CSE341T/CSE549T

Homework 2

Assigned: 09/22/2014 *Due Date: 10/06/2014*

In this homework you will design some efficient parallel algorithms using techniques you have learned, especially, divide and conquer, scan, etc. Your programming assignment consists of implementing a parallel Closest Pairs Algorithm.

Submission

This assignment requires that you submit both code and written answers.

Your written answers must be printed out and submitted in class on the day the assignment is due.

The coding portion of the assignment is distributed in a number of files in our `svn` repository. Perform an `svn update` in the directory where you have checked out your repository and you will see the directory `hw2`. Inside this directory, you will see a `code` directory with the following files in it:

- `StudentSolution.cpp`: **the only file you should modify for this assignment**
- `Point.h`, `PointArray.cpp`: helper classes that store points and arrays of points.
- `Makefile`, `PairResult.h`, `PairResult.cpp`, `PointSupport.cpp`: code you should be glad we wrote for you
- `input1`, `input2`, `input3`: sample input files. The first line of the file indicates the number of lines in the file. For example, if the first line is "5", a correctly formatted file will have 5 more lines, each with a point.

Your assignment is to complete the three methods that are currently empty in `ClosestPair.cpp`. This is the only file you are allowed to modify. Your code must compile cleanly. Make sure that the three methods you are to write maintain their original names and signatures. Your code will be automatically graded; if your method has the wrong name or signature, you will not receive credit for your work.

Take some time to become familiar with the two classes, `Point` and `PointArray`. They contain helper methods that will make your life much easier.

`Point` contains the following methods:

- `Point()`: a constructor with `x` and `y` initialized to 0.0
- `Point(double xVal, double yVal)`: a constructor for which you set the initial `x` and `y` values

- `double Point(Point *other)`: returns the Euclidian distance from this point to the other point
- `int compareX|Y(const void *a, const void *b)`: Static functions that compare two Points by comparing their x/y values, used below by the `sortByX` and `sortByY` methods. When comparing X values, a "tie" is broken by the Y values, and vice versa.
- `int getX(), int getY(), print()`: some obvious ones

`PointArray` contains the following methods:

- `PointArray(int numPoints)`: a constructor that allocates the correct number of points up front
- `PointArray()`: destructor explicitly deletes the array that the constructor calloced
- `PointArray *copy()`: returns an exact clone (shallow copy) of the array.
- `append(Point *newPoint)`: adds a new point to the array at the end. You should use this function only in your sequential solution, not in your parallel solution.
- `append(int x, int y)`: Adds a new point `(x, y)` to the end of the array. Again, this should only be used in the sequential solution, if needed.
- `insert(int i, Point *newPoint)`: adds a new point at position i. You may use this function in your parallel solution, since this allows you to populate an array in parallel.
- `insert(int i, int x, int y)`: Adds New point `(x, y)` at position i. Again, this us useful in your parallel solution.
- `Point *getPoint(int i)`: Returns a pointer to the point at index `i`
- `int size()`: Returns the max size of the array
- `void sortByX|Y()`: Sorts your array by X/Y values in place by calling standard library implementation of quicksort.
- `print()`: prints out the array
- `print2()`: prints out the array with point indices

Problem 2-1. Closest Pair Problem[65 points]

Suppose we are given a set of n two-dimensional points. How fast can we find the distance between the closest pair of points? The most basic algorithm has to examine all $\binom{n}{2}$ pairs and find the minimum, but, perhaps, we don't have to look at every pair of points because these are points in a 2-d plane, which has a lot of structure.

For inspiration, the problem in one dimension can be solved in $O(n \log n)$ work and $O(\log n)$ span via sorting (Do you see how?). Alternatively, there is an equally efficient divide-and-conquer algorithm, which works roughly as follows. On input a sequence S of n numbers,

1. Find a median[1] m and split S into $S_1 = \{x \in S : x < m\}$ and $S_2 = \{x :\in S : x \geq m\}$.

[1]This can be computed directly in $O(n)$ work and $O(\log^2 n)$ span using divide and conquer; however, we won't sketch this solution here. Instead, we will presort the sequence S in $O(n \log n)$ work and $O(\log^2 n)$ span, so median finding takes constant work and span.

2. Recursively find the closest pair distance δ_i on S_i, for $i = 1, 2$; and
3. Derive the closest pair distance on S by taking the smaller of $\min(\delta_1, \delta_2)$ and the distance between the rightmost point of S_1 and the leftmost point of S_2.

In this problem, you will design and analyze three solutions to the closest pair problem: a brute force (quadratic) solution, a work-optimal divide-and-conquer solution, and a parallel version of the divide-and-conquer solution.

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,*

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

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

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

You will implement three functions with the following signatures:

```
void solveClosestPairBrute(PointArray *points, PairResult *result)
```

```
void solveClosestPairRecurse(PointArray *xPoints, int beginX, int endX,
PointArray *yPoints, int beginY, int endY, PairResult *result)
```

```
void solveClosestPairParallel(PointArray *xPoints, int beginX, int
endX, PointArray *yPoints, int beginY, int endY, PairResult *result)
```

all of which can be found in the file `StudentSolution.cpp`.

The four arguments are:

- `xPoints, yPoints`: a list of points sorted by x value and y value, respectively. Note 1: these arrays are already sorted by X and Y using standard quicksort. Sorting is done for you and you are **not** required to write a sorting algorithm of any kind for this homework assignment. Note 2: You may have to create a copy of `yPoints` in order to make the recursive call. Note that you should make this copy in parallel. *Hint:* Use parallel-prefix. It is similar to partitioning in Quicksort as we learned in class.
- `beginX`, `beginY`: at times, you will want to inspect only a portion of the complete point array. Use this argument to specify the beginning of the subset of the array that you are currently interested in. You will definitely want to use the `beginX`, but you may not use `beginY`, it depends on your implementation.

- `endX`, `endY`: use this argument to specify the exclusive end index of the points that you are currently interested in. (10 means up to but not including point 10) Again, you will want to use `endX`, but may not use `endY`.
- `result`: the structure in which you should store your solution.

The class `PairResult` has been defined for you. It has 3 member variables:

- `double distance`: distance between the closest pair of points. This field is initialized to infinity.
- `Point *pointOne, *pointTwo`: the two points that are closest together

A correct answer will have all three fields correctly populated.

There are several helper classes that can be found in the same directory. You will find these classes and their methods very helpful. Do not modify these classes; we will grade your code with the original versions of these classes.

Compiling your code. Your code can be compiled by typing "`make ClosestPair`" into the command line.

Running your code. Your code can be run with many possible arguments. Here are just a few examples:

`./ClosestPair -help`	shows all arguments for customizing your run
`./ClosestPair`	runs on a randomly generated stock sequence
`./ClosestPair -n 100`	runs on 100 randomly generated points
`./ClosestPair -f input1`	runs with the input file `input1`
`./ClosestPair -s 243`	runs on points randomly generated using seed 243

Testing Your Code. Be sure to test your code for correctness with a variety of number of points and examples to convince yourself that your implementations are correct.

Three input files of points `input1-3` are provided for you in the `hw02` directory. Unless specified using `-s`, the random number generator in the code always generates the same points with the same seed. This may help you in your debugging.

Feel free to create new input files, or randomly generate sequences of different lengths with different seeds. We will run your student solution against our reference solution and the solutions must match in order to get full credit. Also, the parallelism reported by cilkview should make sense.

(a) Naïve Implementation[5 points]

Implement a quadratic work brute-force solution to the closest pair problem in a function called `solveClosestPairBrute()`. (Note: once again, if you run the

`ClosestPair` program on an array of points with length larger than 10,000, the program will not call your brute force method.)

(b) Divide and Conquer Algorithm Analysis [10 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

$$W_{closestPair}(n) = 2W_{closestPair}(n/2) + O(n) + W_{combine}(n), \tag{1}$$

where $W_{combine}(n)$ denotes the work needed in the combine step. Now try to implement a combine step in $O(n)$ work so as to get a total work of $O(n \lg n)$.

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 implement 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 δ, such that if R is a $\delta \times \delta$ square and the closest pair distance in R is at least δ, then R contains at most C_P points.*

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

(c) Divide and Conquer Algorithm [15 points]

Implement your recursive algorithm `closestPair` in a method called `solveClosestPairRecurse` in `StudentSolution.cpp`.

(d) Parallelize your Divide and Conquer Analysis [15 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,*[2] where n is the number of input points. We remark

[2] Don't worry as long as your span is logarithmic

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. Explain the implementation in your written handin, calculating the work and the span of your implementation.

(e) Implement your parallel solution [15 points]

You should implement this algorithm in a method called `solveClosestPairParallel()`. Implement it by making use of the Cilk Plus keywords `cilk_spawn`, `cilk_sync`, and `cilk_for`.

(f) Measure the work, span, and parallelism of your parallel divide and conquer implementation [5 points]

Include in your written handin the values that you see for work, span, and parallelism (not the burdened one) reported by `cilkview` under the "Cilk Parallel Region(s) Statistics" section. Report the values for a moderately sized input. Note: don't forget to uncomment the annotations for `cilkview` before you run `cilkview` for these measurements.

Problem 2-2. Computing the Skyline [25 points]

You are given a array B of building, where a building b_i is represented as the triple (ℓ_i, h_i, r_i) such that ℓ_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, 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. $y_i = \max\{h : (\ell, h, r) \in B \; and \; y_i \in [\ell, r)\}$: 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 $O(n \lg n)$ and logarithmic span. The smaller the span, the better.

Problem 2-3. Parse Text in Parallel [10 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.