Project 4-2

Parallel Programming with Cilk II

Out: Thursday, 29 Oct 2009
Due: Friday 10AM, 13 Nov 2009

In this project you will optimize a graphical screensaver program using the Cilk parallel programming environment.

Introduction

With the recent revolution in mobile computing prompted by Apple's iPhone and Google's Android, Snailspeed is actively trying to enter the mobile computing market. Alyssa's boss is starting a new project with the goal of developing a screensaver for a mobile phone.

Since mobile processor trends indicate that mobile processors will soon switch to multi-core architectures, Alyssa's boss wants her to develop a screensaver that is parallelized and multi-core ready. Your job is to help Alyssa with this task.

1 Optimizing Collision Detection

In this part of the project, you will be optimizing a simple screensaver. This screensaver consists of a 2D virtual environment filled with colored line segments. These line segments bounce around with some simple physics.

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 fairly expensive as it requires $\theta(n^2)$ collision tests.

Your job is to optimize how collision detection works. You will be using quadtrees to improve the efficiency of collision detection. Note that you will not be using Cilk for this part of the project. The screensaver will be parallelized using Cilk in the next part.

Running the Screensaver

The screensaver can be executed both with and without a graphical display.

```
Usage: ./Line.64 [-g] [-i] <numFrames>
  -g : show graphics
  -i : show first image only (ignore numFrames)
```

NOTE: While collecting performance data, make sure to execute the screensaver without graphical display. This will lead to more accurate performance results.

Quadtrees

Quadtrees are trees with two important properties. First, quadtrees are used to partition two-dimensional spaces. A quadtree partitions a two- dimensional space by recursively subdividing it into four quadrants. Figure 1 illustrates how a quadtree might partition a two-dimensional space.

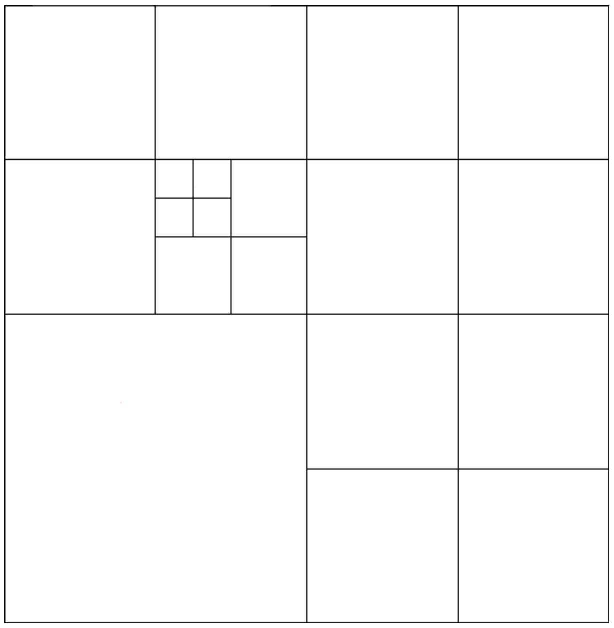

Figure 1: Two-dimensional space partitioned by a quadtree

Second, given elements to store in a 2D space, quadtrees attempt to partition the 2D space such that each partition contains at most N elements. Partitions containing more than N elements are recursively subdivided into four quadrants until each partition contains at most N elements.

Figure 2 illustrates how subdivisions are used to achieve the desired N elements per partition. In areas where elements are dense, more subdivisions are introduced, ensuring that each partition contains at most N elements. In Figure 2, each partition is limited to containing only three points ($N = 3$).

Implementing a quadtree is fairly simple. A quadtree simply consists of quadtree nodes. Each node represents a region in 2D space. Each node stores information on the region that it represents. Each node also stores all of the elements that fall within its region. Initially, the quadtree starts out with only the root node. The root node represents the entire 2D space. As such, the root node initially stores all of the elements in the 2D space.

For each leaf containing more than N elements, the leaf's region is subdivided into four quadrants. The leaf creates four children, one for each of the four quadrants. The elements stored in the leaf are then redistributed to the four children. The leaf becomes an inner node and the four children become new leaves. This process is repeated for any leaf that contains more than N elements. Note that this also applies to the root node since it is the very first leaf in the tree.

This process of subdivision grows the quadtree downwards until all of the leaves contain at most N elements. The final partitioning of the 2D space is represented by the regions in the leaves of the quadtree. Because all leaves contain at most N elements by the end of recursive subdivision, the final partitioning of 2D space contains at most N elements in each partition.

Note that upon subdivision, all of the elements within a node are (usually) passed down to the node's four children. This actually collects all of the elements at the bottom of the tree, in the leaves. This makes sense as the leaves represent the final partitions for the 2D space. Important Message: The most important thing to realize about quadtrees is that given the way 2D space is partitioned, spatially-related elements are 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

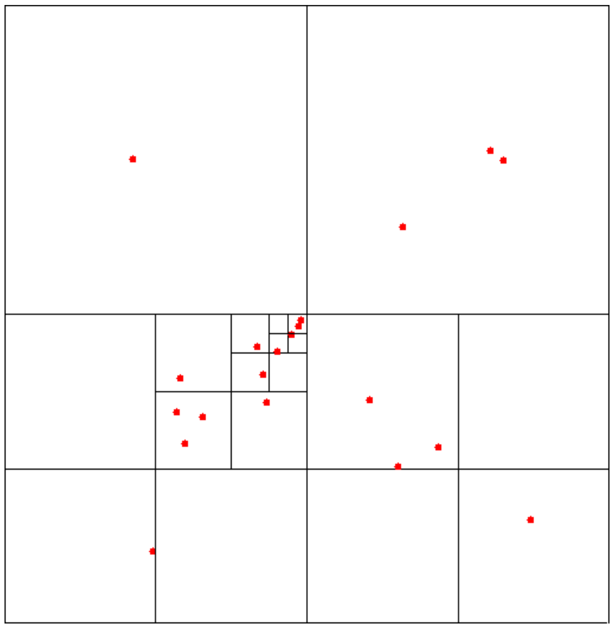

Figure 2: Quadtree partitioning where each partition contains at most three points

partition. Corresponding, if two elements are in different quadtree partitions, they cannot possibly collide.

With this knowledge, you can use quadtrees to limit the number of interactions that you need to consider. For example, to look for colliding line segments, you only need to look at pairs of line segments within the same partition. You can ignore all of the line segments that are in a different partition. While you still need to do $\theta(n^2)$ collection tests within the partition, n is much smaller since you're considering only the line segments in one partition instead of all the line segments simultaneously.

Exercises

1.1

Run the unmodified screensaver on a `cagnode` machine using PNQ and report its runtime for 4000 frames. Also report the number of collisions detected during the screensaver's execution. Make sure to be run the program without the graphics flag to get accurate performance results.

1.2

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 can't 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?

1.3

Using a quadtree, rewrite collision detection to be much more efficient. Note that you should use `intersect()` from `IntersectionDetection.cpp` to test if two line segments will intersect in the next

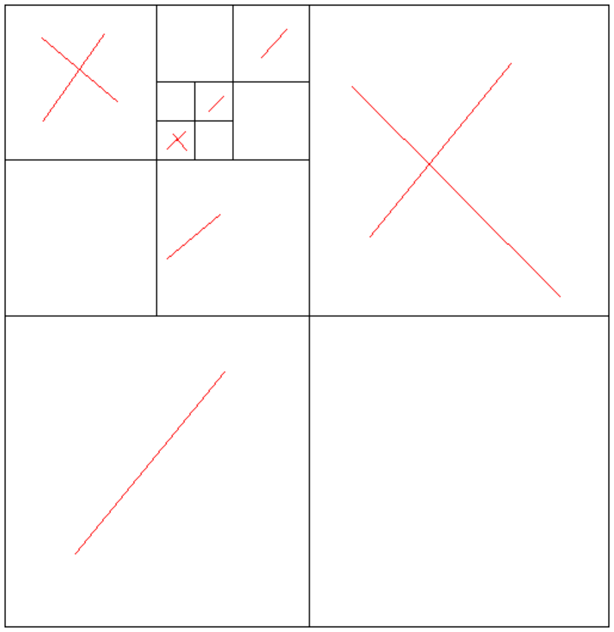
Figure 3: Quadtree storing line segments

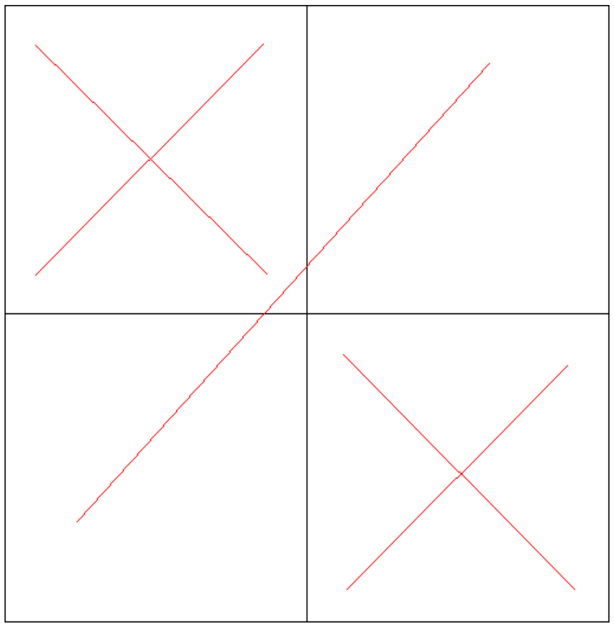
Figure 4: Broken quadtree?

time step.

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.

Measure and report how long it takes for the program to execute with the more efficient collision detection. Compare with the original runtime. Was the speedup what you expected?

Record the number of collisions detected during the screensaver's execution. These numbers should be close to the numbers you recorded for the unmodified code. Since the simulation is sensitive to the order in which collisions are resolved and the calculations are done in floating point, your new results may not be exactly the same as the original version's.

1.4

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?

1.5

Vary the maximum number of elements 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.

1.6

Implement a maximum depth for the quadtree. Vary the maximum depth and measure the performance impact this has on the runtime. Briefly comment on why this did or did not have a performance impact.

1.7

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

- 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.

2 Parallelization

2.1

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 large region of code that comprises of a large percentage of the total execution time, yet still offers a large degree of coarse grained parallelism. To do this, build a version of the binary without Cilk or graphics support by typing `pnqsub make vtune`, then profile the program using the VTune call graph profiling activity. You must use PNQ to compile the binary so that VTune can locate the correct shared objects.
Hint: When selecting the program in VTune, be sure to set the working directory so that the application can find its input file `line.in`. Unless you set a small number of frames to compute, the activity will take a long time – you may stop the activity early after 20 seconds or so.

Once profiled, examine the results ordered by `Total Time`. Collapse all of the modules other than `Line.64.vt` in the results window. This gives you the amount of time spent in each function, including any time spent in called functions. 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 in the `Line.64.vt` module.

Unfortunately, since many functions do not contain code that can be executed concurrently, not all functions are good candidates for parallelization.

2.2

As it turns out, the current code does not have many hot functions that contain lots of parallelism. At first glance, in would appear that `detectIntersections` is both frequently executed and contains lots of parallelism. However, a significant problem is caused by the fact that `collisionSolver` is called immediately after it is determined that two lines will intersect in the next time step. Because `collisionSolver` updates the velocities of each of the two lines, the lines may now be susceptible to further intersections with other lines and thus further updates to their velocities. Since these updates are not commutative (or even associative), they cannot be executed in parallel without changing the semantics of the program.

Fortunately, the code represents only one of many approximations to simulating the interactions between floating lines. As is often common when parallelizing an application, you will need to modify the semantics of the program slightly in order to expose some parallelism, while still meeting the specifications of the application (in this case, to create a semi-realistic simulation of lines bouncing off of each other).

To do this, you can delay the calls to `collisionSolver` until after you obtain a list of all lines that will intersect with each other during the next time step. In this way, you will only need to execute the calls to `collisionSolver` sequentially, freeing you to execute parts of `detectIntersections` in parallel.

Make this change to your sequential program. You should store the list of intersections in a STL list of type `list<IntersectionInfo>`, where IntersectionInfo a struct defined as:

```
extern "C++" {
  struct IntersectionInfo {
    Line *l1;
    Line *l2;
    IntersectionType intersectionType;

    IntersectionInfo(Line *l1, Line *l2, IntersectionType
                     intersectionType) {
      this->l1 = l1;
```

```
        this->l2 = l2;
        this->intersectionType = intersectionType;
      }
    };
  }
```

Be sure to include the `extern "C++"` block as it will be necessary for Cilk later. Now simply add to the list whenever you detect a future intersection between two lines, and call `collisionSolver` on each item in the list once all intersections have been found.

Execute the graphical version of the simulation to verify that the new semantics of the application continue to meet the specifications of the screensaver program (the lines continue to bounce off of each other in a semi-realistic manner).

2.3

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 data race. One solution is to protect the accesses to the list using a lock; however, this solution is undesirable because 1) the lock will quickly become contended, limiting scalability, and 2) the order of items in the list will become non-deterministic. A better approach is to use Cilk reducers.

Cilk solves the problem of accumulating results by providing a unique programming construct called a reducer. Conceptually, a reducer is a variable that can be safely used by multiple Cilk strands running in parallel. When multiple Cilk strands access a reducer, each strand is given a private copy of the reducer. Each strand can update its private copy as much as it wants. When strands are synchronized (using `cilk_sync`), the private copies are merged together into a single variable.

Because each strand has a private copy of the reducer, the possibility of a race is eliminated. In addition, because private copies are merged together during synchronization, results from multiple strands can be accumulated and merged in the same order as when the program is executed sequentially.

Reducers are each defined with the following operations:

- Identity - the default value for a reducer
- Update - one or more operations which updates a reducer's stored value
- Reduce - the operation which merges two reducers when two strands join

Cilk comes with a number of predefined reducers. For example, Cilk defines a summation reducer. The default value for the reducer is 0. Cilk strands can update private copies of the reducer using arithmetic operators. Private copies are merged during synchronization via summation.

Another example of a Cilk reducer is the list concatenation reducer. The default value for the reducer is an empty list. Cilk strands can update private copies of the reducer by appending to the list. Private copies are merged by appending lists to each other.

Reducers have a number of attractive properties:

- Multiple strands can access a reducer without races.
- Reducers are shared without the need for locks, which would normally result in loss of parallelism.
- Reducers can be used without significantly restructuring existing code.

- Defined and used correctly, reducers retain serial semantics. The result of a Cilk++ program that uses reducers is the same as the serial version, and the result does not depend on the number of processors or how the workers are scheduled.
- Reducers are implemented very efficiently, incurring little or no runtime overhead.

Usage Example: Assume we wish to traverse an array of objects, performing an operation on each object and accumulating the result of the operation into an STL list variable:

```
int compute(const X& v);
int test()
{
    const std::size_t ARRAY_SIZE = 1000000;
    extern X myArray[ARRAY_SIZE];
    // ...
    std::list<int> result;
    for (std::size_t i = 0; i < ARRAY_SIZE; ++i) {
       result.push_back(compute(myArray[i]));
    }
    std::cout << "The result is: ";
    for (std::list<int>::iterator i = result.begin(); i != result.end();
         ++i) {
       std::cout << *i << " " << std::endl;
    }
    return 0;
}
```

Changing the `for` to a `cilk_for` will cause the loop to run in parallel, but doing so will create a data race on the result list. The race is solved by changing `result` to a `reducer_list_append` hyperobject:

```
 int compute(const X& v);
 int test()
 {
     const std::size_t ARRAY_SIZE = 1000000;
     extern X myArray[ARRAY_SIZE];
     // ...
     cilk::hyperobject<cilk::reducer_list_append<int> > result;
     cilk_for (std::size_t i = 0; i < ARRAY_SIZE; ++i) {
        result().push_back(compute(myArray[i]));
     }
     std::cout << "The result is: ";
    const std::list &r = result().get_value();
    for (std::list<int>::iterator i = r.begin(); i != r.end(); ++i) {
        std::cout << *i << " " << std::endl;
    }
    return 0;
 }
```

Replace your STL list with the Cilk++ list appending reducer of type:

`cilk::hyperobject<cilk::reducer_list_append<IntersectionInfo> >`
and update the accesses to this object accordingly. Verify that the number of intersections remains the same as before.

Note: The list append reducer does not have an simple way of deleting all the elements in the list. Thus, you will have to create a new reducer for every time step.

2.4

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 intersection detection code so that it performs a recursive depth-first-search through your quad tree.

You will find that `cilk_for` does not work with loops using STL iterators. To parallelize these, you can manually code up the divide and conquer strategy employed by the Cilk++ complier, or modify the loops to eliminate the iterators. Doing the former can be more desirable as it allows you to control the point at which you would like to quit spawning off new tasks.

Describe the changes you have made and verify that your code performs the same number of collisions when executing in parallel and sequentially (remember, you may specify the number of workers using the `-cilk set worker count=N` command line argument). Additionally, verify that the code is race free by running it through the CilkScreen data race detector.

2.5

Determine the span and work of your parallel code by executing it through CilkView. How much parallelism do you see? Vary the parameters of your quad tree (such as the max depth and max 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?

2.6

Tune your code so that it executes as fast as possible when running with 8 threads. Describe any decisions, trade offs, and further optimizations that you made.

Evaluation

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

- Correctness – To receive full points your code should:
  - Compute collisions correctly. When running in graphical mode, the collisions should look semi-realistic as in the code given to you originally.
  - Yield the same number of collisions when running sequentially and when running with 8 threads.
  - Contain no data races.

  You will receive partial credit for each of these requirements.

- Performance – As with the previous projects, your performance grade will be computed by comparing the performance of your submission to the highest performance achieved by a group in the class during the initial submission.
- Style – You code should be decomposed into functions and classes and use as few global variables as possible. It should have plenty of comments throughout, describing the general parallelization strategy as well as low level details of your optimizations.
- Methodology of Performance Optimizations – You are required to submit a group write up discussing the work that you performed. You should answer all questions asked in this handout and include 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.