CS260 Homework 2

March 1, 2020

1 Parallel Merging (30pts)

1.1 The subroutine: select the k-th element (10pts)

The algorithm framework:

1. If one array is empty, return the k-th element in the other array.
2. Find the medians of array A (noted as $A[n'] = x$) and B (noted as $B[m'] = y$), respectively.
3. Compare x and y.
4. x and y split the input into four parts:

   1) elements in A before x,

   2) elements in A after x,

   3) elements in B before y, and

   4) elements in B after y

   Based on the values of n, m, n', m' and k, and result of the comparison between x and y, you can exclude one part above which is unlikely to contain the k-th element in A and B. Remove those elements in that part.
5. Recursively apply the algorithm to the rest of the elements.

Let's call the four parts Part 1, 2, 3, and 4, respectively.

Problems:

(a) Using $n' = n/2$ and $m' = m/2$ for finding the medians of the two arrays. WLOG assume $x < y$ (the other case is symmetric). So there are n' elements in A that are smaller than x and $n - n' - 1$ elements in A that are larger than x. Similarly, there

are m' elements in B that are smaller than y and $m - m' - 1$ elements in B that are larger than y.

For x, all elements in Part 1 must be smaller than it (n' of them). All elements in Part 2 and 4 must be larger than it (since $y > x$). Only part of the elements in Part 3 is smaller than it. So there are **at most $n' + m'$ elements smaller than it**.

For y, we know that there are **at least $n' + m'$ elements smaller than it** because all elements in Part 1 and 3 are smaller than it.

- Case 1, $k > n' + m'$. Since there are at most $n' + m'$ elements smaller than x, the k-th element must be larger than x. So we can exclude all elements in smaller than x, which is Part 1 in the algorithm framework, i.e., elements in A before x. We then recurse to find the $(k - n')$-th element in $A[n'..n]$ and $B[0..m]$.
- Case 2, $k \leq n' + m'$. Since there are at least $n' + m'$ elements smaller than y, the k-th element must be smaller than y. So we can exclude all elements in Part 4. We then recurse to find the k-th element in $A[0..n]$ and $B[0..m']$.

(b) (4pts) Every time we recurse, one of the array size shrink by a half. At most we need $O(\log n + \log m) = O(\log(m + n))$ recursive calls, each costing a constant time.

1.2 Parallel merging in $O(\log n)$ depth (20pts)

We are going to merge two sorted arrays of sizes n_a and n_b, let $n = n_a + n_b$.

(a) **Problems:**

1. The cost of the algorithm lies in two parts: m calls to `findKth`, and m sequential merges.

   For the m `findKth` calls, they can be done in parallel. The total work is $O(m \log n)$. The total depth is $O(\log n)$ (if consider binary forking it is $O(\log n + \log m)$) since this is the cost of each `findKth`.

   For the m sequential merges, we can just use the sequential merge algorithm (as in the merge sort), and let all m of these subtasks do in parallel. The total work is $O(n_a + n_b)$ because we still look at each element exactly once. The depth is $O(k) = O(n/m)$ (if consider binary forking it is $O(\log k + \log m)$).

   In all, the work of the algorithm is $O(m \log n + n)$, and the depth of the algorithm is $O(k + \log n)$.

2. For work we want $O(m \log n + n)$ to be $O(n)$. So $m \log n = O(n)$, $m = O(n/\log n)$, which means $k = \Omega(\log n)$.

   For depth, we want $O(k + \log n) \leq O(\log n)$. So $k = O(\log n)$.

In all, choosing $k = \Theta(\log n)$ and $m = \Theta(n/\log n)$, we can get linear work and $O(\log n)$ depth.

3. Plug in the result in Problem 1 to prove it.

(b) **Problems:**

1. $W(n) = m \log n + mW(k)$

   $D(n) = \log n + D(k)$.

2. In every round the problem size decrease from n to $\sqrt{n}$. From homework 1 problem 1(c) we know that it finishes in $O(\log \log n)$ rounds.

3. Plug in $k = \Theta(\sqrt{n})$ we get $D(n) = \log n + D(\sqrt{n})$. In total the cost is (omitting big-O):

$$\begin{aligned} &\log n + \log \sqrt{n} + \log n^{1/4} + \log n^{1/8} + \cdot \\ =& \log n + \frac{1}{2}\log n + \frac{1}{4}\log n + \frac{1}{8}\log n + \cdot \\ =& \log n \end{aligned}$$

2 Parentheses matching (10pts)

Problems:

1. For the stack, since there can only be "("s, we can just record the number of "(" in the stack. Therefore, we can just scan the array from left to right. We maintain a counter t to record the number of "(" in the stack right now. Whenever we see a "'(", we add one to t. If we see a ")", we subtract 1 from t. As long as during the whole process, t stays non-negative and finally t is 0, the string is a valid parentheses matching.

2. For "("s, we first look at the array in parallel and use a 0/1 array f to denote if the i-th element is a "(". Then using a parallel reduce algorithm gives you the total number of "("s in your input. The work is $O(n)$ and depth is $O(\log n)$. Similar idea applies to ")"s.

3. Create an array B with size n. If the i-th element in the input is "(", $B[i] = 1$, otherwise $B[i] = -1$. Run the parallel prefix sum algorithm on B and get C. Finally check in C to see if $C[i] \geq 0$ for all i, and if $C[n] = 0$. If so, the input is valid, otherwise it's not a matching.

   The cost of the algorithm is bounded by the prefix sum algorithm, which has $O(n)$ work and $O(\log n)$ depth.

3 Maximal Independent Set (15pts)

In a graph $G = (V, E)$, an independent set is a set of vertices $V' \in V$, such that $\forall u, v \in V'$, v and u are not neighbors. A maximal independent set (MIS) is an independent set S such that for any $v \in V \backslash S$, $S \cup \{v\}$ is not an independent set. This means that adding any one more node into S will make it not independent. Note that MIS of a graph is not unique.

Sequentially, finding one maximal independent set is easy. First of all, let's assign a random priority to each vertex. This can be generated by using a random permutation, for example, just like in the SCC algorithm we've seen in the class. We process all vertices in the order of their priority, highest the first. We call the current processed vertex the **pivot**. All vertices are initialized as **active**. When processing a pivot v, we check if it's still active. If not, we skip to the next pivot. If so, we add v into S, and mark v and all its neighbors as **inactive**. This is because we chose v in S, so all its neighbors cannot be chosen in S then. We then continue to the next pivot, until all vertices have been processed.

This does not work well in parallel, since it requires $O(|V|)$ rounds to process all vertices. In this problem, we will design a parallel algorithm to find an MIS.

Although directly parallelizing the above algorithm seems hard, we've seen some techniques in class that might be useful. In particular, we will use the idea we learnt in the lecture about deterministic parallelism, and the prefix doubling approach we learnt in the SCC algorithm.

Problems:

(a) Answer the following questions:

  (a) When v's priority is higher (earlier) than all its neighbors.

  (b) The algorithm is round-based. In each round, we check all vertices to see if they are ready in parallel. If a vertex is ready, we invalidate all its neighbors. Finally we pack the rest of vertices and proceed to the next round.

(b) Another idea is to consider using prefix doubling.

  (a) Prefix doubling deals with an iterative algorithm with input size n, where each iteration deals with one input element (the order is usually decided by a random permutation). Prefix doubling requires to process the elements in $O(\log n)$ rounds. In round i, 2^i elements are processed in parallel. If two elements processed in the same round "conflict" with each other, we have to resolve it depending on the algorithm. The conflict generally means that the two elements cannot be processed in the same round, e.g., one blocks the other.

  (b) When two vertices v and u processed in the same round are neighbors, one of them must have higher priority than the other, which means that the later one has to wait for the first one to finish.

  (c) For all vertices with a random priority, the algorithm works in rounds. In round

i, we process 2^i vertices with highest priority in parallel. In each round, every processed vertex, if still valid, will try to invalidate all their neighbors. For two processed vertices, if one is trying to invalidate the other, we need to resolve the conflict.

4 Programming (25 pts)

4.1 Parallel Matrix Multiplication (10 pts)

(a) (Just testing)

(b) Generally, the one which only parallelizes the outmost for-loop should be the fastest. This is because forking and joining are costly than a simple computation. If each task is too small, the overhead can be significant. Also we only have (usually) no more than 100 processors, so to forking out more tasks is not useful. The reason is similar to why we do coarsening - let each parallel task large enough to show good parallelism.

4.2 Parallel Quicksort (15 pts)

(a) Generally they should be different but not a lot. The algorithm and cost are different so the base case should be slightly different.

(b) -

(c) -

(d) -

(e) Usually randomly choosing 3-7 pivots and selecting the median of them can give good performance.

5 Implementing a divide-and-conquer matrix multiplication algorithm (5pts bonus)

This divide-and-conquer algorithm should have better performance than the version with three parallel for-loops because of better locality.