Project 2: Collision Detection

[Note: This assignment makes use of AWS and/or Git features which may not be available to OCW users.]

In this project you will optimize a graphical screensaver program for multicore processors using the Open Cilk parallel programming interface.

1 Deliverables

- ☐ *Team formation*
- ☐ *Team contract:*
- ☐ *Beta submission*
- ☐ *Beta write-up*
- ☐ *Final submission*
- ☐ *Final write-up*

Remember that progress reports are due weekly at 7:00 P.M. every Thursday.

2 Getting started

Snailspeed Ltd. has been put onto the map due to its newly patented *iRotate* algorithm. Due to your role in Snailspeed's rise to prominence, you have been promoted to CCTO (Co-Chief Technology Officer). Custom engraved door plates and corner offices aren't cheap; and, unfortunately, Snailspeed didn't have enough revenue left over to hire any additional engineers. Instead, Snailspeed has hired a small team of corporate management consultants from InchWorm Advisors.

InchWorm Advisors has suggested that Snailspeed pivot towards the development of multicore software in order to remain on the cutting edge. As a result of these discussions, Snailspeed is planning to launch *SnailSaver*, a disruptively high-performance screensaver application designed for multicore processors.

It is up to you and your fellow CCTO to transform *SnailSaver* from dream to reality. InchWorm Advisors has suggested that you look into the Open Cilk parallel programming interface to parallelize *SnailSaver*. Open Cilk may be the key, according to Inchworm Advisors, to transforming Snailspeed Ltd. into a major player in the high-performance computing industry.

2.1 Team formation

You should begin by finding a teammate (your fellow CCTO).

If you do not fill out the form by the deadline, we will randomly assign you to a partner.

2.2 Team contract

You and your teammate must agree to a team contract. A team contract is an agreement among teammates about how your team will operate — a set of conventions that you plan to abide by. The requirements for the team contract are the same as before. For guidance in writing your team contract, reference Project 1's description.

2.3 Getting the code

We strongly recommend groups practice pair programming, where partners are working together with one person at the keyboard and the other person serves as watchful eyes. This style of programming will lead to bugs being caught earlier, and both programmers always remaining familiar with the code.

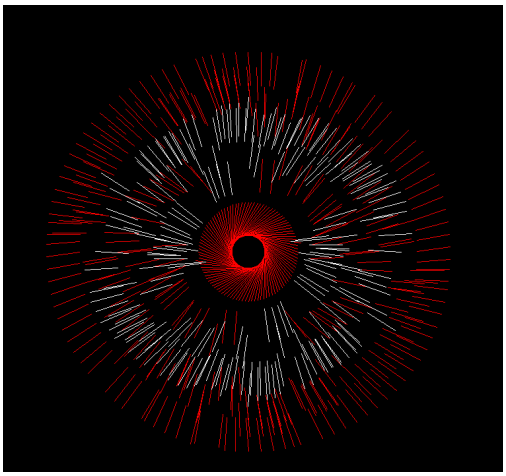

Figure 1: Screenshot from `input/explosion.in` input.

3 Optimizing collision detection

You will be optimizing a screensaver. The screensaver consists of a 2D virtual environment filled with colored line segments that bounce off one another according to simplified physics.

The first part of the project is to make algorithmic changes to the collision detection algorithm to reduce the total work performed by the serial code. Currently, the screensaver uses an extremely inefficient algorithm to detect collisions between line segments. At each time step, the screensaver iterates through all pairs of line segments, testing each pair to see if they've collided. This pairwise method is expensive, as it requires $\Theta(N^2)$ collision tests for N line segments.

You will begin by implementing a quadtree data structure to reduce the total number of line segment pairs that must be checked each time step. You should not, at this point, use Open Cilk (or any other method) to parallelize your collision detection algorithm. The screensaver will be parallelized in the second part of this project (described in Section 4).

3.1 Running the screensaver

The screensaver can be executed both with and without a graphical display. While collecting performance data, make sure to execute the screensaver without the graphical display to obtain more accurate performance results.

```
Usage: ./screensaver [-g] <numFrames> [input-file]
  -g : show graphics over X11
```

The input file is optional and will default to `input/mit.in`. The `input` directory has several other (fun!) examples. Use the `-g` option to look at them. Figure 1 shows a screenshot from `input/explosion.in`.

Here's the output of one run with the default input:

```
$ ./screensaver 4000
Number of frames = 4000
Input file path is: input/mit.in
---- RESULTS ----
Elapsed execution time: 79.925466s
1262 Line-Wall Collisions
19806 Line-Line Collisions
---- END RESULTS ----
```

You should benchmark your serial code on the `awsrun` compute nodes. The `awsrun` nodes, however, are configured to time out after 40 seconds. You should therefore first benchmark your code with 1000 iterations or so. As your code gets faster, you should ramp this up to 4000 iterations or more (on the default input). Be sure to note the number of collisions detected during the screensaver's execution.

3.2 Things to remember

Before you start modifying the implementation, keep the following points in mind:

- The code is written in C and is compiled with `clang -std=gnu99`. You may add other compiler flags, but do not change the `-std=gnu99`.
- *Ordered collision processing:* The unmodified reference implementation establishes an unambiguous order in which detected collisions should be processed at the end of a time step by ordering lines using unique identifiers.[1] As a result, your algorithm should report the same number of collision events as the reference implementation as long as your algorithm detects the same set of collisions at each step. *Implementations that do not report the same number of collisions as the reference implementation will be considered incorrect.*
- *Reference testing:* A technique that you may find useful in this project is reference testing, which is comparing the execution of two different implementations of a function to ensure that the two implementations have identical behavior.

  Let's say you have modified the `intersect` function in `intersection_detection.c` but aren't sure if it is correct. There are two ways you can proceed. The first is to write unit tests, as in 6.005 and 6.031. The second is to test it live by having two versions of the function: `intersect_orig` (the original implementation) and `intersect_new` (your new implementation). Then, for `intersect`, you can write:

```
09 IntersectionType intersect(Line *l1, Line *l2, double time) {
10   assert(intersect_orig(l1, l2, time) == intersect_new(l1, l2, time));
11   return intersect_new(l1, l2, time);
12 }
```

[1] Processing the same set of collisions in two different orders can yield different results in discrete time. For example, in `collision_world.c`, `collisionSolver` updates the positions and velocities of each of the two lines, and these updates are not commutative (or even associative). To ensure that modified implementations follow the same order as the original, the list of intersecting lines is sorted using line IDs before it is processed.

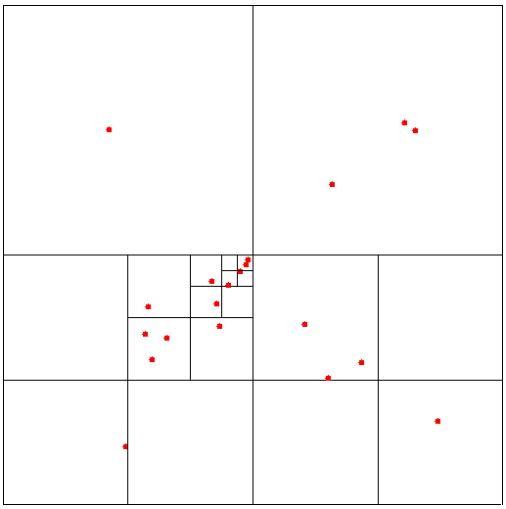

Figure 2: Quadtree partitioning where each partition contains at most three points.

If the assertion fails when you run the screensaver, you can find out what arguments caused it to fail. Remember to take out the dead code in your submission, however, so that your MITPOSSE mentors don't have to look at it!

- *Quadtrees and other data structures:* You're free to implement any data structure you like to make the screensaver faster. We ask, however, that you begin the project by implementing a quadtree and exploring a few questions about its performance characteristics.
- *Serial optimization:* You may be tempted to jump directly into parallelizing your code as a means to gain performance. Generally, however, the best parallel programs are those that are first highly optimized serially and then made parallel afterwards. Consequently, we recommend when approaching this project to make the serial version of the screensaver as fast as possible before going on with parallelization.

3.3 Quadtrees

A quadtree is a spatial data structure that stores elements in a partitioned 2-dimensional space. Quadtrees are often used to efficiently store and lookup $2D$ points. Each quadtree node is associated with a square region in 2D space. During quadtree construction, any node whose region contains more than r points (where r is a tunable parameter) is recursively subdivided into 4 quadrants. Figure 2 illustrates the plane partitioning performed by a quadtree when $r = 3$.

Because of the way 2D space is partitioned in a quadtree, elements at the same location are always placed together in the same partition. This grouping is extremely helpful in collision detection. Since collisions are localized events, in order for two elements to collide, they must both be within the same quadtree partition. Correspondingly, if two elements are in different quadtree partitions, they cannot possibly collide.

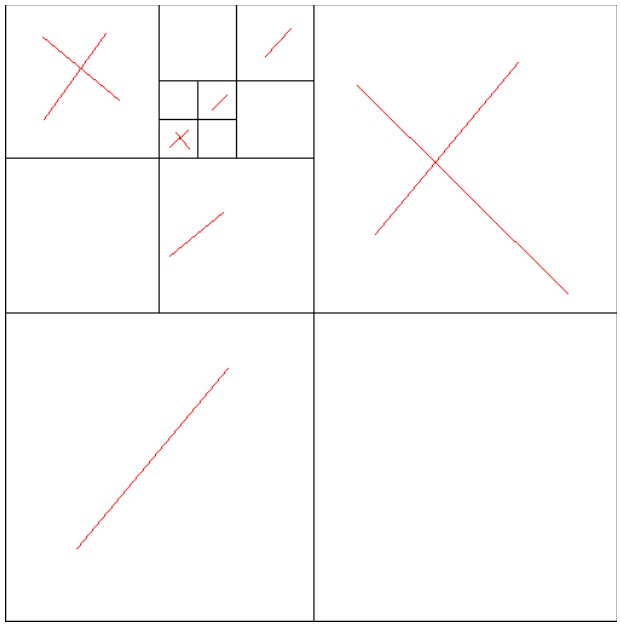

Figure 3: Quadtree storing line segments.

Caution! In order to use quadtrees for collision detection, we actually want to think about storing parallelograms instead of line segments. Imagine a line moving very fast through $2D$ space. During a single time step, this line may exit the region associated with its current quadtree node and collide with lines in other regions. Such collisions would not be detected if we only checked that line against those stored in its current node. A fix for this problem is to instead store the region, a parallelogram, that will be sweeped by the line during the next time step.

Now, making this change to quadtrees is not a trivial matter. For example, while Figure 3 suggests that line segments can at least sometimes be handled by quadtrees, in Figure 4 one of the line segments cannot fit into any of the partitions. What can you do about this problem? Can you still use quadtrees to effectively speed up collision detection? **These sort of issues should be discussed in your write-up.** *Hint:* Do all line segments need to be stored in leaf nodes?

Using a quadtree, rewrite collision detection to be much more efficient. You should use `intersect` from `intersection_detection.c` to test if two line segments will intersect in the next time step. For now, use a reasonable value for r. To simplify your implementation, consider destroying the quadtree and reconstructing it at each time step. This way, you will not need to worry about updating the quadtree when line segments move to new positions.

As you test your quadtree, you may want to consider a few key questions for your write-up:

- What speedup did you achieve? Was this what you expected?
- How does varying the maximum number of elements r a quadtree node can store before it needs to be subdivided impact the performance? Why?
- What key design decisions did you make while rewriting collision detection to use a

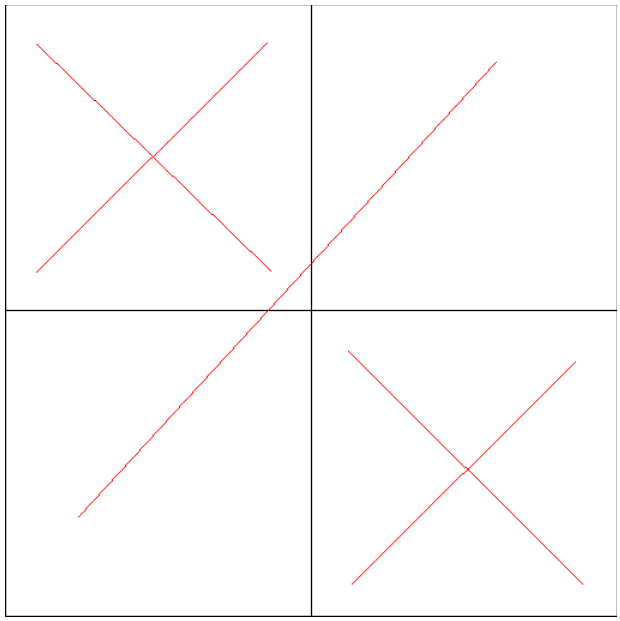

Figure 4: Broken quadtree?

quadtree? For example, once you built a quadtree, how did you use it to extract collisions?

Remember that the number of collisions detected during the screensaver's execution should be the same as the number you recorded for the unmodified code. Moreover, make sure to run the program without the graphics flag to get accurate performance results.

3.4 Further optimization

Look for other opportunities for optimization within the screensaver and describe what you did. Here are some optimization ideas to help you get started:

- Implement a maximum depth for the quadtree and vary it.
- A large percentage of calculations are repeated with each time step. For example, the collision detection code recalculates the length of each line segment, a relatively expensive operation, in each time step, even though the length never changes. For calculations that are repeated in each time step, it might be worthwhile to precompute these calculations and store results for reuse.
- To simplify the implementation, we suggested that you destroy and recreate the quadtree on each time step to avoid having to figure out how to effectively update the quadtree. While simple, this method is a bit wasteful. You can try finding an effective way to update the quadtree so that you don't need to destroy it.
- While our method for testing if two lines intersect is fairly efficient, you could try finding a more efficient way of testing if two lines intersect.

Keep track of the optimizations that you tried and how well they worked.

4 Parallelization

Once you have optimized your serial execution, you can parallelize your code.

4.1 Profiling the serial program

It is useful to profile your application to determine where you should focus your time when parallelizing your code. The goal is to identify a region of code that comprises of a large percentage of the total execution time but is amenable to a potentially large degree of coarse grained parallelism. Run `awsrun perf record` and `aws-perf-report` to see the profiling statistics.

4.2 Converting global variables to reducers

Before inserting any `cilk_spawn` and `cilk_sync` keywords, you will need to make any accesses to your list of intersecting lines thread-safe. Without such a change, your parallel code may see two threads attempting to concurrently insert an element into the list, causing a determinacy race. Implement a solution based on what you learned from Homework 4: Reducer Hyperobjects.

4.3 Parallelizing the application

Parallelize your code by inserting `cilk_spawn` and `cilk_sync` keywords to the regions of code you have identified as worth parallelizing. If you do not see much code suitable for parallelization using the Cilk primitives, you may want to modify your collision detection code so that it performs a recursive depth-first search through your quadtree.

In order to benchmark your parallel code, use:

```
$ awsrun8 ./screensaver 4000
```

The `awsrun8` command will run your code on an 8-core AWS machine. **Warning:** If you try to benchmark your code using `awsrun`, it will run on a single core machine, and you will get wildly inaccurate results.

For any changes you make, verify that your code reports the same number of collisions as before. Additionally, verify that the code is race free by running it through the Cilksan determinacy race detector.

4.4 Profiling the parallel code

You can determine the span and work of your parallel code by executing it through Cilkscale. How much parallelism do you see? Vary the parameters of your algorithm — such as the maximum depth and maximum number of nodes per quadrant, if you are still using a quadtree — as well as any other spawn cut offs you have used in your code. What is the maximum amount of parallelism you can achieve?

4.5 Performance tuning

Since you are running on an 8-core machine, you should tune your code so that it executes as fast as possible when running with 8 workers. To tell the Open Cilk runtime that you want 8 workers, set the `CILK_NWORKERS` environment flag:

```
$ awsrun8 CILK_NWORKERS=8 ./screensaver 4000
```

In your write-up, describe any decisions, trade-offs, and further optimizations that you made.

5 Evaluation

Please remember to add all new files to your repository explicitly before committing and pushing your final changes. Your grade will be based on all of the following:

- *Performance (of correct code):* As with Project 1, your final performance grade will be computed by comparing the performance of your submission to a baseline to be announced. Your code will be benchmarked against inputs not made available to you. Therefore, you should keep your code general and be aware to not over-optimize to any specific type of screensaver patterns. Additionally, we will be benchmarking your code on larger machines (> 8 cores) than what you have access to. So be sure your code is processor oblivious. You can use Cilkscale to validate whether your parallel code scales well or not.
  For correctness, your code should:
  - Yield the same number of collisions as the original code.
  - Compute collisions correctly. When running in graphical mode, the collisions should look semi-realistic as in the code given to you originally.
  - Contain no determinacy races.
- *Progress reports:* You must individually submit a personal progress report every Thursday that briefly describes the work you performed during the past week. The reports should be short: typically, a paragraph in length, describing what project work you engaged in and about how much time you spent on the various activities. Additionally, if necessary, describe any issues that may have arisen, such as with teammates — or rather, especially with teammates.
  Along with your paragraph description, include a summary of the daily number of lines of code you committed using the following script:

  ```
  $ cd <your/projects/base/directory>
  $ loc_summary
  ```

- *Beta group write-up:* You are required to submit a group write-up discussing the work that you performed following a similar structure as with Project 1. **Important:** Please

also push your write-up to your git repository as a `.txt` file. This is so that your MITPOSSE mentor will have access to the write-ups and can read a description of the optimizations you've made.

- *Addressing MITPOSSE comments:* The MITPOSSE will give you feedback on your code quality. We expect you to respond thoughtfully to their comments in your final submission. We will review the MITPOSSE comments and your write-up to ensure that you are addressing them.
- *Final group write-up:* You are required to submit a group write-up discussing the work that you performed since the Beta submission. You should answer all questions asked in this handout and address the following:
  - Provide a clear and concise description of your final strategy and implementation.
  - Discuss any experimentation and optimizations performed.
  - Justify the choices you made.
  - Discuss what you decided to abandon and why.
  - Briefly comment on meeting with your MITPOSSE mentors.
- *Final individual write-up:* You are required to submit an individual write-up where you state what work you did, what work your partner did and how you worked together. Understand that we expect both students to be very familiar with the code sub-mitted. This will show in your write-up and how you describe the project.
- *Screensaver aesthetic test inputs:* The staff has provided a few input files in the *input* directory for your screensaver application. A small portion of your grade is based on the quality of the new inputs you create to test and benchmark your program. These new inputs will be judged based on whether they satisfy *both* of the following criteria: 1. the input has challenging performance or correctness properties; 2. the input is aesthetically interesting, i.e., it is entertaining to watch your screensaver run on your input.

  The staff will curate the most interesting inputs produced by all teams and show them in class.

We will grade your project submission based on the following point distribution:

	Beta	*Final*
Performance (of correct code)	36%	39%
Team Contract	3%	
Addressing MITPOSSE comments		10%
Write-up	5%	5%
Screensaver aesthetic test inputs	1%	1%
Total	45%	55%

As with Project 1, this point distribution serves as a *guideline* and not as an exact formula. The staff will also review your Git commit logs to assess the dynamics of your team. Please ensure that commits are balanced between each team member.

6.172 Performance Engineering of Software Systems

Fall 2018