed again to the benefits of cascading algorithms as well as Parallel Prefix to reach work optimal algorithms. The topics covered in class include:

- The `merge` Algorithm
- Dijkstra's Guarded Command Language
- The `rank` Operation
- List Ranking (Pointer Jumping)

This set of lecture notes will briefly re-examine the topics covered in this lecture, in the order in which they appeared during class.

7.2 The `merge` Algorithm

When we discussed `merge` in class, we are specifically referring to the merging of two sorted arrays of size=`n` into an array of size=`2*n`.

The professor then asked us to write a sequential algorithm for merging two arrays. A proposed solution was the following:

```
1  i,j,k = 0,0,0;
2  while (i < n and j <n) {
3     if (A[i] < B[j]) { C[k] = A[i]; i++;}
4     else             { C[k] = A[j]; j++;}
5     k++;
6  }
7  while (i<n) {
8     C[k] = A[i];
9     i++; k++;
10 }
11 while (j<n) {
12    C[k] = A[j];
13    j++; k++;
14 }
```

$T(n) = O(n)$

$W(n) = O(n)$

7.2.1 Dijkstra's Guarded Command Language

We were introduced to Dijkstra's Guarded Command Language in relation to the sequential merge algorithm. For example, the if statement inside the while loop in the above code would be re-written to:

```
1  [ // <- guarded statement
2    (A[i] <= B[j]) C[k] := A[i]; i++; k++;
3    (B[j] <= A[i]) C[k] := B[j]; j++; k++;
4  ]
```

If both guards are equivalent and true then the program is non-deterministic. This is not necessarily a bad thing!

7.3 The `scatter` Operation

Here were recapped the terminology used in parallel algorithms. There are

- `reduce`
- `scan`
- `scatter` // this is new

To merge, we pick a random element in array A and see where it goes into C.

$Rank(A[i], C) = (i-1)$ number of elements in A less than $A[i] + rank(A[i], B)$.

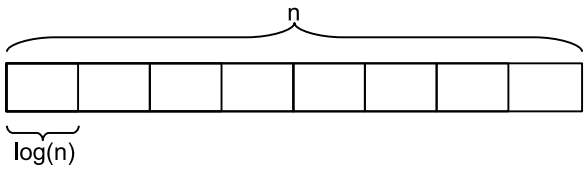
The reason we sort is to do search easy i.e. binary search in $O(\log(n))$ time. **So the overall time is:**

$T(n) = O(1) + O(\log(n))$

This requires concurrent read but no concurrent write so a $CREW$ PRAM. However this is not work optimal since it requires n processors doing $O(\log(n))$ work, or:

$W(n) = O(n * \log(n))$

So how do we improve this? Use cascading!



Think of each element at first block. Call these splitters.

The number of splitters $= O(n/\log(n))$

Merge the splitters: $T(n) = O(n)$

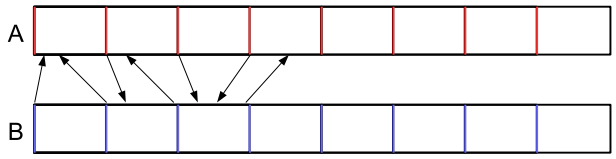


Figure 7.1: The splitters could go anywhere but the key is they can never cross. A crossing line indicates it was smaller in A but no bigger in B). We can order it in $O(\log(n))$.

The algorithm would look like:

1. Break array into splitters and non-splitters
2. Merge all splitters: $O(n/\log(n)) * O\log(n)) = O(n)$ work in $T(n) = O(\log(n))$
3. Apply sequential algorithm to small segment to merge non-splitters in parallel: $T(n) = O(\log(n))$, $W(n) = O(n)$

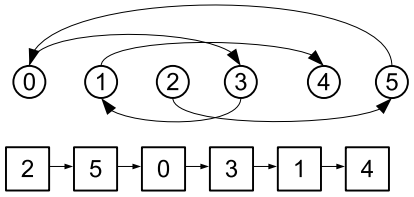
Overall:

$T(n) = O(\log(n))$

$W(n) = O(n)$ so this is work optimal

7.4 List Ranking

Goal: Find the distance from the tail of linked-list.



In finding the parallel implementation, first write down the sequential algorithm to find the work-optimal solution.

Sequential Algorithm:

Iterate through all elements until the tail is reached.

$T(n) = O(n)$

$W(n) = O(n)$

Parallel Algorithm: *We are looking for* $T(n) = \log(n)$

Let's get closer to god/nirvana! We will add another pointer that points to grandparent. This reduces everybody by 2 since grandparent also pointed to their grandparent.

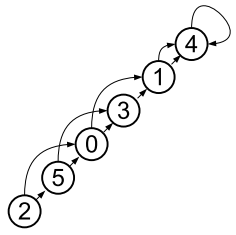


Figure 7.2: This is called **Pointer Jumping**

The reduction by 2 is the log(n) we are looking for: $T(n) = O(\log(n))$

7.4.1 Pointer Jumping Algorithm

Input: Given to you for every `i`, `next[i]`.

```
1  for all i in parallel do:
2      Q[i] := next[i] // make a copy of next[]
3
4  for all i in parallel do:
5      if (Q[i]==-1) then R[i] := 0 // If this is the tail
6      else          then R[i] := 1
7
8  for all i in parallel do:
9      while ((Q[i] != -1) && (Q[Q[i]] != -1)) do // if parent and
           grandparent exists
10         Q[i] := Q[Q[i]]
11         R[i] := R[i] + R[Q[Q[i]]]
12     end while
```

This is for a PRAM, for a GPU there needs to be a barrier so everyone is in sync

7.4.2 Pointer Jumping Example

Let's work through an example that will show us the algorithm.

Array:	2	5	0	3	1	4
`i:`	0	1	2	3	4	5
`Next:`	3	4	5	1	-1	0
`Q:`	3	4	5	1	-1	0
`R:`	1	1	1	1	0	1

Table 7.1: Simulation at time T=0

Array:	2	5	0	3	1	4
`i:`	0	1	2	3	4	5
`Q:`	1	4	0	4	-1	3
`R:`	2	1	2	2	0	2

Table 7.2: Simulation at time T=1

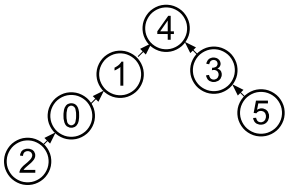


Figure 7.3: Linked list as a tree after 1 iteration

Array:	2	5	0	3	1	4
`i:`	0	1	2	3	4	5
`Q:`	4	4	1	4	-1	4
`R:`	3	1	4	2	0	4

Table 7.3: Simulation at time T=2

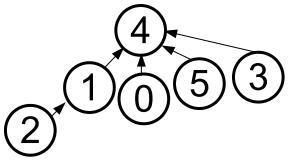


Figure 7.4: Linked list as a tree after 2 iterations - at every iteration we are compressing the height

Array:	2	5	0	3	1	4
`i:`	0	1	2	3	4	5
`Q:`	4	4	4	4	-1	4
`R:`	3	1	5	2	0	4

Table 7.4: Simulation at time T=3

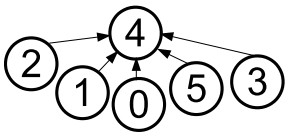


Figure 7.5: Linked list as a tree after 3 iterations

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.

7.4.3 Work-Optimal Pointer-Jumping

Let's see if parallel prefix can help us. In the past implementations, we jumped by even entries:

1. Divide array into even nodes/entries and odd entries
2. Only the even entries do the work (compute parallel prefix recursively)
3. Odd entries in parallel use the even's parallel prefix answers to compute theirs

Q: So how do we do this? In an array it was easy to choose the odd and even entries. This is not so in a linked list.

A: Use a randomizer (coin flip). Each element will flip a coin to decide whether that element will be chosen. To do this effectively we will set some rules for choosing elements:

- two consecutive nodes cannot be chosen
- two dots are not too much apart (in order to do the sequential algorithm efficiently)

Independent Set: In a graph, nodes that do not share an edge are independent.

Everybody flip a coin to get Tail or Heads. Node chooses itself if they get heads and their `Next` is tails.

Probability guarantees that the time until `Next` is soon (probability increases 1/2 every turn). Expected distance between chosen node is $O(1)$.

P(node is not chosen) = 3/4

My Flip	My `Next`'s flip	Chosen
H	H	✗
H	T	✓
T	H	✗
T	T	✗

Table 7.5: Choosing of element in linked list given its and its neighbor's flip Truth table

Recursively move to next level to then reduce to 1. The list is shrunken by constant fraction so therefore in expected $O(\log(n))$ iterations, the list is shrunk to 1.

Overall:

$T(n) = O(\log(n))$

$W(n) = O(n)$