CS260 Homework 2

February 16, 2020

Deadline. The homework is due **Wednesday Feb. 19**. You can submit your write-up to me after class or before that.

Late days. You have five late days to use throughout the quarter. This includes homework, course project proposal and project final report.

Collaboration policy. Discussion on the homework problems is allowed, after you have thought about the problems on your own. It is OK to get inspiration (but not solutions) from books or online resources, again after you have thought about the problems on your own. You must cite your collaborators and resources fully and completely (e.g., *Alice explained to me the definition of work in Problem 1* or *I used Wikipedia site `https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)` about Master theorem for Problem 3*). You must write up your solution independently: close the book and all of your notes, and with no one else helping you when you are writing your final answers.

Write-up. Please do not use hand-writing submissions. Using LaTeX is highly-recommended. For all problems, please explain how you get the answer instead of directly giving the final answer.

Programming Problems. For all programming assignments, you need to submit your source code through iLearn by 23:59 Feb. 19. In the write-up, you need to provide detailed report of the experimental results and analysis. The programming part gives you up to a 5% bonus when you get very good performance or write good "report" to analyze the experiments.

1 Parallel Merging (30pts)

In class we've seen a parallel merge sort algorithm using $O(\log^3 n)$ depth. This is because the merging algorithm itself takes $O(\log^2 n)$ depth. In this problem, we'll design some other merging algorithms with lower depth.

For simplicity, in the following problems you can assume all keys are distinct.

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

Assume you are given two arrays of size n and m, each with keys in sorted order, and an integer $k, 0 \leq k < n+m$. Let x_k denote the k-th smallest element out of the $n+m$ total elements. Describe an algorithm that returns the cut point in each array such that all elements below the cut point are less than or equal to x_k, and all elements above are greater than x_k.

Hint: you can consider the following 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.

Problems:

(a) (6pts) Based on the hint above, describe the algorithm in detail: you need to compliment the details omitted by the above description. For example, which part to exclude in step 4? Why? How to recursively apply the algorithm?

(b) (4pts) Prove your algorithm takes $O(\log(m+n))$ work.

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

To get rid of an extra factor of $O(\log n)$ in the bound, the idea is to use multi-way divide-and-conquer instead of two-way (see an example in Figure 1). Next, you will design two different algorithms for parallel merging.

(a) The first algorithm works as follows.

   1. For two arrays A and B with sizes n_a and n_b, let $k > 0$ be a parameter and $m = \lfloor (n_a + n_b)/k \rfloor$. Assume the output will be stored in array C.

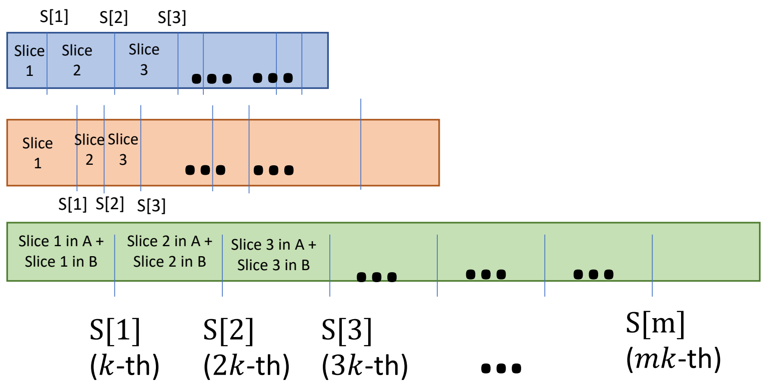


Figure 1: The multi-way merging algorithm.

2. First we need to find the global k-th, $2k$-th, ... $m \cdot k$-th elements in the two sorted array, using the select k-th element algorithm above. Note these elements as $s[1..m]$ and call them pivots.

3. The pivots split each of the two arrays into $m + 1$ slices.

4. Start m tasks in parallel, the i-th task is to merge the i-th slice in A and the i-th slice in B, **sequentially**. Write the results in $C[ik..(i + 1)k - 1]$.

Problems:

1. (4pts) Write down the work and the depth of this algorithm, respectively.
2. (3pts) Choose the value of parameter k to make the algorithm work-efficient (i.e., $O(n)$ work) and bound the depth in $O(\log(n_a + n_b))$.
3. (3pts) Prove that the work and the depth of the algorithm.

(b) The second algorithm works as follows.

1. For two arrays A and B with sizes n_a and n_b, let $k > 0$ be a parameter and $m = \lfloor (n_a + n_b)/k \rfloor$. Assume the output will be stored in array C.

2. First we need to find the global k-th, $2k$-th, ... $m \cdot k$-th elements in the two sorted array, using the select k-th element algorithm above. Note these elements as $s[1..m]$ and call them pivots.

3. The pivots split each of the two arrays into $m + 1$ slices.

4. Start m tasks in parallel, the i-th task is to merge the i-th slice in A and the i-th slice in B, **recursively**. Write the results in $C[ik..(i + 1)k - 1]$.

Problems:

1. (4pts) Write down the recurrence to compute the work and the depth of this algorithm, respectively.
2. (3pts) Prove that if we choose $k = \Theta(\sqrt{n})$, the algorithm finishes in $O(\log\log n)$ rounds.
3. (3pts) Prove that if we choose $k = \Theta(\sqrt{n})$, the depth is $O(\log(n_a + n_b))$.
4. (3pts bonus) Prove that if we choose $k = \Theta(\sqrt{n})$, the work is $O(n)$.

2 Parentheses matching (10pts)

The parentheses matching problem checks if a set of a single kind of parentheses match. For example, "(()(()()))" matches. "((()()))(()()" or "(()))()()" does not.

To check if a string is a valid match, the idea is to scan the array of parentheses from left to right. Every right parenthesis cancels out the closest left parenthesis in front of it, and they match. Then we can maintain a stack to do this. When we see a left parentheses "(", push it in the stack. Whenever a right parenthesis is read, we pop one "(" out from the stack. As long as there are enough "("s in the stack to use for the right parentheses, and at the end none "(" is left in the stack, return true. Otherwise return false.

Problems:

1. (3pts) Give a sequential algorithm to solve the parentheses matching problem. Your algorithm must take $O(n)$ work and $O(1)$ **extra space**.
2. (3pts) Give a parallel algorithm to compute the total number of left parentheses ")" and the right parentheses ")", respectively. Your algorithm must run in $O(n)$ work and $O(\log n)$ depth. You can use $O(n)$ extra space. If these two numbers does not match, we directly know that the string is not a valid parentheses match.
3. (4pts) Give a parallel algorithm to solve the parentheses matching problem. Your algorithm must take $O(n)$ work and $O(\log n)$ depth. You can use $O(n)$ extra space. Prove the cost of your algorithm.

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:

1. We want to borrow the idea of deterministic parallelism for designing an MIS algorithm. In the class about deterministic parallelism, we've seen how to parallelize an iterative algorithm, where the algorithm is a list of tasks performed one after the other. The key idea is to **make the parallel version of the algorithm to have exactly the same effect (and execution) as the sequential version**. In particular, we can consider the following (you can plug in the random permutation algorithm to help you understand the general idea):

   - For a task i, is there any unfinished task j before task i that could block task i? If not, or if task j has finished, we say task i is **ready**. We can then start performing task i immediately, instead of waiting until task $i-1$ finish.
   - When multiple tasks are ready at the same time, we can execute them all in parallel.

   Answer the following questions:

   (a) (3pts) First of all, consider a vertex v in the graph with priority p_v, what is the condition that makes v ready such that v can be used as the pivot right away? Hint: consider v's priority and all its neighbors' priority.

   (b) (4pts) Describe a parallel algorithm based on the idea in Problem (a). You do not need to prove the cost of your algorithm.

2. Another idea is to consider using prefix doubling.

   (a) (2pts) Describe the basic idea of prefix doubling.

   (b) (4pts) In prefix doubling, we may need to process multiple pivots in parallel at the same time. Is there any potential conflicts? Please describe the scenario of conflicts.

   (c) (2pts) Assume that we can resolve the conflicts as you described in Problem (b) efficiently. Describe your algorithm for MIS using prefix doubling.

Remark: In a round of prefix doubling, we may be processing 2^i pivots at the same time. Actually, when two pivots conflict (in the scenario as you described in Problem 2(b)), we just postpone the execution of the pivot with lower priority. All those vertices that had conflicts with another vertex with a higher priority will be buffered, and we try to restart processing them in parallel again. This is similar to what we do in the parallel random permutation algorithm. There can still be conflicts, so we again buffer those with lower priority and restart, until all of them finish. Using a random permutation to generate the priority of each node, the number of sub-steps in each round is $O(\log n)$ w.h.p. (You don't need to prove this.)

Actually, both parallel algorithms above can compute an MIS within $O(\log^2 n)$ depth w.h.p. You don't need to prove this.

4 Programming (25 pts)

You can use the server provided by the department, or any other multicore machine that you have access to (better with more than 10 cores). Instructions of logging into the ti-05 server, sample code, and suggestions about write-up are available in Homework 1.

4.1 Parallel Matrix Multiplication (10 pts)

Implement the parallel algorithm you learnt in class for matrix multiplication using parallel reduce.

(a) **(4pts) Test your algorithm and evaluate your algorithm.**

(b) (6pts) For the version using parallel reduce algorithm, in class we saw the pseudocode as follows:

```
par_for i = 1 to N do
  par_for j = 1 to N do
    par_for k = 1 to N do
      temp[k] = X[i][k] * Y[k][j];
    Z[i][j] = parallel_reduce(temp, N);
```

However, the performance might not be satisfying. Again, this is because of the parallel tasks are too fine-grained, and the overhead of forking and joining is large. Let's step back to use a simpler version.

```
//version 1
par_for i = 1 to N do
  for k = 1 to N do
    par_for j = 1 to N do
       Z[i][j] += X[i][k] * Y[k][j];
```

Note that we are now moving the for-loop on k to the middle. You might have seen this trick in your previous courses: putting the loop of k in the middle gives better cache locality to the algorithm, such that it makes performance better.

However, we'll then further "remove" parallelism from the above algorithm, in the following two ways:

```
//version 2
for i = 1 to N do
  for k = 1 to N do
    par_for j = 1 to N do
       Z[i][j] += X[i][k] * Y[k][j];
```

, and

```
//version 3
par_for i = 1 to N do
  for k = 1 to N do
    for j = 1 to N do
       Z[i][j] += X[i][k] * Y[k][j];
```

Test all the three versions above and compare the running time. Which one performs best? Does more parallel for-loops mean better performance in practice? Try to explain the result you get.

4.2 Parallel Quicksort (15 pts)

Implement a parallel quicksort algorithm. There is sample code provided in the repository, but again, that won't give you the best performance. You need to add some optimizations to help your code get better performance. Consider the following:

(a) (2pts) Coarsening. Use the similar idea as shown in Homework 1. Try to find the best threshold to determine a sequential or a parallel run. It this threshold the same as what you got from last homework (the reduce/scan algorithms)? why?

(b) (1pts) You can call STL sorting algorithm or other existing sorting algorithms to work on the base cases. Since STL is highly-optimized, it's likely to give better performance than your sequential version.

(c) (1pts) Using the parallel partitioning algorithm that we've seen in the class instead of two filters.

(d) (6pts) Test your code on different input distributions, in particular:

- Inputs with all distinct keys.
- Inputs with heavy duplicates, e.g., a Zipfian distribution (`https://en.wikipedia.org/wiki/Zipf%27s_law`), an exponential distribution, or a uniform distribution with key range in $0, n/k$ for different values of k. Adjust the frequency of duplicates, and test your algorithm on these different distributions.

For input with heavy duplicates, a simple optimization to improve performance is to split your input into three groups instead of two groups. For a given pivot p, you can split all the input elements into (1) those larger than p, (2) those smaller than p and (3) those equal to p. Then you only need to recurse on the first two groups, and output the element in the third group directly. This saves much work. Of course for inputs with light duplicates or even all distinct keys, this does not help much. Experiment on different levels of key duplicates with your optimization, show and analyze your results.

(e) (5pts) Choosing an arbitrary key as the pivot gives good theoretical bound using randomization, but in practice things are not always ideal. You can improve the performance of your code by choosing the pivot more carefully. For example, you can pick up x random elements in the original array, find the median of them, and use the median as the pivot. This helps to split your input in a more balanced way. x is usually selected as a small constant. Try to find the best x with experiments. You don't need to find the "exact" best value of x since that may cause **overfitting**. The setting, environment, and input all affects the best choice of your parameter. Just a very rough analysis would be good.

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

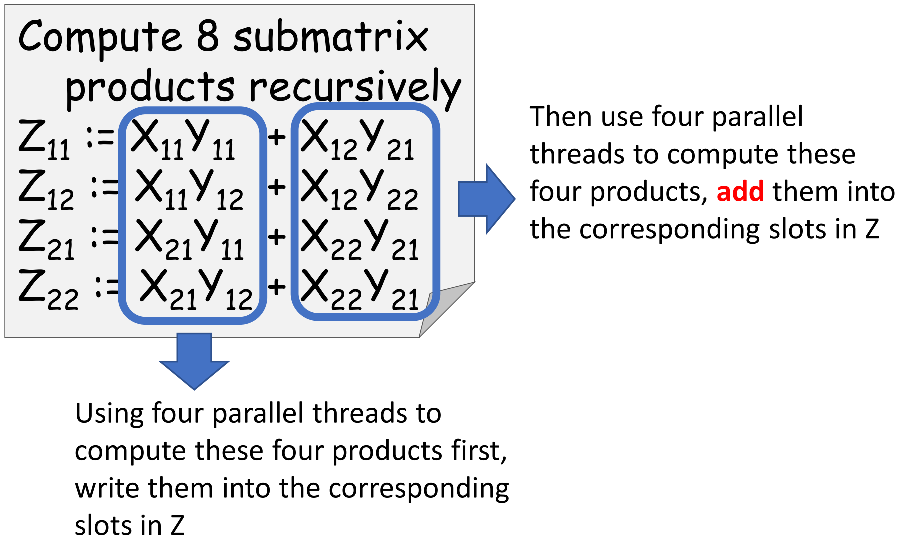
If you want to implement the divide-and-conquer matrix multiplication algorithm, you might find it a bit tricky since you may get write conflicts when taking the sum. Of course you can allocate new space to hold the intermediate results, but that is unlikely to give you good performance (or you may need to make it very complicate to get good performance).

To do this, a solution is to make the 8-way divide and conquer to be a 4-way divide-and-conquer. See Figure 2 (a) for an illustration.

A sample pseudocode is shown in Figure 2 (b).

Implementing this algorithm and evaluating the performance will give you a 5 bonus points. You should expect this algorithm to be faster than the version of three for-loops.

Please remember to check the correctness of your algorithm by comparing it with a sequential implementation under small inputs. Another easy way to check the correctness is like this: let $X[i][j] = i + j$, $Y[i][j] = i - j$ (or any deterministic function). After compute the output



(a)

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    cilk_spawn Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    cilk_spawn Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    cilk_spawn Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
    cilk_sync;
    cilk_spawn Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    cilk_spawn Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    cilk_spawn Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    cilk_sync;
} }
```

(b)

Figure 2: 4-way divide-and-conquer matrix multiplication.

Z, you check the sum of all values in Z. Compare the sequential version with your parallel version. In this way you only need to compare one value instead of the whole matrix.