Lab 06: Approximating Pi with OpenMP

Assigned: 2019-02-26 10:00:00

Due: 2019-02-28 23:59:00

Instructions:

- Written portions of this assignment are submitted via Canvas. Unless specified otherwise, the written portion of the assignment is to be completed using LaTeX. All derivations, images, graphs, and tables are to be included in this document. Handwritten solutions will receive zero credit.
- Code portions of this assignment are submitted via `code.vt.edu`. Source code must be in the private repository, to which the CMDA 3634 instructors must have access.

Deliverables: For this assignment, you are to submit the following:

1. (Canvas) `<pid>_Lab_06.pdf`: A PDF file, renderd by `pdflatex` (the file generated by Overleaf is sufficient) containing the answers to the questions requiring written answers. Use the template provided in the project repository.
2. (`code.vt.edu`) The source files required to compile and run your solutions to the lab and the tex and image files for your report, in the appropriate directories.

Collaboration: This assignment is to be completed by yourself, however, you make seek assistance from your classmates. In your submission you must indicate from whom you received assistance.

Honor Code: By submitting this assignment, you acknowledge that you have adhered to the Virginia Tech Honor Code and attest to the following:

> I have neither given nor received unauthorized assistance on this assignment. The work I am presenting is ultimately my own.

Resources

- OpenMP:
  - General tutorial `https://computing.llnl.gov/tutorials/openMP/`
  - Timing `https://gcc.gnu.org/onlinedocs/gcc-4.5.0/libgomp/omp_005fget_005fwtime.html`
- Monte Carlo Pi: `https://academo.org/demos/estimating-pi-monte-carlo/`

Task

In this exercise you will first implement serial code for approximately computing pi and then add OpenMP compiler directives so that your program will parallelize the calculation and use an arbitrary number of threads to compute pi. Once the code runs on your virtualbox you will transfer it to Cascades via git and test it there using the SLURM batch job queueing system.

Monte Carlo algorithm for computing π: randomly choose points in a box with unit area, with bottom-left corner at the origin and check each random point to see if it is in the unit circle centered at the origin. The percentage of points inside this box converges to a number proportional to π.

Serial implementation: The serial implementation is similar to the serial implementation presented in lecture.

Parallel implementation: For the parallel implementation, the `srand48()`/`drand48` seed and pseudo-random number generating functions are not thread safe because they maintain a single internal state. You will need to replace them with the reentrant versions `srand48_r` and `drand48_r`. We recommend you use an omp parallel region, and inside that region each thread initializes the random seed based on its thread number. In the parallel region, perform a regular for loop (i.e not an `omp parallel for` loop) and use a reduction clause to sum up all the thread's hit counts.

Warning: I have indicated where you should be running commands in a terminal with the > character. This character is **not** part of the command!

1. **Setup** your coding environment.
   (a) Pull the lab materials from the upstream repository.
2. **Implement** the following requirements in C. Be sure to use git to commit your code regularly. Push early, push often!
   (a) First look at the `readme` to understand what files and scripts are available.
   (b) Now fill in the missing code in `pi.c`. Recall that we are throwing random points on the unit square resting on the first quadrant of the graph with its lower left vertext at the origin and seeing if they are within a unit circle of radius 1 centered at the origin.
   (c) Run `pi.c` to make sure it's prediction seems reasonable.
   (d) Copy your code from part (a) into the respective parts of `pi_omp.c`.
   (e) Determine which statement to use before the `for` loop in `pi_omp.c`. (Hint: we need to keep `n` and `randBuffer` private and form a reduction on `Ninside`.)
   (f) Now run this code with at least two cores. Confirm that you are still approximating pi and that you see a reasonable speedup.
   (g) Now run `scale.sh` on your laptop. Use `plotScale.py` or the plotting tool of your choice to plot the scaling.
   (h) Push your work to `code.vt.edu`. `ssh` into Cascades and clone your code. Submit the bash script as a job using `sbatch`. Then use `scp` to copy the results back to your laptop and plot them.
3. **Answer** the questions listed below. You may use Overleaf, but your tex source must be commited to the `reports/` directory.
4. **Submit** your results.

(a) After you have completed this lab (which we'll continue in class on Thursday), upload a PDF of your report to Canvas.

(b) Push your source code and latex files to `code.vt.edu`.

(c) Examine your assignment repository on `code.vt.edu` to be sure that all of your materials have been correctly submitted.

Questions

Answer the following questions. No template is provided (you may copy one from a previous week if you like). Place a copy of your report in the `reports/` directory.

1. Include a screenshot of the welcome screen that you see when you log into Cascades.
2. Explain in your own words why this algorithm works. How many samples are necessary to approximate π to 3 digits? 4 digits? 5 digits? Create a plot which shows the quality of approximation as a function of the number of samples.
3. Put your scaling plots for the laptop and Cascades in your pdf.
4. Observe that we plateau on the laptop (assuming you have less that 16 cores). Why is this?
5. The scaling for Cascades is approximately linear. Why is this? Can we expect this same scaling for other problems?
6. Why is `#pragma omp parallel for` insufficient for completing `pi_omp.c`?