CSE 5441 (Fall 2019, Dr. Jones)

`OpenMP` (Lab 3)

Caleb Lehman
lehman.346@osu.edu

November 4, 2019

Overview

For this lab, I made slight modifications to my serial `C` program to perform Adaptive Mesh Refinement (AMR)[1] to work in parallel using the Intel implementation of the OpenMP API. I created 2 programs, `disposable` and `persistent`, which mirror the disposable and persistent threads models used in the previous lab. In particular, the `disposable` code opens a new parallel region for each iteration, while the `persistent` code has a single parallel region encompassing the entire convergence loop, inside of which, each iteration is divided between the threads.

Tests

Environment

The program was developed and tested on 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.

For testing, I loaded the `python/3.6-conda5.2` module, which loads a python environment with the `NumPy`, `SciPy`, and `Matplotlib` packages, amoung others. `Python` is only necessary for collecting and plotting the data from testing, not for the actual exectuion of the program.

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`.

As with the second lab, 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 I 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. I used values of $(\alpha = 0.9, \varepsilon = 0.9)$ for this test case, since they produced a serial runtime of around 3 minutes.

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

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

Table 1: Basic statistics for the test files.

Results

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

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

The relevant runtimes for the serial, `pthreads`, and `OpenMP` versions on each of the two test cases are plotted in the figure below. Note that the `OpenMP` `persistent` and `disposable` versions perform nearly identically. This suggests that (at least this `Intel` implementation of) `OpenMP` uses some sort of thread pool that creates the threads early in the program and doesn't destroy them until the end, simply assigning threads at the beginning of each parallel section.

For the smaller test case, `OpenMP` outperformed `pthreads`, both yielding consistently lower runtimes, as well as scaling better (improved performance up to 12 threads; 4.6x speed-up over serial version). I suspect this has to do with how optimized the threading mechanisms are for their various tasks. In particular, in the `pthreads` implementation, I had to manually write to code to create and destroy threads, as well as create, pass, and collect the structures/paramaters passed to the threads. While this certainly provides more flexibility that then basic `OpenMP` `pragma` directives, which are limited by comparison, the corresponding operations in the `OpenMP` implementation are almost certainly highly optimized. In this smaller test case, each iteration is relatively short, so the overhead from basic thread management is probably significant for the overall runtime.

For the larger test case on the other hand, each iteration is longer, so the basic thread management overhead is less impactful compared to actual parallelization of the work. In this case, `OpenMP` was outperformed by the `persistent` version of the `pthreads` program. One explanation for this is that my use of `pthreads` was specifically targetted at this particular problem, while the `OpenMP` directives are more general (by design) and therefore may not be as well-suited for this specific computation.

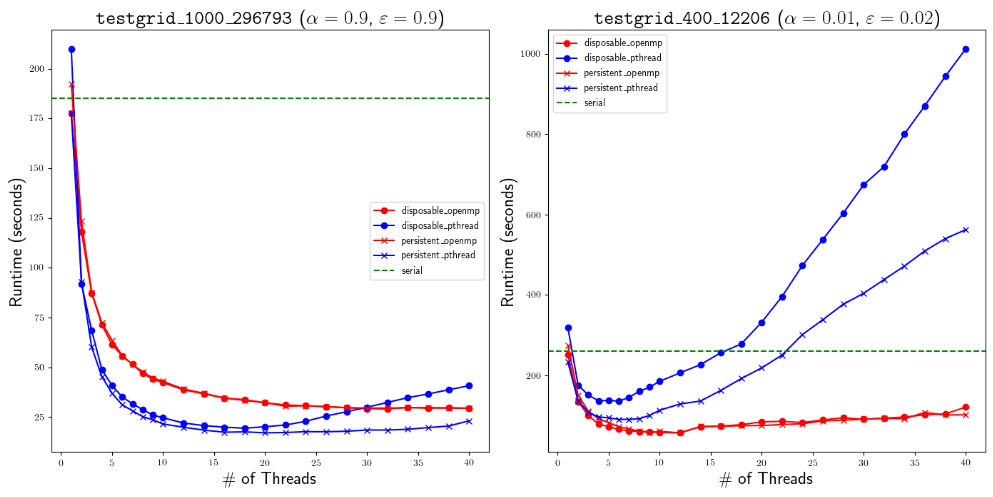

Figure 1: The runtimes of the parallel versions (`pthreads` and `OpenMP`) of the program plotted against the number of threads used. Serial runtime is included for comparison. Note that `OpenMP` `persistent` and `disposable` versions consistently perform nearly identically.

It is worth noting that the OpenMP parallelization was much less intrusive compared to the pthreads parallelization. The persistent OpenMP version required a few pragma directives and some code to compute the workload division for each thread, but used much less new code than the corresponding pthreads version. The disposable OpenMP version required just 3-4 pragmas to parallelize and achieved just as good results as the persistent version. Furthermore, both versions compile[2] and execute correctly as serial programs when compiled without OpenMP.

Finally, since the OpenMP specifications don't have many strict requirements about creating the exact number of threads requested in each parallel region, I included checks in both of the programs to verify that the number of threads was the same as the number requested. For all reasonable requests (between 1 and 40 threads on Pitzer), I never got an unexpected number of threads.

Project Usage

Building

To build the disposable and persistent executables, navigate to the top level of the submitted directory and build as follows:

```
# Ensure that you have icc compiler

$ make
$ ls
... persistent disposable ...
```

Running

The syntax to run the program is:

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

where program is one of {persistent, disposable} and num-threads is positive integer number of threads.

[2]not including harmless warnings about unrecognized pragmas