CSE 5441 (Fall 2019, Dr. Jones)
POSIX Threads AMR (Lab 2)

Caleb Lehman
lehman.346@osu.edu

October 29, 2019

Summary

For this lab, I modified the previous serial `C` program to perform Adapetive Mesh Refinement (AMR)[1] to work in parallel using the POSIX Threads (`pthreads`) library. I created 4 programs, `disposable`, `persistent`, `disposable_equal_boxes`, and `persistent_equal_boxes`, which employ different strategies for 1) thread creation and destruction and 2) distribution of boxes to threads. I tested these programs on 3 test files with different numbers of threads to compare performance against the serial version and against each other.

As expected, the *persistent threads* model outperformed the *disposable threads* model, most likely due to less overhead from not destroying and recreating threads each iteration. All parallel programs outperformed the serial version (at least for optimal choices of number of threads).

- The optimal number of threads for the smaller test case (`testgrid_400_12206`) was 6 for both programs.
- The optimal number of threads for the larger test case (`testgrid_1000_296793`) was ∼20 for both programs.
- At the optimal numbers of threads, the `disposable` program was 1.9x faster than the serial program on the small test case and 9.2x faster on the large test case.
- At the optimal numbers of threads, the `persistent` program was 2.9x faster than the serial program on the small test case and 10.9x faster on the large test case.
- The different box-distribution strategies didn't make much of a difference, so I created a third test case, `testgrid_512_196576` to emphasize disparities in the number of neighbors handled by each thread. Even in this case, there wasn't much difference (see the Test Files section for small discussion on this).

I used all 4 timing methods from lab 1 to collect timing data. All the timing results I show in this report were collected from `clock_gettime`. Of particular note is that the `clock` function from the `"time.h"` header is not particulary useful for our purposes, as it reports CPU time, not wall time. As a result, it consistently returns a significantly longer time than the actual time the program took to run.

[1] See previous lab or project descriptions for details about AMR computation.

Pseudocodes

This section contains rough pseudocodes for the (`*_equal_boxes` versions of the) required programs. The pseudocode for the main `disposable`/`persistent` programs is not listed, but is equivalent except for the computation of *range* being slightly more involved in order to balance the number of neighbors handled by each thread.

Disposable Threads Model

Given α, ε, a description of grid-aligned boxes, initial Domain Specific Values (DSV), and a number of threads, the rough pseudocode for my implementation following the *disposable threads* model is as follows:

```
 1: procedure DISPOSABLE(α, ε, N, Boxes, InitialDSV, nthreads)
 2:     DSV ← InitialDSV, DSV′ ← InitialDSV
 3:     while (max DSV − min DSV) / max DSV > ε do
 4:         Create nthreads threads
 5:         for all threads do
 6:             WORKER(α, ε, N, Boxes, InitialDSV, nthreads)
 7:         end for
 8:         Join and destroy all threads
 9:         DSV ← DSV′
10:     end while
11: end procedure
12: procedure WORKER(α, ε, N, Boxes, InitialDSV, nthreads)
13:     tid ← current thread number
14:     for all i ∈ [tid · N/nthreads, (tid + 1) · N/nthreads] do
15:         DSV′[i] ← (1 − α) · DSV[i] + α · Σ_{j∈nhbr} DSV[j] · OVERLAP(i, j, Boxes)
16:     end for
17: end procedure
```

To summarize, under this model, threads are created at the beginning of each iteration and execute updates in parallel. Then the threads are joined and destroyed and the updates are committed.

Persistent Threads Model

Given α, ε, a description of grid-aligned boxes, initial Domain Specific Values (DSV), and a number of threads, the rough pseudocode for my implementation following the *persistent threads* model is as follows:

```
 1: procedure PERSISTENT(α, ε, N, Boxes, InitialDSV, nthreads)
 2:     DSV ← InitialDSV, DSV′ ← InitialDSV
 3:     Create nthreads threads to execute WORKER(α, ε, N, Boxes, DSV, DSV′, nthreads)
 4:     Join and destroy all threads
 5: end procedure
 6: procedure WORKER(α, ε, N, Boxes, DSV, DSV′, nthreads)
 7:     tid ← current thread number
 8:     range ← [tid · N/nthreads, (tid + 1) · N/nthreads]
 9:     while (max DSV − min DSV) / max DSV > ε do
10:         for all i ∈ range do
11:             DSV′[i] ← (1 − α) · DSV[i] + α · Σ_{j∈nhbr} DSV[j] · OVERLAP(i, j, Boxes)
12:         end for
13:         BARRIER to sync all threads
14:         DSV[range] ← DSV′[range]
15:         BARRIER to sync all threads
16:     end while
17: end procedure
```

To summarize, under this model, threads are created at the beginning of the program and destroyed at the end. The threads run in parallel, updating each of their blocks of DSVs. The threads are synchronized twice per iteration: 1) after computing the updates and 2) after commiting updates.

Tests

Environment

The program was developed and tested on a single, 40-core node of the Pitzer cluster at the Ohio Supercomputer Center.

For development, I loaded the `intel/18.0.3` module, which allowed the program to be compiled with version 18.0.3 of the `icc` C-compiler. I linked against the `librt` and `libpthread` libraries.

For testing, I loaded the `python/3.6-conda5.2` module, which loads a python environment with the `NumPy`, `SciPy`, and `Matplotlib` packages, among others.

Timing

I collected timing data using the same 4 methods as the first lab: `time`, `clock`, and `clock_gettime` from the `"time.h"` header; and the `UNIX` utility `time`.

I didn't use the results from the `clock` function from the `"time.h"` header, since it reports CPU time, not wall time. The other methods all returned values within 1 second of each other. The `time` function declared in `"time.h"` returns an integer number of seconds, but the other two methods return with sub-second precision. *For consistency, I used the `clock_gettime` function for all results in this report.*

Test Files

Dr. Jones provided the `testgrid_400_12206` test file. As part of lab 1, I reduced the α (affect rate) and ε parameters until the serial runtime increased into the 3 to 6 minute range. In particular, I selected $\alpha = 0.01$ and $\varepsilon = 0.02$, for which the serial program completed in 261 seconds.

The above approach of changing the α and ϵ parameters allows us to make the programs run longer, but doesn't affect the actual length of each iteration. With only 12206 boxes, each iteration is fairly short (on the order of $\frac{261 \text{ sec}}{1589637} = 0.16$ms per iteration), so it is expected the overhead of synchronizing threads would inhibit parallelizing beyond a small number of threads. In order to investigate this, I generated another test file, `testgrid_1000_296793`, which has more boxes and therefore longer iterations.

As mentioned above, in addition to the two required programs, I created two more with different distributions of boxes to threads. I expected the programs which balanced the number of neighbors for which each thread was responsible (`disposable` and `persistent`) to outperform the programs which simply gave equally-sized, contiguous chunks of boxes to each thread the number of boxes (`*_equal_boxes`), since number of neighbors controls how much "work" is done for each box. However, both programs performed similarly on the first two test files. I created a third test file `testgrid_512_196576`, in which different are areas of the grid have different average neighbors per box. Even on this specialized test case, there was no non-trivial performance difference. Reflecting on this comparison, I think attempting to optimize the number of neighbors handled by each thread is not too promising, since the overall average number of neighbors per box in a grid (actually for any planar graph) is always strictly < 6, so the typical grid is not pathalogical enough for it to make a difference.

Some summary statistics for the test files are presented below. Per submission instructions, I didn't include the test files with the code submission, but I can produce them if needed.

test	# rows	# cols	# boxes	mean # neighbors	std. dev. # neighbors
`testgrid_400_12206`	400	400	12206	5.89	1.55
`testgrid_1000_296793`	1000	1000	296793	4.79	1.57
`testgrid_512_196576`	512	640	196576	5.32	18.24

Table 1: Basic statistics for my test files. Note that the file in the second row was engineered to have many boxes and the file in the third row was engineered to have high variance/std. dev.

Results

The output for each test case was consistent across all variations and was as follows:

- `testgrid_400_12206`: 1589637 iterations, $(max, min) = (0.085900, 0.084182)$
- `testgrid_1000_296793`: 51684 iterations, $(max, min) = (0.000000, 0.000000)$
- `testgrid_512_196576`: 63380 iterations, $(max, min) = (0.000000, 0.000000)$

The main timing results are shown in the figure below. I ran the programs for various amounts of threads between 1 and 40. Note that for the provided test file `testgrid_400_12206`, the optimal number of threads was around 6 for each program, while the optimal number of threads was around 20 for the larger test files that I created. This confirms that increasing the number of boxes, thereby increasing the (serial) length of each iteration provides more opportunities for parallelism. The iterations for the small test file are so short that the overheads associated with creating/destroying/synchronizing threads quickly becomes the bottleneck.

Other notes are that `persistent` versions consistently outperformed `disposable` versions, as expected, and that my variations with how boxes were distributed had very little effect. Finally, the parallel versions of the program did outperform the serial version, except for the smallest test case, where, for large number of threads, the overheads outweighed the benefits of parallelism and the parallel programs actually performed worse.

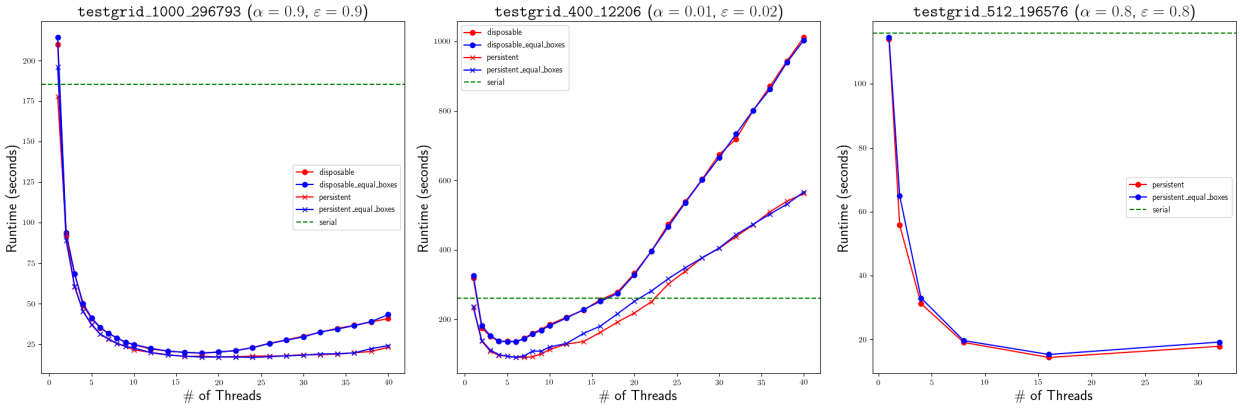

Figure 1: The runtimes of the parallel versions of the program plotted against the number of threads used. Serial runtime is included for comparison. Note that `persistent` consistently outperforms `disposable` and that the `*_equal_boxes` versions perform similarly to their counterparts. Also note that `persistent` switches from `x` markers to circle markers in the rightmost plot.

Project Usage

Building

To build the required `disposable` and `persistent` executables (as well as the `*_equal_boxes`) variations, navigate to the top level of the submitted directory and build as follows:

```
# Ensure that you have icc compiler

$ make
...
$ ls
... disposable disposable_equal_boxes persistent persistent_equal_boxes ...
```

Running

The syntax to run the program is:

```
$ ./[program] [affect-rate] [epsilon] [num-threads] <[test-file]
```

where `program` is one of the built programs and `num-threads` is any positive integer number of threads.