ICS 443: Parallel Algorithms Fall 2017

Lecture 10 — September 25, 2017

Prof. Nodari Sitchinava *Scribe: Isaac DeMello, Brandon Doan, Robert Figgs*

1 Overview

In the last lecture we detailed an algorithm that finds the minimum of an array in O(log(log(n))) time, and with O(n(log(log(n))) work.

In this lecture we define a work-efficient algorithm that finds the minimum in O(log(log(n))) and with O(n) work. We also begin to look at searching algorithms. ....

2 Work Efficient Min

We begin by breaking down the algorithm into parts of size k, where k = O(log(log(n)))

2.1 WE-CRCW-min

In order to break down the size of the array and apply the previously defined CRCW-min algorithm covered in lecture 9. We use the following algorithm:

Algorithm 1 WE-CRCW-min(A, n)

```
k = ⌈log(log(n))⌉
n' = ⌈n/k⌉
B = new array of size n'
for i = 1 to n' in parallel do
    B[i]=A[(i-1)k+1]
    for j = (i − 1)k + 2 to ik in parallel do
        B[i] = min(B[i], A[j])
    end for
end for
return CRCW-min(B, 1, n')
```

2.1.1 WE-CRCW-min Algorithmic Analysis

The first 3 lines of the algorithm can be done in constant time, this means that the runtime of the algorithm is dominated by the for-loop. The dominating for-loop has a runtime of O(log(log(n))) therefore the overall time of this algorithm is $T_\infty(n) = O(\log(\log(n)))$

Work Analysis : Work of this algorithm can be shown as follows:

$\sum_{i=1}^{n} \sum_{j=(i-1)k+2}^{ik} O(1) = \Theta(n'k) = \Theta((n/k)k) = \Theta(n)$

3 Searching

We start with a trivial parallel searching algorithm. Note that this algorithm assumes that every element in the array is distinct. This algorithm will also return a value of -1 if the element is not found in the array.

Algorithm 2 parallel-search(A, n, x)

```
answer = -1
for i = 1 to n in parallel do
  if A[i] == x then
    answer = i
  end if
end for
return answer
```

We can see that because this all runs in parallel, our run time is $T_\infty(n) = O(1)$ and our work is $W(n) = O(n)$

If we plug these values into Brent's scheduling principle we get $T_P(n) = O(\frac{n}{p} + 1)$ This isn't a fantastic run time so lets see if we can make this any better. We'll start by reviewing the sequential algorithms for Binary Search, and Binary Search Trees. Note that these two algorithms assume that the input is sorted.

3.1 Binary Search

Binary search is a sequential algorithm. It can only be used on a pre-sorted array. This algorithm returns the index of the element if it is found between the left and right indices . Otherwise it will return -1.

Algorithm 3 Binary-Search(A, l, r, x)

```
if r < l then
  return -1
else
  mid = ⌊(l+r)/2⌋
  if A[mid] == x then
    return mid
  else if x < A[mid] then
    return Binary-Search(A, l, mid − 1, x)
  else
    return Binary-Search(A, mid + 1, r, x)
  end if
end if
```

The run time for this algorithm is $T(n) = T(\frac{n}{2}) + O(1)$. Using master theorem we get $T(n) = O(\log n)$

3.2 Binary Search Tree

A binary search tree is a special type of binary tree. The only difference is that in binary search tree, the children on the left of the node are all less than the value of the node, and the children on the right side of the node are all greater than the node. To search through the Binary Search Tree we can use the following sequential algorithm. Note that this algorithm returns nil if the element is not found.

Algorithm 4 BST-Search$(root, x)$

```
if root == nil or root.key == x then
  return root
else if x < root.key then
  return BST-Search(root.left, x)
else
  return BST-Search(root.right, x)
end if
```

The run-time of this algorithm depends on how well balanced the binary search tree was before running this algorithm. More specifically this algorithm depends on the depth of this tree. A well balanced binary tree has a depth of $O(\log n)$, meaning that the BST-Search algorithm has a run-time of $O(\log n)$.

4 Parallel Search w/ parallel predecessor algorithm

In the previous parallel searching algorithm, if the element being searched for was not in the array, it would return a -1 value. Instead for our purposes, we can utilize a predecessor algorithm to provide us with the index of the largest element $\leq$ x, in the case where x was not found within the array. This will provide useful later on when implementing a revised parallel search algorithm.

We also went over how a binary search works with an array as well as a binary tree. One benefit of having multiple processors is the ability to increase the size of the tree. For example we are able to store more entries and children nodes based upon the amount of processors at our disposal. For p amount of processors we have p amount of entries per node, and p + 1 children.

4.1 Parallel Predecessor

The following algorithm is a sequential algorithm that is used to find the predecessor within a sorted array.

Algorithm 5 par-predecessor(A, n, x)

```
answer = 0
for i = 1 to n in parallel do
  if A[i] = x then
    return i
  else if A[i] < x && (A[i = 1] > x && i < n) then
    answer = i
  end if
  if A[n] < x then
    answer = n
  end if
  return answer
end for
```

The runtime for par-predecessor is $T(n) = O(1)$, as the algorithm checks all n elements at the same time in parallel and all the operations within are constant. Using the same logic, the work for par-predecessor will be.

$$\sum_{i=1}^{n} O(1)$$

$W(n) = O(n)$

If we were to use a multi branch search tree algorithm similar to the binary search tree algorithm, we would be able to find the predecessor in constant runtime.

4.2 Revised Parallel Search

Now we can implement our revised parallel search on a multi-branched tree with multiple entries per node, while using a par-predecessor algorithm. Before starting we can specify some of the properties of the tree. The node object shall contain a key and child such that, V.key[i] will be the index of a specific entry in a particular node and v.child[i] will be a pointer to a child of the node. The following algorithm takes three arguments, V the node, p the amount of processors, and x the element being searched for. A reminder that the size of key is equal to p, and the amount of children is p + 1 in size.

The algorithm will search one node at each level in constant runtime. Because the amount of children per node in the tree is based upon the amount of processors, the height will correspond

Algorithm 6 Parallel-Search w/predecessor(Node V, p, x)

```
if V == nil then
  return "Not Found"
end if
child_idx = par − predecessor(V.key, p, x)
if (child_idx > 0 && V.key[child_idx] == x) then
  return (V.key[child_idx])
else
  return Parallel-Search(V.child[child_idx], p, x)
end if
```

directly to the value of p. Therefore the height of the tree is $\log_{p+1} n$, as there is p + 1 branches. The run-time of this algorithm, like the binary search tree depends on how well balanced it was before running the algorithm. In the case of a balanced tree, the height is $O(\log_{p+1} n)$ therefore in the case of a balanced tree, the runtime will be $T(n) = O(\log_{p+1} n)$. The worst case scenario for work is where it compares p items per level and traverses down to the lowest level. Then the work for such an algorithm can be described as $\sum_{j=0}^{\log_{p+1}} 1 \sum_{i=1}^{p} 1$. Which equates to $\sum_{j=0}^{\log_{p+1}} p$ or $W(n) = O(p \log_{p+1} n)$