CSE 5441 (Fall 2019, Dr. Jones)

`MPI` + Hierarchical Parallelism (Lab 5)

Caleb Lehman
lehman.346@osu.edu

Overview

For this lab, I parallelized my serial `C` program to perform Adaptive Mesh Refinement (AMR)[1] using `MPI` and `OpenMP`. In particular, the program launches 5 MPI ranks, 4 of which execute computations in parallel, which utilizes the inter-node parallelism available in a large cluster. Within each of the 4 computational ranks, I used `OpenMP` to exploit intra-node parallelism. I found that I was able to get some additional speed-up from the `OpenMP` threads, but it did not scale to using all of the available cores on the nodes. Full results are in the results section.

Tests

Environment

The program was developed and tested on the Pitzer cluster at the Ohio Supercomputer Center. Note that the original assigment specified that we should use the Owens cluster, however, my first 3 labs were developed and tested on Pitzer, and after bringing this up with Dr. Jones, he permitted me to perform this lab on Pitzer, for comparison's sake.

For development and testing, I loaded the `mvapich2/2.3` and `intel/18.0.3` modules, which allowed the program to be compiled with the `mpicc` compiler, using version 18.0.3 of the `icc` compiler a the default `C` compiler, as well as setting some environment variables pointing to `MPI`-related headers and libraries.

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 several labs: `time`, `clock`, and `clock_gettime` from the `"time.h"` header, and the `UNIX` utility `time`.

As with the second and third labs, I didn't use the results from the `clock` function from the `"time.h"` header, since it reports CPU time, not wall time. In this case, that would have been fine, since the rank 0 `MPI` process doesn't spawn any other threads, but to be consistent with previous labs, *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 $\frac{261 \text{ sec}}{1589637} = 0.16\text{ms}$ per iteration), so I expected the overhead of synchronizing processes and threads would inhibit parallelizing beyond a small number of processes and 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. I used this test case for each of labs 2 and 3, so I also tested with it in this lab.

[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 both test cases was as follows:

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

I obtained the following timing results (current lab results in Figure 1, overall course results in Table 2):

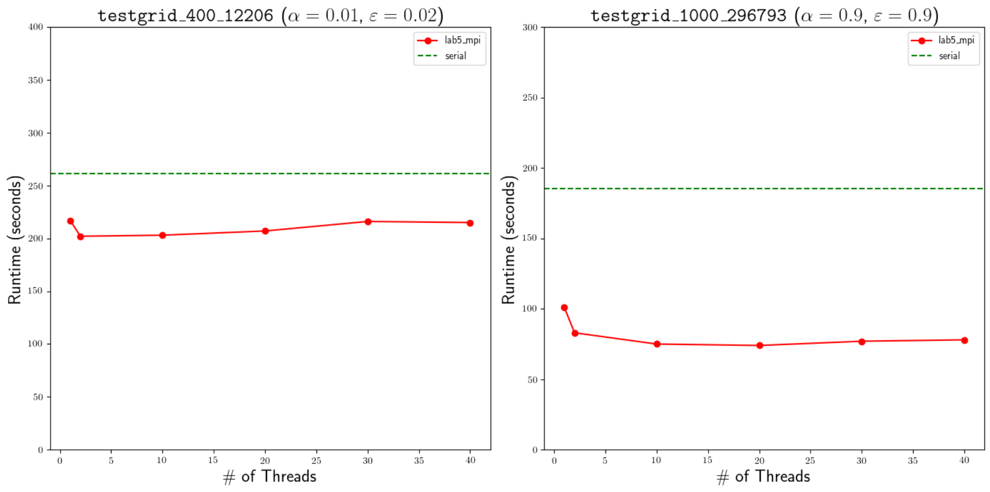

Figure 1: Runtimes for various numbers of threads. The MPI program uses 5 ranks (4 computational ranks).

$\alpha = 0.01$, $\varepsilon = 0.02$	**runtime**	**details**	**speed-up (vs. serial)**
serial	261.09		1x
pthreads	92.08	8 threads	2.84x
OpenMP	57.02	12 threads	4.58x
CUDA	54.03	10 blocks, 32 blocks per thread	N/A
MPI + OpenMP	216.85	5 ranks, 40 threads	1.20x
$\alpha = 0.9$, $\varepsilon = 0.9$	**runtime**	**details**	
serial	185.29		1x
pthreads	17.04	20 threads	10.87x
OpenMP	32.13	20 threads	5.77x
CUDATODO	N/A		
MPI + OpenMP	77.65	5 ranks, 40 threads	2.39x

Table 2: Comparison of optimal timing results from each lab for testgrid_400_12206 (top) and testgrid_1000_296793 (bottom). Recall that the CUDA program was executed on Owens with parameters $\alpha = 0.1$, $\varepsilon = 0.1$ (see lab 4 report for details). Also, the choices for number of ranks and threads for the MPI program are not optimal based on my testing, but are the values requested in the assignment.

Summary (lab 5)

On the larger test case, the `MPI` program got better performance relative to the `serial` version than it did on the small test case, which makes sense given that there is more work to parallelize per iteration in the large test case. However, the `MPI` program didn't show very good scalability in the number of threads spawned by `OpenMP` for either test case. The suggests to me that the bottleneck was the synchronization and communication between the nodes/ranks, since adding more parallelism within a node wasn't changing much. Improvements could probably be made to the program by ensuring that the minimal amount of information is transmitted between the ranks inbetween each iteration.

Summary (all labs)

`pthreads` provided a flexible interface for intra-node parallelism. It gave me complete control over how my threads were created, used, and destroyed. However, it was somewhat intrusive. `pthreads` is most easily compared and contrasted with `OpenMP`, which is also an interface for intra-node parallelism. `OpenMP` was slightly more rigid/less flexible than `pthreads`, but far less intrusive, to the point where my lab could be compiled and executed without linking against any `OpenMP` libraries. Which API yielded better performance, with `OpenMP` seeming to perform better for small test cases, when synchronization was more frequent/iterations were shorter, and `pthreads` performing better for larger test cases.

`CUDA` was much more intrusive than any of the other APIs. It was also a different form of parallelism; where the other APIs used a very general fork-join model of threads, in which the threads basically operated totally independently until they were synchronized/joined, `CUDA` followed the "single-instruction multiple-data" (SIMD) model, in which each thread executes the same instruction on different pieces of data. My unfamiliarity with this model of parallelism, as well as it using different hardware (GPU) than the other APIs (CPU), lead to me not being able to get good performance with it. In addition to indicating that I should learn more about GPU programming, this serves as evidence that it is probably easier to translate programs between using combinations of `MPI`/`pthreads`/`OpenMP` than it is to translate to and from `CUDA` code.

Finally, I used the `MPI` API. This was the only API we learned that could utilize distributed memory devices, such as a cluster of nodes. It is obviously increasingly important in modern high performance computing, since core counts in clusters are growing *much* faster than in a single shared-memory architecture. However, while `MPI` allows for potentially sharing work between much higher amounts of cores, it is more costly in two ways. First, instead of creating (relatively light-weight) threads, it creates full processes. Second, it is more costly to communicate between processes, since you can't just read/write to variables in memory to communicate and passing messages requires going through the network layer. This means that while you don't have to worry as much about race conditions, you do have to worry more about minimizing the amount of communication between processes.

Project Usage

Building

To build the `lab5_mpi` executable, navigate to the top level of the submitted directory and build as follows:

```
# Ensure that you have mpicc and icc compilers
$ make
$ ls
... lab5_mpi ...
```

Running

The syntax to run the program using the `mpirun` command on the OSC clusters is:

```
$ export MV2_ENABLE_AFFINITY=0
$ mpirun -genv OMP_NUM_THREADS <num-threads> -genv KMP_AFFINITY scatter \
  -ppn 1 -n <num-nodes> ./lab5_mpi [affect-rate] [epsilon] [test-file]
```

where `num-threads` is the number of threads for `OpenMP` to spawn on each node and `num-nodes` is the number of nodes to use. Note that, unlike previous labs, `test-file` is passed an argument, not simply re-directed to `stdin`. Note that several environment variables have to be set to ensure that the `MPI` processes and `OpenMP` threads distribute appropriately amoung the available nodes/cores.