CS39002: Operating Systems Lab
Spring 2013

Assignment 4
Due: March 18, 2013, 1 pm

In this assignment, you make a parallel implementation of Preparata and Hong's algorithm for the computation of convex hulls in two dimensions. The algorithm consists of two phases. In the first phase, you sort the given points in the increasing order of their x-coordinates. In the second phase, the convex hull is computed using a divide-and-conquer method. You are supposed to parallelize both the phases using threads. You are given a maximum depth d of recursion to be used in both the phases. A typical choice for d is three.

Your program first generates the input data set in two possible ways governed by a command-line argument. In the first case, your program should be executed as

```
./a.out 1 n d
```

If the first command-line argument is 1, then n points are generated uniformly randomly in the unit square. In the second case, the first command-line argument is 2, like the following:

```
./a.out 2 d CH.dat CH1.dat CH2.dat
```

Now, the input data is to be read from the file `CH.dat` which is organized as follows. The first line of the file stores the value of n. This is followed by n lines, each containing the x- and y-coordinates of a point (again inside the unit square) separated by a space. In this case, you should also output two files `CH1.dat` and `CH2.dat` described later. The value of d is now read from the second command-line argument.

In both the running modes, the coordinates of the n points are stored in a global array. In order to simplify the second phase, you may take n to be of the form 3×2^s for some positive integer s.

Phase 1: *Parallel merge sort*

The main thread (assumed to be at level zero in the recursion tree) creates two worker threads at the first level, each of which recursively sorts one half of the array. In the recursive call, each of these threads again creates two worker threads in the second level. This procedure is continued until the depth of recursion equals d and/or the number of entries in the subarray becomes too small (like less than or equal to twelve). The threads at the bottommost level can use any algorithm to sort the subarray assigned to it.

All the threads (except the main thread) also take part in parallel merging. Let U be a thread at some level greater than zero, and V its sibling thread in the recursion tree. Suppose that U gets the unsorted subarray $A[i \ldots j]$, and V gets the unsorted subarray $A[j+1 \ldots k]$ as their respective inputs. These two subarrays are sorted recursively (or by an explicit sorting algorithm if U and V belong to the bottommost level of the recursion tree). The threads then merge the two sorted subarrays in parallel, and store the sorted result in the array segment $A[i \ldots k]$. They use a temporary array B of size $k - i + 1$. The thread U fills the first $j - i + 1$ entries in B (using the standard merging loop, but stopping as soon as the required number of elements are copied), whereas the thread V fills the last $k - j$ entries in B (working backward from the largest to the smaller entries). When both the threads are done with parallel merging, they copy their respective parts from the temporary array B to the argument array A. Both the threads then exit, and their creater thread (unless at the root of the recursion tree) takes part in parallel merging with its sibling.

When the main thread finds that the two threads at level one have exited, it knows that the entire array A is sorted. It then proceeds to the second phase. This recursive merge sort implementation requires no mutual exclusion. Use appropriate **barriers** for synchronizing the threads. For example, both siblings must synchronize before they start parallel merging.

If you are using running mode 2, then the main thread should save the sorted list of points in the file `CH1.dat`. You do not need to store n and d (as in the input file). You store only the x- and y-coordinates of the n points with each line storing two floating-point values separated by a space.

Partial credit: If you cannot handle the above recursive implementation, do the following. The main thread reads an integer $k > 2$ from the user, and creates k worker threads, each of which sorts a $1 / k$-th portion of the array. After all the k worker threads exit, the main thread uses a k-way merging algorithm to merge the k sorted subarrays. This is to be evaluated in 70% of the total credit for Phase 1.

Phase 2: *Parallel Convex Hull Construction*

From the viewpoint of implementation, this phase is quite similar to the first phase. The main thread creates two worker threads for computing the left hull and the right hull in parallel. These two threads also merge the left and right hulls in parallel. More precisely, the upper and the lower tangents are computed in parallel by the two threads. Moreover, the portions of the two subhulls between the tangents are thrown out by the worker threads in parallel. When they terminate, their creater sees the complete hull as a single list.

The two worker threads at the first level recursively compute their respective hulls by creating two threads each. Recursion by creation of new threads stops after level d. The threads at the d-th level proceed recursively except that no new threads are created. Recursion also stops when the number of points becomes three. In this case, it is easy to compute the hull manually.

In this phase too, there is no need for mutual exclusion. The sibling threads must synchronize before they start the tangent computations in parallel. This time, you use **condition variables** for this purpose.

Print the vertices on the hull and their count, on the screen. In running mode 2, the main thread should additionally write the output of the convex hull in a file `CH2.dat`. This file will contain only a clockwise listing of the vertices of the hull with the first point repeated in the last line. Each line should store the x- and y-coordinates of one vertex separated by a space. The listing may start from any vertex. Only the first point should be repeated at the end so that a complete closed polygon can be generated by joining the points by lines.

Partial credit: Take $d = 1$, that is, only the main thread creates two threads. These threads proceed independently to compute the left and right hulls. When they both exit, the main thread merges the two hulls by computing the two tangents. Again, this will be evaluated in 70% of the total credit for Phase 2.

In order to know more about this recursive algorithm, you are referred to an earlier assignment found here:

http://cse.iitkgp.ac.in/~abhij/course/lab/CSL-I/Autumn06/Assgn3.pdf

The last three pages are also linked here as a supplement. You can use gnuplot to verify the correctness of your implementation, as suggested in the old assignment.

We suggest storing a convex polygon as a doubly linked list (circular if needed) of the vertices of the polygon. Traversal using the forward links should produce a clockwise listing of the vertices in the polygon.

Submit a single C file that implements both the phases. The data files must not be submitted.