CSS 503
Program 1: Parallelizing a Convex-Hull Program with Multi-Processes

Professor: Munehiro Fukuda

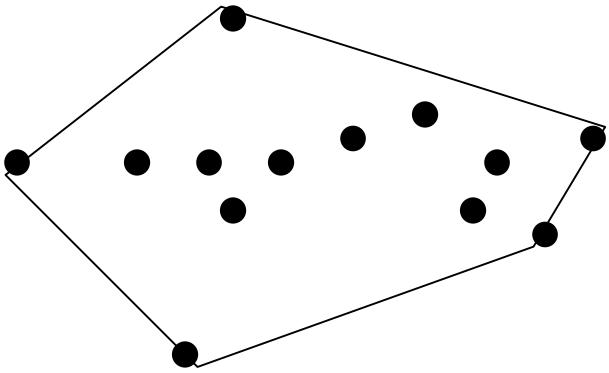
Due date: see the syllabus

1. Purpose

In this programming assignment, we will parallelize a convex hull program with multiple processes. The algorithm that we use is a combination of divide-and-conquer and Graham's scan. Divide-and-conquer divides an entire problem into a tree of sub-spaces, each small enough to be solved with an independent process. We fork child processes in a tree as partitioning a problem with divide-and-conquer and let these processes work on their own sub-space in parallel.

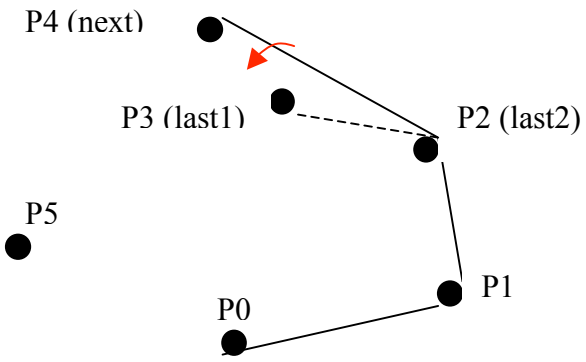
2. Convex Hull

Definition: The convex hull CH(Q) of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior. Intuitively, we can think of each point in Q as being a nail sticking out from a board. The convex hull is then the shape formed by a tight rubber band that surrounds all the nails, (from Introduction to Algorithms by T. H. Cormen et al).



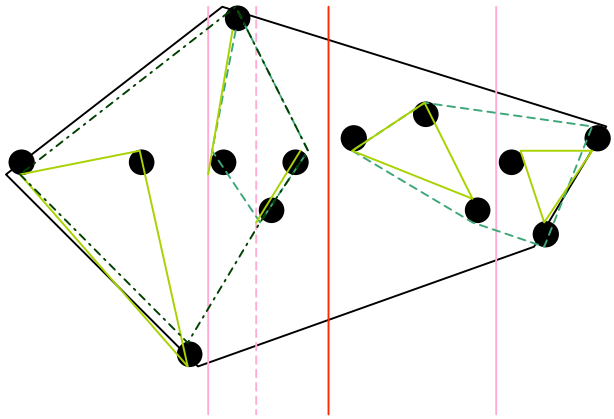
3. Graham's Scan

This algorithm maintains a stack S of candidate points. Each point of the input set Q is pushed once onto the stack, and the points that are not vertices of CH(Q) are eventually popped from the stack. When the algorithm terminates, stack S contains exactly the vertices of CH(Q), in counterclockwise order of their appearance on the boundary. The scan will be performed as follows: Choose three consecutive points, last2, last1, and next; check if the leftward scanning bar that emanates from last2 actually hits last1 first before the next point; if so keeps last1 as a candidate, otherwise drop off last1; shift next to last1; and get a new point from the stack S as next. Repeat until we reach the very first point.



4. Combination with Divide-and-Conquer

Graham's Scan completes in O(n log n) but runs in sequential. We will make this program easier to be parallelized, using divide-and-conquer that recursively divides Q into two subsets until each becomes small enough to have at least two points and thus to create a triangle or a line segment quickly as a convex hull. Thereafter, we will go through the conquer phase as repeatedly merging two convex hulls into a larger subset, excluding inner points within this union of the hulls, and applying Graham's Scan to it at each repetitive stage.



5. Program Structure

Your work will parallelize the convex hull program that the professor got prepared for. Please login uw1-320-lab.uwb.edu and go to the ~css503/prog1/ directory. You can find the convex.cpp that consists of the following functions.

Function Name	Description
void **init**(deque<Point> &q, int nPoints)	Initializes points, each with (x, y) axes and an id.
void **divide**(deque<Point> &q, deque<Point> &s1, deque<Point> &s2)	Divides q into s1 and s2. It is used in divide_and_conquer( ).
double **polar_angle**(Point &o, Point &p, double &distance)	Computes the polar angle and distance between the origin o and the other point p.
void **merge**(deque<Point> &q, deque<Point> &s1, deque<Point> &s2)	Merges two deques s1 and s2 into q as removing those points not in polar angles.
bool **leftturn**(Point &a, Point &b, Point &c)	Checks if B is hit by line that revolves left on A to B.
void **graham**(deque<Point> &q)	Performs the graham algorithm that scans all points from the bottom as revolving a line leftward and eliminating non-convex points.
void **divide_and_conquer**(deque<Point> &q)	Divides a list of all points by two until each subset includes at least two, and thereafter merges those that make a convex.
int **main**(int argc, char *argv[])	Creates and initialize a given number (argv[1]) of points, starts a timer, calls divide_and_conquer( ) to find the convex hull for these points, and stops a timer to check the execution time.

6. Parallelization with Multi-Processes

Let's assume that you would like to parallelize this convex-hull program, (i.e., convex.cpp) with N processes or N CPU cores. When you recursively divide Q into two smaller subsets, (say left and right subsets), you should fork a child process and let it work on the right subset, whereas have the original parent take care of the left subset. If you still need to create a child process, have these parent and child

processes create a new child respectively. This process fork should be performed within the divide_and_conquer( ) function of the original program, convex.cpp. Since all point data including global and local variables are automatically copied to new children, you don't have to worry about any data transfer from a parent to new children. Therefore, you don't even have to modify any functions than divide_and_conquer( ) of the original convex.cpp.

However, in the conquer phase where a parent receives a sub convex hull from a child, you need to use a pipe that allows the child to send its sub convex hull to its parent. For this purpose, every time a parent forks a new child, you should create a pipe in advance so that the parent and child process can share the same pipe. This pipe creation and data transfer from the child back to the parent should be also implemented within divide_and_conquer( ).

When sending a sub convex hull from a child to its parent, add the following send( ) and recv( ) functions to your program so that you can only focus on process management in divide_and_conquer( ).

```
/*
 * Sends a deque to a destination process through a pipe fd.
 * @param fd  the file descriptor of a pipe
 * @param q   a deque to send
 */
void send( int fd, deque<Point> &q ) {
  int size = q.size( );
  double x[ size ];
  double y[ size ];
  int id[ size ];

  // serialize all deque items to x, y, and id arrays
  for ( int i = 0; i < size; i++ ) {
    Point p = q.front( );
    x[ i ] = p.x;
    y[ i ] = p.y;
    id[ i ] = p.id;
    q.pop_front( );
  }

  // send all data through a pipe
  write( fd, &size, sizeof( int ) );
  write( fd, x, sizeof( double ) * size );
  write( fd, y, sizeof( double ) * size );
  write( fd, id, sizeof( int ) * size );
}

/*
 * Sends a deque from a source process through a pipe fd.
 * @param fd  the file descriptor of a pipe
 * @param q   a deque to receive
 */
void recv( int fd, deque<Point> &q ) {
  // receive all data through a pipe
  int size = 0;
  read( fd, &size, sizeof( int ) );

  double x[ size ];
  double y[ size ];
  int id[ size ];
  read( fd, x, sizeof( double ) * size );
  read( fd, y, sizeof( double ) * size );
  read( fd, id, sizeof( int ) * size );

  // de-serialized x, y, and id arrays to all deque items
  for ( int i = 0; i < size; i++ ) {
    Point *p = new Point( x[ i ], y[ i ], id[ i ] );
    q.push_back( *p );
  }
}
```

7. Statement of Work

Follow through the parallelization steps described below:

Step 1: Copy the original convex.cpp to convex_mp.cpp.

Step 2: Add the send( ) and recv( ) functions just above divide_and_conquer( ) to convex_mp.cpp.

Step 3: Add the following header files to the top of convex_mp.cpp.

```
#include <sys/types.h> // fork, wait
#include <sys/wait.h>  // wait
#include <unistd.h>    // fork, pipe
#include <stdlib.h>    // exit
#include <stdio.h>     // perror
```

Step 4: Parallelize convex_mp.cpp by modifying the divide_and_conquer( ).

Step 5: Conduct performance evaluation and write up your report.

Focus on #process = 1, 2, and 4, (because uw1-320-16 ~ 23 machines have four CPU cores). Run your program with the following scenario and attach your execution output to your report.

```
css503@uw1-320-18 prog1]$ ./convex 100
do you want to display initial data? n
elapsed time = 1032
do you want to display result data? y
point[47].x = 7739.000000 .y = 12.000000
point[72].x = 9503.000000 .y = 19.000000
point[27].x = 9956.000000 .y = 1873.000000
point[45].x = 9932.000000 .y = 5060.000000
point[74].x = 9708.000000 .y = 6715.000000
point[28].x = 6862.000000 .y = 9170.000000
point[99].x = 5928.000000 .y = 9529.000000
point[15].x = 3929.000000 .y = 9802.000000
point[64].x = 709.000000 .y = 8927.000000
point[32].x = 336.000000 .y = 6505.000000
point[85].x = 124.000000 .y = 4914.000000
point[53].x = 97.000000 .y = 2902.000000
point[33].x = 846.000000 .y = 1729.000000
point[50].x = 795.000000 .y = 570.000000
point[5].x = 2362.000000 .y = 27.000000
[css503@uw1-320-18 prog1]$ ./convex_mp 100 4
do you want to display initial data? n
elapsed time = 2386
do you want to display result data? y
point[47].x = 7739.000000 .y = 12.000000
point[72].x = 9503.000000 .y = 19.000000
point[27].x = 9956.000000 .y = 1873.000000
point[45].x = 9932.000000 .y = 5060.000000
point[74].x = 9708.000000 .y = 6715.000000
point[28].x = 6862.000000 .y = 9170.000000
point[99].x = 5928.000000 .y = 9529.000000
point[15].x = 3929.000000 .y = 9802.000000
point[64].x = 709.000000 .y = 8927.000000
point[32].x = 336.000000 .y = 6505.000000
point[85].x = 124.000000 .y = 4914.000000
point[53].x = 97.000000 .y = 2902.000000
point[33].x = 846.000000 .y = 1729.000000
point[50].x = 795.000000 .y = 570.000000
point[5].x = 2362.000000 .y = 27.000000
[css503@uw1-320-18 prog1]$ ./convex 20000
do you want to display initial data? n
elapsed time = 200391
do you want to display result data? n
[css503@uw1-320-18 prog1]$ ./convex_mp 20000 1
do you want to display initial data? n
elapsed time = 211026
do you want to display result data? n
[css503@uw1-320-18 prog1]$ ./convex_mp 20000 2
do you want to display initial data? n
elapsed time = 113914
do you want to display result data? n
[css503@uw1-320-18 prog1]$ ./convex_mp 20000 4
do you want to display initial data? n
```

```
elapsed time = 61255
do you want to display result data? n
```

Your minimum requirements to complete this assignment include:

(1) Your program creates argv[1] – 1 child processes and involve all argv[1] processes in parallel computation.

(2) The performance improvement with four processes applied to 20,000 points should be equal to or larger than 211026 / 113914 > 1.8 times.

(3) The performance improvement with four processes applied to 20,000 points should be equal to or larger than 211026 / 61255 > 3.4 times.

8. What to Turn in

This programming assignment is due at the beginning of class on the due date. Please turn in the following materials in a hard copy. No email submission is accepted.

Criteria	**Grade**
Documentation of your parallelization strategies including explanations and illustration in one or two pages. (No more than two, otherwise – 2pts)	20pts
Source code that adheres good modularization, coding style, and an appropriate amount of commends. • 25pts: well-organized and correct code to run the same number as CPU cores, using fork( ) and wait( ). (Four CPU cores use only four processes including parent.) • 23pts: messy yet working code or code with minor errors receives • 20pts: code with major bugs, no parallelization, or incomplete code receives	25pts
Execution output that verifies the correctness of your implementation and demonstrates any improvement of your program's execution performance. • 25pts: Correct execution and performance improvement up to 4 CPUs. • 20pts: Correct execution but performance improvement up to 2 CPUs • 15pts: Correct execution but little performance improvement • 10pts: Incorrect results • 5pts: No results	25pts
Discussions in one or two pages. (No more than two, otherwise – 2pts) • Analysis of the effectiveness of your parallelization. (+10pts) • Possible performance improvement of your program. (+10pts) • Limitation of your program. (+5pts)	25pts
Lab Sessions 1 If you have not yet turned in a hard copy of your source code and output or missed any session(s), please turn in together with program 1.	5pts
Total Note that program 1 takes 11% of your final grade.	100pts