Assignment 3

Due Mar 16 by 4pm **Points** 10

A3: Processes and Pipes

- March 7th: To avoid some memory deallocation issues with heap memory and child processes, we've revised the starter code to store the points on the stack rather than on the heap. Update your code as follows:
  - use `git pull` to get updated versions of `utilities_closest.c` and `utilities_closest.h`
  - download this updated version of **closest.c** and merge with any work you've done. The changes you need to make are:
    - under the comment `// Read the points`, replace the original one line with the new three lines
    - remove the call on `free()` at the end of the program
  - rerun the checker on MarkUs to verify your work

Introduction

The `fork` system call is often used to create multiple cooperating processes to solve a single problem. This is especially useful on a multiprocessor where different processes can truly be run in parallel. In the best case, if we have N processes running in parallel and each process works on a subset of the problem, then we can solve the problem in 1/N the time it takes to solve the problem using one processor.

If it were always that easy, we would all be writing and running parallel programs. It is rarely possible to divide up a problem into N independent subsets. Usually, the results of each subset need to be combined in some way. There is also a difficult tradeoff to make between the benefits of parallelism and the cost of starting up parallel processes and collecting results from different processes.

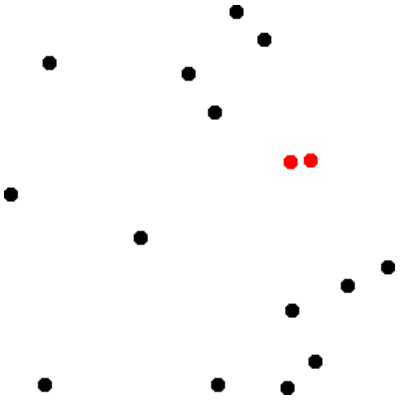
The goal of this assignment is to practice creating and using multiple processes in C. For this assignment, you will write a multi-process implementation of the **Closest Pair of Points** (https://en.wikipedia.org/wiki/Closest_pair_of_points_problem) that takes advantage of parallel computation.

Background

Given a set of `n` points (where `n` is at least 2), find the closet pair of points in the set. Distance is measured using Euclidean distance. For points (x_1, y_1) and (x_2, y_2), the distance between them is:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Visually, in the set of points below example, the red points are the two closest points in the set:



The closest pair of points problem is a computational geometry problem that can be solved using a *divide and conquer* algorithm.

As a reminder of what a divide-and-conquer algorithm is:

> A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution

> to the original problem.

Source: **Wikipedia** (https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm)

A high-level explanation of the problem is explained in this handout, but a more detailed explanation is available **in this PDF printout of Section 33.4** from "Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein.

Solutions

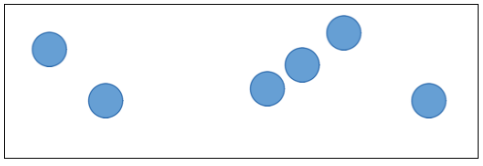
Brute-force Approach

The brute-force (naive) solution can be implemented in just a few lines, by finding the distance between all pairs of points and returning the distance between the closest pair. This approach is $O(n^2)$.

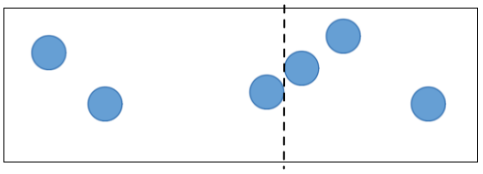
Recursive Divide-and-Conquer Approach

The problem can be solved in $O(n\log_n)$ time using the following recursive divide and conquer approach:

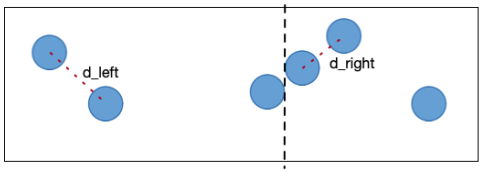
1. Sort the points in ascending order by their x-coordinates.
2. Split the list of points into two equal-sized halves. This is like drawing a vertical line through the plane to divide the points into a left-hand-side and right-hand-side, each containing an equal number of points.
3. Solve the problem recursively on the left and right halves.
4. Find the distance between the closest pairs of points where one point lies on the left-hand-side, and the other point lies on the right-hand-side. This can be done in linear time.
5. The final answer is the minimum among the:
    i. Distance between the closest pair of points on the left-hand side.
    ii. Distance between the closest pair of points on the right-hand side.
    iii. Distance between the closest pair of points where one point is on the left-hand-side and the other point is on the right-hand side.



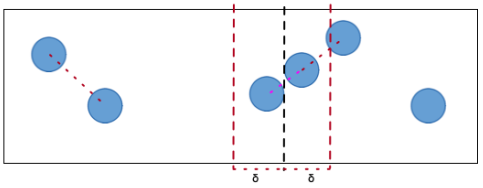
Step 1: Sort the points by their x-coordinate



Step 2: Split the set in half



Step 3: Recursive step — Find the closest points in each half



Step 4: Consider the points on either side of the dividing line.
Only consider points within δ =min(d_left, d_right).

Starter code

Brute-force (utilities_closest.c)

We provide the brute-force implementation in the starter code. For the brute-force approach, calculate the distance between each pair of points, keeping track of the pair with the smallest distance. This implementation is $O(n^2)$.

Recursive Divide-and-conquer Single-Process Implementation (serial_closest.c, utilities_closest.c)

We provide the recursive divide-and-conquer single-process implementation in the starter code.

Generate binary file with points (generate_points.c)

This program can be used to produce a file containing a set of randomly generated points.

Your Tasks

Implement the main program (closest.c)

You will write a C program called `closest` that takes 2 arguments:

- the name of the input file containing the points, and
- the maximum process tree depth.

The main program will be called as shown below (note: the option `-d` and its argument could appear before `-f` and its argument). You must use `getopt` to read in the command-line arguments. The command `man 3 getopt` will give you the correct man page, which has a nice example that you can use as a template.

```
closest -f filename -d pdepth
```

If the correct number of command-line arguments and the correct options (`-f` and `-d` are provided), you may assume that the values corresponding with the options are valid. If the incorrect number of command-line arguments or incorrect options are provided, you must report that using the `print_usage` function (see `closest.c` starter code) and exit the program with an exit code of `1`.

You may assume that `pdepth` will be less than or equal to 8.

Once you have written code to parse the command line arguments, the starter code provided will read the points from the input file, sort the points, run your parallel-process implementation of the divide-and-conquer algorithm, and report the results.

Implement a Recursive Divide-and-conquer Multi-process Implementation (parallel_closest.c)

Your task is to implement the divide-and-conquer multi-process implementation described below.

`double closest_parallel(struct Point *p, int n, int pdmax, int *pcount)`

- `p`: Pointer to an array of points sorted according to x coordinate.
- `n`: Number of points in array
- `pdmax`: The maximum depth of the worker process tree rooted at the current process. (For the first call on this function, `pdmax` is equal to the maximum depth of the process tree as specified by the argument of the command line option `-d`.)
- `pcount`: To be populated with the number of worker processes. Initialized to 0 at the start of the program (the parent process does not count as a worker process).

Note: This is the only function in your program that should be invoking `fork().`

Let *current_depth* be the depth of the current process in the process tree. The parent process is at depth 0. The first two children are at depth 1, and so on.

Below is the logic for the **creation of** and the **co-ordination between** the worker processes, carried out by function `closest_parallel`:

1. If *n < 4*, or *pdmax == 0* (i.e., the maximum depth has been reached), invoke `closest_serial()` to obtain the answer without creating any more processes.
2. Otherwise, split the array of points into two equal sized halves. The left half must consist of the leftmost *floor(n/2)* elements, and the right half must consist of the rest of the elements. (If `n` is odd, the right half will consist of one more point than the left.)
3. Create two child processes to to solve the left half and right half of the problem:
   a. Create a pipe, which the first child process will use to communicate with its parent (i.e., the current process).
   b. Fork a child, which will:
      i. Invoke `closest_parallel()` on the left half of the array, to obtain the distance between the closest pair of points on the left-hand side.
      ii. Send the distance between the closest pair of points back to the parent process by writing it to the pipe.
      iii. Exit with status equal to the number of worker processes in the process tree rooted at the current process (not counting the current process).
   c. Create a pipe, which the second child process will use to communicate with its parent (i.e., the current process).

   d. Fork a second child, which will:
      i. Invoke `closest_parallel()` on the right half of the array, to obtain the distance between the closest pair of points on the right-hand side.
      ii. Send the distance between the closest pair of points back to the parent process by writing it to the pipe.
      iii. Exit with status equal to the number of worker processes in the process tree rooted at the current process (not counting the current process).
4. Wait for both child processes to complete (each process should have at most two child processes).
5. Read from the two pipes to retrieve the results from the two child processes.
6. Follow step 4 from the single-process recursive divide-and-conquer solution to determine the distance between the closest pair of points, with distance is smaller than `d`, where one point is in the left half of the array and one point is in the right half of the array. (See `serial_closest.c`)
7. The final answer is the minimum among the:
   i. Distance between the closest pair of points on the left-hand side, found by the first child process.
   ii. Distance between the closest pair of points on the right-hand side, found by the second child process.
   iii. Distance between the closest pair of points where one point is on the left-hand-side and the other point is on the right-hand side.

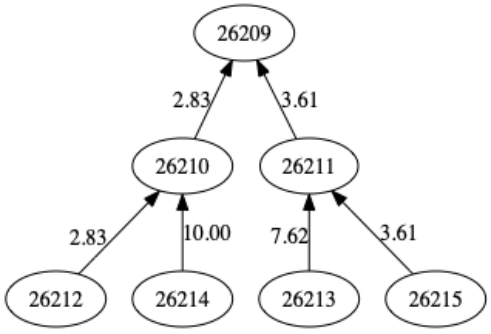
If any process terminates unsuccessfully due to any unexpected situation (e.g., a failed system call) it should return with exit status `1`.

Sample input: `ex_input_20.b`

In the starter code, we provide an example points file `ex_input_20.b` that contains 20 points. This data is used by the checker program on MarkUs and you can also run it locally.

We include two diagrams below to show the structure of the process tree (with the root being the process for the `closest` program). The nodes are labelled with a process id, which will vary from one invocation to the next. The arrows are labelled with values to represent the data written from the child to its parent via the pipe.

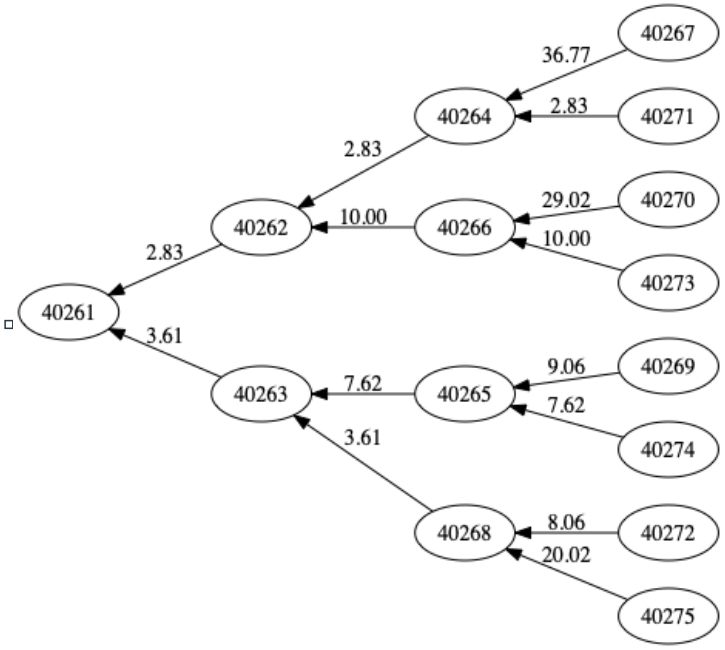
Example input with depth 2 (`./closest -f ex_input_20.b -d 2`):



Assuming the processes run successfully, the exit status for the processes above would be 0 for the four leaf processes, 2 for processes 26210 and 26211, and 0 for the parent process.

The program output would be: `The smallest distance: is 2.83 (total worker processes: 6)`

Example input with depth 3 (`./closest -d 3 -f ex_input_20.b`):



Assuming the processes run successfully, the exit status for the processes above would be 0 for the eight leaf processes, and 2 for processes 40264, 40266, 40265 and 40268, 6 for processes 40262 and 40263, and 0 for the parent process.

The program output would be: `The smallest distance: is 2.83 (total worker processes: 14)`

Marking

We will use testing scripts to evaluate correctness. As a result, it's very important that your output matches the expected output precisely. Also, it is not sufficient to produce the correct output -- following the algorithm specified is required.

For this and future assignments, your code may be evaluated on:

- **Code style and design**: At this point in your programming career you should be able to make good choices about code structure, use of helper functions, variable names, comments, formatting, etc.
- **Memory management**: your programs should not exhibit any memory leaks. This means freeing any memory allocated with `malloc`Use `valgrind` to check your code.
- **Error-checking**: library and system call functions (even `malloc`!) can fail. Be sure to check the return values of such functions, and terminate your program if anything bad happens.
- **Warnings**: your programs should not cause any warnings to be generated by `gcc -Wall`

MarkUs Checker

We have provided a checker program for Assignment 3 on MarkUs. You can run the checks yourself up to once per hour on MarkUs (if server load becomes an issue, we may need to increase the time between runs). After the deadline, we will run a larger set of tests on your submission for marking, so you should test your code thoroughly and not rely solely on the checker. Note that these checks are only for correctness and do not check for issues related to the other marking criteria.

Submission

Please commit to your repository often. We will be looking for the following files in the a3 directory of your repository:

- `closest.c`
- `parallel_closest.c`

Do not commit `.o` files, executables, or test files to your repository.

You must *NOT* change the `.h` files, the other `.c` files, or the provided `Makefile`. We will be testing your code with the original versions of those files. We will run a series of additional tests on your full program (in `closest.c`) and also on your `closest_parallel` function (from

`parallel_closest.c) so it is important that they have the required signatures.

Remember to also test your code on teach.cs before your final submission. Your program must compile cleanly (without any warning or error messages) on teach.cs *using our original Makefile* and may not crash (seg fault, bus error, abort trap, etc.) on our tests. Programs that do not compile or which crash will be assigned a 0.

Timing Your Program (Optional)

This part is optional and not for any marks, so complete it only to satisfy your curiosity!

Using `clock_gettime`: You should read the man page for `clock_gettime`, but here is an example of how to use it, and how to compute the time between two readings of `clock_gettime`

```
#include <time.h>

struct timespec start, end;
double time_diff;

if (clock_gettime(CLOCK_MONOTONIC_RAW, &start)) {
    perror("clock_gettime");
    exit(1);
}

// code you want to time

if (clock_gettime(CLOCK_MONOTONIC_RAW, &end)) {
    perror("clock_gettime");
    exit(1);
}

timediff = 1e3 * (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1.0e6;

fprintf(stdout, "%.4f\n", time_diff);
```

The real question is how many processes should we use to get the best performance out of our program? To answer this question, you will need to find out how long your program takes to run. Use `clock_gettime` to measure the time from the beginning of the program until the end, and print this time to standard output. Now try running your program on different numbers of processes on different sized datasets.

Think about what the performance results mean. Would you expect the program to run faster with more than one process? Why or why not? Why does the speedup eventually decrease below 1? How does the speedup differ between data sets? Why? Did the performance results surprise you? If so, how?