Concepts in Multicore Programming February 9, 2010
Massachusetts Institute of Technology 6.884
Charles E. Leiserson Handout 6

Lab 1c: Collision detection

In this lab, you will use Cilk++ to parallelize a piece of real-world software provided to us by SolidWorks Corporation, a major vendor of computer-aided design (CAD) software. You will compare different ways to deal with race conditions, using locks and reducers. Finally, you will write a reducer from scratch and reason about its performance characteristics. The write-up for this portion of the lab, together with the write-ups for Lab 1a and Lab 1b, is due on Friday, February 12.

Although you should do all nonoptional steps of this lab, you need not submit documentation demonstrating that you have done each step. You should feel free to investigate what you see as the interesting issues in this lab and to document those. Try not to wander too far from the topic of the lab, however.

In this particular lab, you will implement a reducer from scratch. Although this reducer is fairly simple in concept, there are many interesting issues to deal with in implementation and analysis. Investigating some of these issues and documenting your findings is a satisfactory direction to take this lab.

As with Lab 1a and Lab 1b, we would like you to comment on some aspect of this lab using NB.

Getting started

We have given you a git repository with the code for this portion of the lab. You can obtain this code using the following command:

```
git clone /afs/csail/proj/courses/6.884/spring10/labs/lab1c/ lab1c
```

Once this repository has been cloned, you should see one subdirectory within the new directory `lab1c`, called `collision`.

1 Collision detection

The collision-detection program reads two data files, each of which comprises a model of an assembly of three-dimensional parts. The program then compares the models and outputs another data file listing all the parts (if any) from one assembly that collide with (intersect in three-space with) parts from the other assembly.

Each model is stored in a ternary (3-way) tree in which the collection of parts is grouped into ever-smaller partitions of 3D space. At each node of the tree, space is divided into two child partitions. The left child holds the collection of parts that are located wholly in the first partition, the right child holds the collection of parts that are located wholly in the second partition, and the middle child holds the collection of parts that straddle the two partitions.

The structure of the collision-detection search is a recursive in-order traversal through these ternary trees. Traversal starts at the root of both trees. If the bounding box of one tree does not intersect the bounding box of the other tree, then there cannot be any collisions, and the search ends. Otherwise, we visit each child node of the smaller tree and repeat the comparison. At the leaves of both trees, individual parts are compared and collisions are prepended to a linked list. In this way, both trees are traversed towards the leaves while branches where the bounding boxes do not intersect are quickly pruned. This algorithm is quite efficient, especially if the three children of each node are traversed in parallel. The Cilk++ work-stealing scheduler is highly adept at handling this kind of unbalanced parallelism.

An interesting feature of this implementation of this algorithm is that the ternary tree is constructed lazily, during the collision-detection process. In the full CAD application, the cost of building the trees is amortized over a large number of collision-detection invocations. In our lab, however, we are comparing only two models. The cost of building the ternary tree and computing the bounding boxes actually swamps the cost of traversing the tree and finding the collisions. Little speedup would therefore be noticeable when we parallelized the collision-detection code. For this reason, the `cilk_main` function calls the collision-detection code twice: once to build the trees with the performance timers turned off, then again with the timers turned on. This approach gives us a much more accurate view of the speedup obtained through parallelization.

2 Initial parallelization

In the `collision` subdirectory, you will find the source code for the collision-detection program. This industrial code is essentially unmodified from what SolidWorks gave us (and is used with their permission). The program is invoked with three command-line arguments: two data files containing models and one output file to receive the list of collisions. The heart of the recursive tree traversal is the function `aabbTreeNode::collideWith`, which operates in two modes:

1. If the `listOut` global pointer variable is set to `NULL`, then `collideWith` returns only a boolean value indicating the presence or absence of a collision, stopping at the first collision. We call this mode "existential mode".
2. If the `listOut` global pointer variable contains the address of a list object, then `collideWith` prepends all collisions to that list as they are detected. We call this mode "list mode".

The main program calls `collideWith` in the existential mode if the output file is omitted from the command line. Otherwise it calls it in list mode.

(a) Compile the project in the `collision` directory. Run it in both existential and list mode on a pair of small models as follows:

```
$ collision camera1.wrl camera2.wrl
$ collision camera1.wrl camera2.wrl cameras.serial
```

(b) In `aabbtree.cilk`, find function `aabbTreeNode::collideWith` (near the end of the file). Toward the end of the function, you will see three recursive calls to `nodeIn->collideWith`. (Please ignore similarly named functions, such as `iBBox.collidesWith`, which are not part of the recursion.) These three recursive calls are structured sequentially so that the result of the first call partially determines whether the second call is performed. Rewrite this logic so that this sequential behavior is retained in existential mode (`listOut == NULL`), but where all three calls are made unconditionally in list mode (`listOut != NULL`).

(c) Change `aabbTreeNode::collideWith` so that the three recursive calls are made in parallel when in list mode. **Note: If you run this program on multiple cores, it may crash!** Double-check that the program still works on a single core:

```
$ collision -cilk_set_worker_count=1 camera1.wrl camera2.wrl
$ collision -cilk_set_worker_count=1 camera1.wrl camera2.wrl
cameras.onecore
```

(d) Run the program within Cilkscreen on a couple of small models:

```
$ cilkscreen collision camera1.wrl camera2.wrl
$ cilkscreen collision camera1.wrl camera2.wrl
cameras.parallel
```

Do both the existential and list modes have races, or just one of them? Identify the cause of the race(s). We will address this problem in the next steps.

3 Resolving the race with locks

The collision-detection logic is parallel because the program can traverse multiple subtrees simultaneously. Each parallel call tries to update the same global list, however, resulting in a race somewhere in the list code. One possible solution is to restructure the code, but that is not always feasible for programs in a large codebase. As we did for the many-bodies lab, we can resolve this race using mutex locks, with the expectation of low contention. The use of locks here is less problematic than it was for the N-bodies case for two reasons:

1. We need only one mutex for the whole program, rather than one per item, as we did in the many-bodies case.
2. The expectation is that collisions are relatively rare and, therefore, that acquiring the mutex is relatively rare. Recall that acquiring a mutex can be a fairly expensive operation, even in the absence of contention.

Although we will ultimately use a reducer instead of a lock to resolve the race on the list, it is instructive to implement a locking solution first and compare the approaches.

(e) In the file `aabbtree.h`, add `#include <cilk_mutex.h>`. Find the line that declares `listOut`, and add a declaration for a `cilk::mutex` variable called `list_mutex`.

(f) Find the line in `aabbtree.cilk` where `push_front` is called on `(*listOut)`. Put a `list_mutex.lock();` and a `list_mutex.unlock();` before and after the `push_front`, respectively. Confirm that the program now runs without races on the small models, again sending the output to `cameras.parallel`.

(g) Compare the output from the serial execution (`cameras.serial`) with the results of the parallel execution (`cameras.parallel`) using `diff` on Linux. Why might the outputs be different?

4 Resolving the race with a reducer

Locks resolve data races, but often leave nondeterminism in the code. This nondeterminism might be benign (e.g., if you don't care about the order in which collisions are added to the list), or it could be a bug (e.g., if you do care about the order).

We can preserve the order of the output list by using a `reducer_list_prepend` hyperobject as a drop-in replacement for `std::list` in code that prepends to a list using `push_front`. Parallel subcomputations can call `push_front` on a `reducer_list_prepend` without locks and without causing data races. When the parallel use of the reducer is complete, the `get_value` function returns a reference to an `std::list` containing the same values in the same order as would have occurred in the serial execution. Thus, the result of this exercise is a collision-detection program that runs in parallel while preserving serial semantics.

(h) Remove the use of mutexes that were added in the previous steps. (A search for "mutex" should find all occurrences easily.) Alternatively, you can copy source files from the `solutions/collision_parallel` directory.

(i) Add "`#include <reducer_list.h>`" to `aabbtree.h`. Change the data type `aabbTreeCollisionList` from `std::list<aabbFacetPair>` to `cilk::reducer_list_prepend<aabbFacetPair>`.

(j) In `MultiCoreCollisionDetection.cilk`, find the place where `outputToBV` is called, and change the use of `listOut` to `listOut->get_reference()`. The use of `push_front` should not change.

(k) Recompiling and running the program at this point fails. The reason for this failure is because the function `collideWith()` in `meshprocess.cilk` calls the `empty()` method on the *reducer* `listOut`, which has no such method. Why doesn't the `reducer_list` provide this method?

(Because we are only examining one pair of objects for collision, it is simple to resolve this problem for this lab. This problem is less obviously solved, however, if we were examining in parallel more than one pair of objects for collisions. Because the `collideWith()` function in `meshprocess.cilk` may be used in either situation, consider solving this problem for the more difficult case.)

There are several ways to fix this problem. One way is to remove this check from the `collideWith` method in `meshprocess.cilk`, changing the semantics of function in the process. (Note that this method, like much industry code, lacks complete documentation.) Another way is to change the use of `listOut` to `listOut->get_value()` in this place. Since querying the state of a reducer is generally dangerous, why is it okay to examine the reducer's value in this case? Choose some way of dealing with this problem, and justify your choice.

(l) Compile and run the program on the same small models as before, saving the output into a new file:

```
$ cilkscreen collision camera1.wrl camera2.wrl
cameras.reducer
```

Compare the output file of the reducer version (`cameras.reducer`) to the output file of the serial version (`cameras.serial`). Are they the same?

(m) Use Cilkscreen to confirm that the program is now race free. Use Cilkview to test its performance characteristics when colliding a large model with itself:

```
$ cilkview -trials one 2 collision highhitch1.wrl
highhitch1.wrl highhitch.out
```

4.1 Writing a reducer from scratch

The parallelization of this program using a `reducer_list` may not give the best performance. The `reducer_list` reducer uses a linked list as its underlying data structure. A list is a convenient data structure to implement as a reducer, because two linked lists may be combined in constant time, and therefore the `reduce()` operation takes constant time. Walking a linked list can have poor performance in practice, however, because there is little spatial locality. Moreover, a linked list is hard to manipulate in parallel.

To improve performance, we would like to use an array or vector as the reducer's underlying data structure. Combining two vectors is an expensive operation, however, because it requires copying every element to a new chunk of memory. We can overcome this performance barrier by combining the concepts of a linked list and a vector into a data structure that we shall call a ***hypervector***.

The `hypervector` class is a monoid that supports the following operations. Creating an identity `hypervector` effectively produces an empty vector. An element may be appended to a `hypervector` using a `push_back()` method. A `hypervector` *right* may be appended to the end of a `hypervector` *left* using an associative `concatenate()` method, which destroys *right*. Finally a `hypervector` may be converted into a normal vector using a `get_vec()` method.

A simple implementation of a `hypervector` is a linked list of vectors. The identity `hypervector` contains an empty vector and a NULL pointer to the next `hypervector` in the list. A `hypervector` *right* may be appended in constant time to a `hypervector` *left* using an operation similar to appending linked lists. Finally, the `get_vec()` operation can convert this linked list of vectors into a single vector by walking the linked list and copying the contents of each vector in the list into a newly created vector. We can avoid some slowdown on this last step by parallelizing the `get_vec()` operation.

(n) We have given you most of the code to implement the basic vector reducer in `reducer_vector.h`, but the implementation of the underlying `hypervector` is missing. Complete the implementation of `hypervector`, and ensure that your implementation passes all tests in the `hypervector_test.cilk` test file.

To implement the `get_vec()` method, you must fill in the private `parallel_get_vec()` member function. In the current version of Cilk++, a reducer must be implemented as a C++ class, and its `reduce()` operation must be serial. We have provided the necessary declarations and calls to embed a Cilk++ routine in this C++ class to allow `get_vec()` to be implemented in parallel, which has created the less-obvious location for implementing `get_vec()`.

(*Note:* We have found issues using this reducer with the Miser memory manager. For now, please do not compile your code with `-lmiser`.)

(o) The `reducer_basic_vector` implemented in `reducer_vector.h` is not immediately useful in our application. The `reducer_basic_vector` only supports append-style operations, while `aabbtree.cilk` uses prepend operations to create `listOut`. There are several ways to fix this problem, from modifying the semantics of `get_vec()` in the `hypervector` class to writing a separate (parallel) method to reverse the output of `get_vec()`. Choose a solution, implement it, and justify your choice. Then, using your solution, modify `aabbtree.cilk` and `MultiCoreCollisionDetection.cilk` to use the `reducer_basic_vector` reducer, and examine its performance.

(p) *Ideas for exploration:* What other ways of implementing hypervectors can you come up with? Is a list of vectors the best strategy, or might a tree of vectors work better? What is the trade-off between a fast `reduce()` operation and efficient indexing of the hypervector as an array after it has been computed? Is there a way of making the `get_vec()` operation run in constant time and amortizing the conversion over indexing operations after the fact? What theoretical bounds can be proved?

4.2 Using speculation (optional)

The first thing we did when parallelizing the collision-detection logic was to separate the existential-mode code, which remained serial, and the list mode code, which we made parallel. The main difference between these modes is that the list mode always visits every node in the ternary trees, whereas the existential mode stops executing at the first collision. We can parallelize the existential mode using ***speculative parallelism***, where more work is performed in the parallel case than in the serial case in an effort to expose parallelism. This exercise touches on just a couple of approaches to speculation.

(q) Before you change any code, use Cilkview to benchmark the current version in existential mode when colliding two large models and when colliding one large model against itself:

```
$ cilkview -trials one 2 collision highhitch1.wrl
highhitch2.wrl
$ cilkview -trials one 2 collision highhitch1.wrl
highhitch1.wrl
```

Retain the timing numbers (not just the speedup numbers) for comparison with subsequent versions.

(r) Modify the existential-mode code to store the boolean result of each recursive call to `collideWith` in a separate `bool` variable. Then, remove the conditional logic so that the three recursive calls are performed unconditionally and the return value is computed as the logical OR of the three results. Finally, insert `cilk_spawn`s so that the three recursive calls operate in parallel. The existential mode code should now look similar to the list-mode code (and you might want to merge them back into a single piece of logic). Use Cilkview to test the performance of this version of the program when colliding two large models and when colliding one large model against itself, as we did in the previous step. Do you obtain consistent speedup or slowdown versus the serial version?

As is the case with speculation, the existential-mode logic does more work than is necessary to solve the problem. Our current version of the logic, however, does *so much* more excess work that we have an overall slowdown, even with parallelism. What we would like to do is to terminate the recursion as soon as possible after one of the branches has found an answer. We can do so by introducing a boolean `doneFlag` variable and testing it at strategic places.

(s) In `aabbtree.h`, declare `extern volatile bool doneFlag;` (preferably near where `listOut` is declared). In `aabbtree.cilk`, define `volatile bool doneFlag = false;`. Set the flag to `true` upon seeing a collision (creating a ***benign*** race). Check the flag just before recursing in `collide`, and return immediately if it is `true`. Finally, in the `cilk_main` function in `MultiCoreCollisionDetection.cilk`, set `doneFlag = false` just prior to calling `mesh1->collideWith(mesh2)` (in both places it is called). Rerun your timing tests.

(t) (Optional) Because setting and reading `doneFlag` in parallel is a race, albeit a benign one, the Cilkscreen race detector reports a race on every access to `doneFlag`. To quiet the race detector, we can use a ***fake lock*** for the `doneFlag`. First, include `fake_mutex.h` in `aabbtree.h`, and add a declaration for a `cilk::fake_mutex` variable called `fake_mutex` near the declaration of `doneFlag` in `aabbtree.h`. Next, insert a call to `fake_mutex.lock()` before every access to `doneFlag`, and `fake_mutex.unlock()` afterwards. These calls add no overhead to the compiled code but tell Cilkscreen that the access is safe. (Be careful, because fake locks can also be used to tell Cilkscreen that something is safe even when it is not!) Run the new code in the Cilkscreen race detector (use small models for speed), and confirm that there are no races detected.