In this homework you will design some efficient parallel algorithms using techniques you have learned, especially, divide and conquer, scan, etc.

**Problem 2-1. Closest Pair Problem** [55 points]

You have probably seen how to solve this problem in CSE 241; now you will design a parallel algorithm for this problem. You are given a set $S$ of $n$ points, and you have to find the distance between the pair of points that are closest to each other.

(a) [10 points]

You are given a set of points along a line. Design and analyze a parallel algorithm to find the distance between the closest pair of points using $O(n \log n)$ work and poly-logarithmic span. The smaller the span of your algorithm, the better.

For the rest of this problem, we will try to solve the problem in 2 dimensions. The most basic algorithm would involve just computing the distances between all $\binom{n}{2}$ pairs and find the minimum. We want to design a more efficient algorithm.

**Definition 1 (The Closest Pair Problem)** Given a sequence of points $S = \langle p_1, \ldots, p_n \rangle$, where $p_i = (x_i, y_i)$, the closest pair problem is to report the distance between the closest pair of points, that is,

$$\min \{ d(p_i, p_j) : p_i, p_j \in S \land i \neq j \},$$

where $d(\cdot, \cdot)$ is the Euclidean distance measure.

Remember that the distance $d(p_i, p_j)$ between $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$ is given by

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

(b) **Divide and Conquer Algorithm Analysis** [20 points]

You will devise a divide and conquer algorithm; some of you may have already done it in CSE 241. It involves the following steps: (1) **divide** that splits the input sequence in approximately half (according to some criteria); (2) **recurse** that recursively solves the two halves of the problem; (3) **combine** that finds the solution to your input, presumably with help of the recursive solutions you have found.
The work of your divide-and-conquer steps must satisfy the recurrence

\[ W_{\text{closestPair}}(n) = 2W_{\text{closestPair}}(n/2) + W_{\text{divide}}(n) + W_{\text{combine}}(n), \quad (1) \]

where \( W_{\text{combine}}(n) \) denotes the work needed in the combine step and \( W_{\text{divide}}(n) \) is the work needed in the divide step. Now try to implement the divide step and the combine step in \( O(n) \) work so as to get a total work of \( O(n \log n) \). Hint: You will want to pre-sort the points by both \( x \) and \( y \) coordinates as step 0. Then do a divide an conquer algorithm on a function that takes two arrays as input, one contains all the points sorted by \( x \)-coordinate and one contains all points sorted by \( y \)-coordinate. You do not want to sort as part of your divide step — otherwise, your work will be too high.

Note that, in your combine step, you must check to see if the points on the opposite sides of your partition are closer than the closest pairs you have found in either partition. In order to do this, you can not check all pairs, since that would be \( O(n^2) \) time. Therefore, you have to design your combine step carefully to make sure that you only do \( O(n) \) work. Note that you have probably already studied and implemented this algorithm in CSE 241.

You may want to use the following lemma to analyze your combine step’s running time:

**Lemma 1** Let \( \delta > 0 \). There is a constant \( C_P \), independent of \( \delta \), such that if \( R \) is a \( \delta \times \delta \) square and the closest pair distance in \( R \) is at least \( \delta \), then \( R \) contains at most \( C_P \) points.

Another way of thinking about this lemma is the following: How many circles of radius \( r/2 \) can you pack into an \( r \times r \) square such that none of the circles overlap?

(c) **Parallelize your Divide and Conquer Analysis** [25 points]

Parallelize your recursive algorithm. The interesting part is the parallelization of the combine step. For full credit, we expect your solution to have \( O(n \log n) \) work and \( O(\log^3 n) \) (or \( O(\log^4 n) \)) span,\(^1\) where \( n \) is the number of input points. We remark that the \( O(n \log n) \) work bound is known to be optimal. Significant partial credit will be given to \( O(n \log^2 n) \)-work \( O(\log^4 n) \)-span solutions.

**Problem 2-2. Computing the Skyline** [30 points]

You are given an array \( B \) of building, where a building \( b_i \) is represented as the triple \((\ell_i, h_i, r_i)\) such that \( \ell_i \) is the left \( x \)-coordinate and \( r_i \) is the right \( x \)-coordinate of the base of the building, and \( h_i \) is the height of the building. All the bottoms of the buildings are at 0 on the \( y \)-coordinate (that is,\(^1\)Don’t worry as long as your span is logarithmic

\[ ^1 \]
contrary to popular belief, the city is flat). You want to draw the skyline that shows the outlines of
the top of buildings that are not obscured by other buildings.

You should return an array with tuples such that \((x, y)\) such that:

1. \(x_i < x_{i+1}\) for all \(i\): That is the array is sorted by \(x\) coordinate.
2. \(y_i = \max \{h : (\ell, h, r) \in B \text{ and } y_i \in [\ell, r]\}\): That is \(y_i\) represents the highest building that
   spans \(x_i\).
3. \(y_i \neq y_{i+1}\): You don’t report redundant heights.

In words, this problem produces a sequence of “instructions” \(\langle (x_1, y_1), \ldots, (x_m, y_m) \rangle\) that reports
that the height at \(x_i\) is \(y_i\) and the skyline will remain at that height until further instruction. This
naturally describes a piecewise constant function.

Give a divide and conquer solution to solving this problem. Your solution should have work
\(O(n \log n)\) and logarithmic span. The smaller the span, the better.

**Problem 2-3. Parse Text in Parallel [15 points]**

One of the most common tasks in your career as a computer scientist is that of parsing text files.
The basic idea of parsing is to take a string and break it up into parts; exactly how the string is split
up depends on the “grammar” of what’re parsing. Here, you will develop a simple, but quite useful,
forms of parsing, called **tokenizing**, which breaks up a string into a sequence of tokens separated
by one or more delimiters (such as white space for sentences or commas for csv files). A **token** is a
maximal nonempty substring of the input string consisting of no delimiter. By maximal, we mean
that it cannot be extended on either side without including delimiter. For example, if you tokenize,

"it will rain today"

you should get

\'<"it", "will", "rain", "today"'>

Given a string of length \(m\) with \(n\) tokens, give an efficient algorithm to tokenize the string.