Project 2: Collision Detection

Beta Due: 11:59pm on Monday, October 3, 2016
Beta Write-up Due: 11:59pm on Friday, October 7, 2016
Final Due: 11:59pm on Friday, October 21, 2016
Write-Up Due: 11:59pm on Friday, October 28, 2016

Last Updated: September 22, 2016

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

1 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 desiged 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 Cilk Plus parallel programming interface to parallelize *SnailSaver*. Cilk Plus may be the key, according to Inchworm Advisors, to transforming Snailspeed Ltd. into a major player in the high-performance computing industry. However, the InchWorm Advisors know the importance of making it work before making it fast, so the beta submission should be serial. The final submission should be parallel.

1.1 Getting the Code

You should begin by finding a teammate (your fellow CCTO) and recording their name in the following form by midnight on **Monday September 26th**:

```
https://goo.gl/forms/Sbohk8OR6YQIRaoj2
```

Feel free to post on Piazza if you are looking for a teammate. If you do not fill out the form by the deadline, we will randomly assign you to a partner. Within 24 hours of the form submission deadline, you will receive an email notifying you of your `team-name`, which you can use to get the project code via:

```
$ git clone \
/afs/athena.mit.edu/course/6/6.172/student-repos/fa16/projects/project2/team-name.git project2
```

You can also browse the code at:

```
https://github.mit.edu/6172-public/project2
```

If you would like to get started before your team repo is set up, we recommend cloning the code from Github and then changing the remote once you receive the team repo, which you can do with the following commands:

```
$ git remote remove origin

$ git remote add origin \
/afs/athena.mit.edu/course/6/6.172/student-repos/fa16/projects/project2/team-name.git
```

(You will not be able to push to the team repo until it is set up, of course.)

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.

2 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 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. Note that you should not, at this point, use Cilk Plus (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 3).

2.1 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 graphical display. This will lead to more accurate performance results.

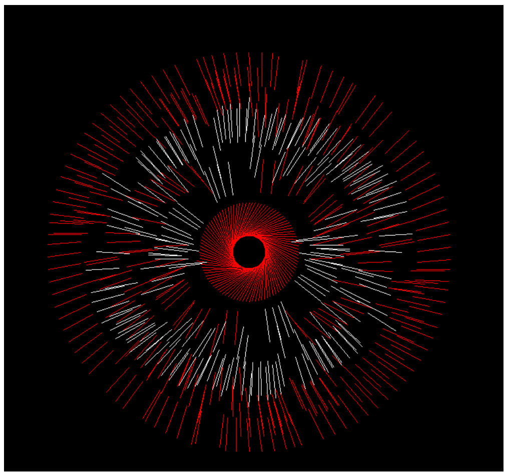

Figure 1: Screenshot from `input/explosion.in` input.

```
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 code on the `cqrun` compute nodes. However, these are configured to time out after 30 seconds. You should therefore at first benchmark your code with 1500 iterations or so, but as your code gets faster you should ramp this up to 4000 iterations or more.

Write-up 1: Run the unmodified screensaver using `cqrun` and report its runtime for 1000 frames. Report the number of collisions detected during the screensaver's execution.

2.2 Things to Remember

Before you start modifying the implementation, keep these points in mind.

- The code is written in C and is compiled with `gcc -std=gnu99`. Do not change this.
- *Ordered Collision Processing:* The unmodified reference implementation orders lines using unique identifiers. This is used to determine an unambiguous order in which detected collisions should be processed at the end of a time step. [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 which do not report the same number of collisions as the reference implementation may be considered incorrect.*
- *Reference Testing:* A technique you may find useful in this project is reference testing where the results of two different implementations of a function are compared during the execution of a program to ensure that the two implementations have identical behavior.

  Let's say you 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. 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:

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

  If the assertion fails when you run the screensaver, you can find out what arguments caused it to fail. However, remember to take out the dead code in your submission 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.

2.3 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 N points (where N is a tunable parameter) is recursively subdivided into 4 quadrants. Figure 2 illustrates the plane partitioning performed by a quadtree when $N = 3$.

[1] This is necessary because 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.

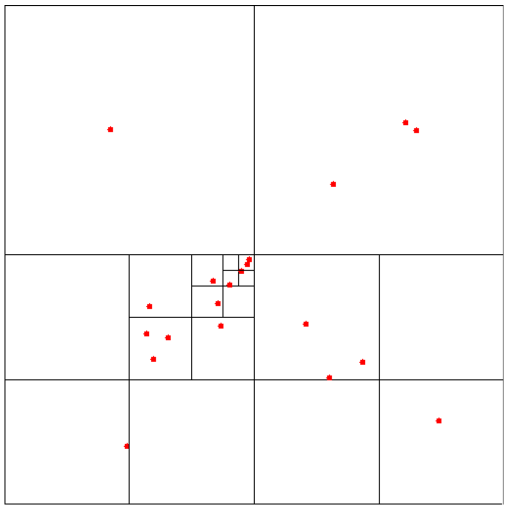

Figure 2: Quadtree partitioning where each partition contains at most three points.

Because of the way 2-D space is partitioned in a quadtree, elements at the same location are always placed together in the same partition. This is extremely helpful in collision detection. 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.

Caution! Note that, in order to use quadtrees for collision detection, we actually want to think about storing parallelograms instead of line segments. Imagine a line is 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 a line during the next time step.

Write-up 2: Figure 2 illustrates that quadtrees can readily handle points, and Figure 3 suggests that line segments can at least sometimes be handled by quadtrees. However, in Figure 4 one of the line segments cannot fit into any of the partitions. Is this a problem for the quadtree? What can you do about it? Can you still use quadtrees to effectively speed up collection detection? *Hint:* Do all line segments need to be stored in leaf nodes?

Using a quadtree, rewrite collision detection to be much more efficient. Note that 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 N. 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.

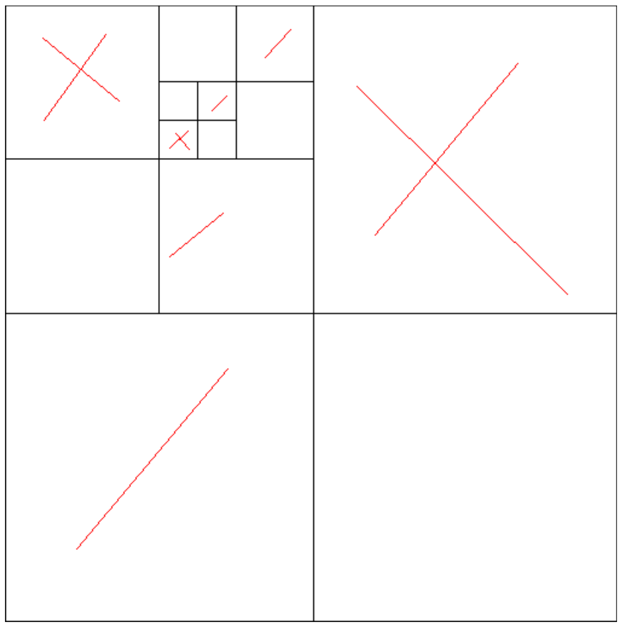

Figure 3: Quadtree storing line segments.

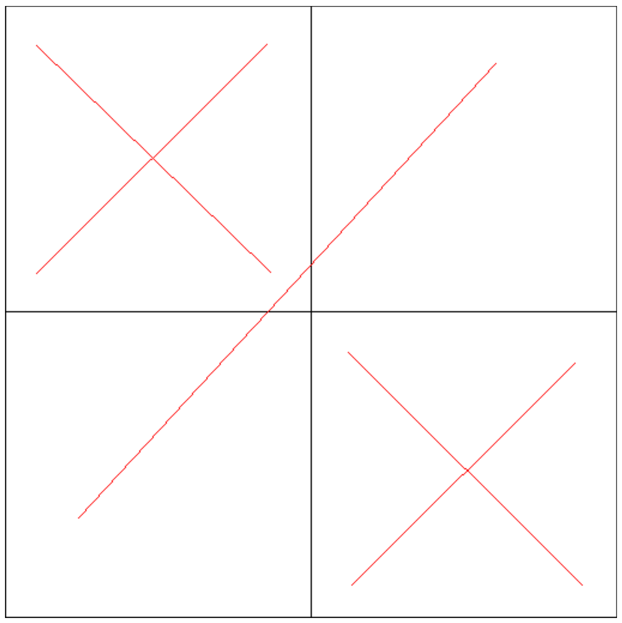

Figure 4: Broken quadtree?

Write-up 3: Report the number of collisions detected during the screensaver's execution. This should be the same as the number you recorded for the unmodified code.

Write-up 4: Measure and report how long it takes for the program to execute with the more efficient collision detection. Make sure to run the program without the graphics flag to get accurate performance results. Compare with the original runtime. Was the speedup what you expected?

Write-up 5: Describe any design decisions you made while rewriting collision detection to use a quadtree. For example, once you built a quadtree, how did you use it to extract collisions? How did you store line segments in each quadtree node?

Write-up 6: Vary the maximum number of elements N a quadtree node can store before it needs to be subdivided. Measure the performance impact that this has on the runtime. Briefly comment on why this did or did not have a performance impact.

2.4 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 in each time step. This is an expensive calculation that is unnecessarily repeated in each time step. 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 this does make things simple, it is a bit wasteful. You can try finding an effective way to update the quadtree so that you don't need to destroy it.

- Our method for testing if two lines intersect is fairly efficient. However, you could try finding a more efficient way of testing if two lines intersect.

Write-up 7: Describe the optimizations that you tried and how well they work.

3 Parallelization (Final Submission)

For the final submission you may use parallelization to speed up execution. However, for the beta submission we ask that you hand in a serial version.

In this section we tell you what is expected for the final submission. However, we are still putting together the brand new Cilk tool suite you will get to use, and have therefore left out the commands to use Cilksan and Cilkscale. For the final part of the project we will update this document with those commands.

3.1 Profiling the Serial Program

Before beginning, 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. To do this, build a `gprof`-enabled version of the screensaver binary by typing `make screensaver.prof` (run `make clean` first). Run it for 1000 frames using `./screensaver.prof 1000` and then run `gprof ./screensaver.prof` to see the profiling statistics.

For more information about gprof:

```
https://sourceware.org/binutils/docs/gprof/
```

Write-up 8: Examine the `gprof` results, which are ordered by self seconds. This gives you the amount of time spent in each function. The top of this list gives you the names of functions that, if parallelized, could offer significant performance improvements to the running time of the application as a whole. List the top six functions on the list.

3.2 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.

3.3 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.

Write-up 9: Describe the changes you have made and 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.

3.4 Profiling the Parallel Code

Write-up 10: 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 quadtree (such as the maximum depth and maximum number of nodes per quadrant), as well as any other spawn cut offs you have used in your code. What is the maximum amount of parallelism you can achieve?

3.5 Performance Tuning

Write-up 11: Tune your code so that it executes as fast as possible when running with 8 workers. Describe any decisions, trade-offs, and further optimizations that you made. To tell the Cilk Plus runtime that you want 8 workers, set the `CILK_NWORKERS` environment flag. E.g. `cqrun "CILK_NWORKERS=8 ./screensaver 4000`.

4 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 and input suite we set after the beta submission. You can run a sample benchmark on the cloud compute nodes using the command

```
$ cqrun ./screensaver 4000
```

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.

• *Addressing MITPOSSE comments:* The MITPOSSE will give you feedback on `https://github.mit.edu` 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.

• *Write-up:* You are required to submit (on LMOD) a group write-up discussing the work that you performed. You should answer all questions asked in this handout and address the following:

- Provide a clear and concise description of your parallelization strategy and implementation.
- Discuss any experimentation and optimizations performed.
- Justify the choices you made.
- Discuss what you decided to abandon and why.
- Provide a breakdown of who did what work.
- Briefly comment on meeting with your MITPOSSE mentors.

You will probably find it easiest to update your *Beta* write-up for your *Final* submission.

• *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; and, 2. the input is aesthetically interesting — i.e. it is entertaining to watch your screensaver run on your input.

Although this is only a small portion of your grade, there will be a prize for the best team's submission

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

	Beta	*Final*
Performance (of correct code)	34%	44%
Addressing MITPOSSE comments		10%
Write-up	5%	5%
Screensaver Aesthetic Test Inputs	1%	1%
Total	40%	60%

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 each team member makes a substantial contribution to the project.