L14: Parallel: Selection

scribe(s): *Nitin Yadav, Xinran Luo*

14.1 Selection

The selection problem is to find a k-th smallest element of A, and can be stated as follows:
Given an array A of n elements and an integer k, such that $1 \leq$ k $\leq$ n, find a $\in$ A, such that
|a' $\in$ A : a' $<$ a| $\leq$ k-1, and
|a' $\in$ A : a' $>$ a| $\geq$ n-k

We describe here a technique known as Accelerating Cascades, to solve the selection problem. The Accelerating Cascades technique, in general, provides a way for taking several parallel algorithms for a given problem and deriving out of them a parallel algorithm, which is more efficient than any of them separately.

14.1.1 Model of Computation

The model of computation used here to solve the selection problem, is a PRAM model with concurrent reads and concurrent writes. That is, the model consists of a number of CPUs sharing a memory unit, and all CPUs can concurrently read and write from that memory unit.

The two quantities for this algorithm that we will measure are:
PTime: This is the maximum time that any one CPU could take for computation.
Work: This is defined as the sum of the number of operations that CPUs perform.

14.1.2 Accelerating Cascades

For devising the fast $O(n)$-work algorithm for the selection problem, we will use two algorithms to be run one after the other:
(1) Algorithm 1 works in $O(\log n)$ iterations. Each iteration takes an instance of the selection problem of size m and reduces it in $O(\log m)$ time and $O(m)$ work to another instance of the selection problem whose size is bounded by a fraction of m (specifically, $3m/4$). The total running time of this algorithm is $O(\log^2 n)$ and its total work is $O(n)$.
(2) Algorithm 2 is a sorting algorithm that runs in $O(\log n)$ time and $O(n \log n)$ work.

The advantage of Algorithm 1 is that it needs only $O(n)$ work, while the advantage of Algorithm 2 is that it requires less time. The benefit of accelerating cascades technique is that it combines these two algorithms into a single algorithm that is both fast and needs only $O(n)$ work. The main idea is to start with Algorithm 1, but, instead of running it to completion, switch to Algorithm 2.

Algorithm 1 Algorithm 1 works in reducing iterations. Input to each iteration is an array B of size m and an integer t, $1 \leq t \leq m$. Given a selection problem to be solved for an array A of size n and an integer k, we begin by passing A as the array ($B = A$), size as n ($m = n$) and integer as k ($t = k$). Algorithm 1 is applied for $O(\log \log n)$ rounds, which reduces the original instance of problem to a size $\leq n/\log n$. An iteration is described as follows:

Algorithm 14.1.1 Selection(B, m, t)

Partition $B \rightarrow B_1, B_2, B_3, .. B_i, .. B_{m/\log m}$
for $i = 1$ **to** $m/\log m$ **pardo**
  $x_i = seq - median(B_i)$
$x = median(x_1, x_2, x_3, ... x_{m/\log m})$
$B \rightarrow \{L, M, R\}$, where
  $L = \{a \in B : a < x\}$
  $M = \{a \in B : a = x\}$
  $R = \{a \in B : a > x\}$
if $|L| > t$
  Do the iteration as $Selection(L, t)$
else if $|L| + |M| < t$
  Do the iteration as $Selection(R, t - |L| - |M|)$
else
  return x

Algorithm 2 Algorithm 2 is a parallel-sorting algorithm for which,
PTime is $O(\log m) = O(\log n)$, and
Work is $O(m \log m) = O(n)$
*$m = n/\log n$

Complexity Analysis We first prove that $r = O(\log \log n)$ rounds are sufficient to bring the size of the problem below $n/\log n$. To get $(3/4)^r n \leq n/\log n$, we need $(4/3)^r \geq \log n$. The smallest value of r for which this holds is $\log_{4/3} \log n$, which is equivalent to $O(\log \log n)$. Therefore, the Algorithm 1 takes $O(\log n \log \log n)$ PTime. Amount of Work is $\sum_{i=0}^{r-1}(3/4)^i n = O(n)$. Algorithm 2 takes $O(\log n)$ PTime and $O(n)$ Work. So in total we take $O(\log n \log \log n)$ PTime and $O(n)$ Work.

14.2 Max

The input is going to be an unsorted set A. $|A|$ = n. We should find the largest element. So the sequential time should be $O(n)$. Also, the cost of PRAM and work are $O(\log \log n)$ and $O(n)$ separately.

14.2.1 Algorithm 1

The PTime is $O(1)$, and work is $O(n^2)$. To find the max number, we need do a lot of comparisons among elements. There are n^2 possible comparisons in this operation. What we gonna do is compare all the $O(n^2)$ pairs in parallel. For example:

A			a_i	...	a_j	

B	1	1	1	...	1	1

Let's compare the a_i and a_j. If a_j smaller than a_i, a_j would be lost. Then change the 1 to 0 correspondingly in array B. Compare all elements like a_i and a_j and change the corresponding 1 to 0 in array B. After all these comparisons, only the elements which larger than the others should be 1 in array. Naturally these elements should be the max elements.

Since it is parallel operation, it is totally possible that after every comparison, they will modify array B concurrently. The hardware should allow concurrently write like this. It doesn't matter the order to write the 0 because they should all should be written. Of course it is more easy for hardware to implement this ans the array B can be seen as bit array.

14.2.2 Algorithm 2

The next algorithm PTime is $O(\log\log n)$, and work is $O(n\ \log\log n)$. What we are going to do is subdivide A into $\sqrt{n}$ equal sized sub-arrays. For example:

$$A1 = \begin{array}{|c|c|c|c|c|c|}\hline & a_1 & a_2 & \dots & a_{\sqrt{n}} & \\ \hline\end{array}$$

$$A2 = \begin{array}{|c|c|c|c|c|c|}\hline & a_{1+\sqrt{n}} & & \dots & a_{2\sqrt{n}} & \\ \hline\end{array}$$

$$\dots$$

$$A\sqrt{n} = \begin{array}{|c|c|c|c|c|c|}\hline & a_{n-\sqrt{n}} & & \dots & a_n & \\ \hline\end{array}$$

Algorithm 14.2.1 Algorithm 2

---

for $i = 1$ **to** $\sqrt{n}$ **do**
  $x_h = Algorithm2 - Max(A_h)$
$X = x_1, \dots, x_{\sqrt{n}}$
return $Algorithm1 - Max(X)$

Algorithm 2 Analysis The big O notation of time is T(n) = T($\sqrt{n}$) + $O(1)$ = $O(\log\log n)$, and the work takes W(n) = $\sqrt{n}$ W($\sqrt{n}$) + $O(n)$ = $O(n\ \log\log n)$. Note that for some t, n = 2^{2^t}, then $\sqrt{n}$ = $\sqrt{2^{2^t}}$ = $2^{2^{t-1}}$ $<-$ doubly geometrically decreasing.

Accelerating Cascades The steps of Accelerating Cascades are as followed:

- 1. Divide A into $n/\log\log n$ blocks $A_1, A_2, \dots, A_{n/\log\log n}$ each of size $\log\log n$.
- 2. Get the max element and return.

Algorithm 14.2.2 Accelerating Cascades

---

for $i = 1$ **to** $\log\log n$ **do**
  $x_h = Linear - Max(A_i)$
$X = x_1, \dots, x_{n/\log\log n}$
return $x = Algorithm2 - Max(X)$

Step 1 takes $O(\log\log n)$ time, and $O(n)$ work.
Step 2 takes $O(\log\log n)$ time, and $(n/\log\log n) * \log\log n = O(n)$ work.