CPSC 424/524 Spring 2020
Assignment 2

Due Date: Sunday, 2/23/2020 (11:59 p.m.)

Area of the Mandelbrot Set

The Problem: The Mandelbrot Set is the set of complex numbers c for which the iteration $z \leftarrow z^2 + c$ does not diverge, starting from the initial condition $z = c$. To determine (approximately) whether a point c lies in the set, a finite number of iterations are performed, and if the threshold condition $|z| > 2$ is satisfied at any iteration, then the point is considered to be outside the Mandelbrot Set. The problem in this assignment is to estimate the area of the Mandelbrot Set. There is no known theoretical value for this, but many estimates have been based on a procedure similar to the one described here.

The method used in this assignment is rather simple:

(a) Generate a grid of equal-size square cells covering a rectangular region **R** in the complex plane that contains the upper half of the Mandelbrot Set. (The Set is symmetric with respect to the real axis, so it is only necessary to work with half of it.) For this assignment, we will use the region **R** with lower-left corner –2.0+0.0i and upper-right corner 0.5+1.25i. Each cell will be a square with side length 0.001.

(b) Carry out the iteration above using as c, in turn, a randomly selected point in each cell. For this assignment, compute up to 20,000 iterations for each c. If the threshold condition $|z| > 2$ is satisfied for any iterate, then terminate the iteration and mark the cell as outside the Set. Otherwise, mark the cell as inside the Set. Let N_I and N_O denote, respectively, the total numbers of cells determined to be inside and outside of the Mandelbrot Set.

(c) To estimate the area of the Mandelbrot Set, use

$$A = 2 \times \frac{N_I}{(N_I + N_O)} \times \text{Area of } \mathbf{R}$$

Task 1: Serial Program (20 points)

Create a serial program that estimates the area of the Mandelbrot set using the method described above. I have provided a Makefile (**`/home/cpsc424_ahs3/assignments/assignment2/Makefile`**) as a template for use in building this serial program and the OpenMP versions described below. Feel free to change the program names if you wish. Use a batch job to build and run your code, requesting a single task and 1 core. **IMPORTANT: Directory structures on Grace have changed!! This impacts the locations of timing routines and other files. Be sure to check any hard-coded file paths.**

For this assignment, please use a simple linear congruential pseudorandom number generator that I have provided, unless instructed otherwise. (Note: I do not recommend this choice for "real" computational work, since there are many better random number generators available in libraries. However, it is pedagogically useful for this assignment.) I have provided the random number generator in the file **`drand.c`** in the directory **`/home/cpsc424_ahs3/assignments/assignment2`**. To initialize the random number generator, call **`dsrand(12345)`** to set an initial seed (which will actually be **`12344`**).

Once you have created your serial program, run it several times to verify that it consistently produces the same answer and to measure its performance. (As in Assignment 1, you may use my timing routines, now in **`/home/cpsc424_ahs3/utils/timing`**.) When I ran my serial program, I found the answer was around 1.506678, and the elapsed time was around 75.45 seconds. Your results could vary somewhat depending on the order in which you process the grid cells. (Your report should explain why that might be true.)

Task 2: OpenMP Program (Loop Directives) (35 points)

In this task, you will use loop directives (pragmas) to create parallel, multithreaded versions of your program.

1. Modify your program to create parallel threads using the "`omp for`" pragma without using either a **`collapse`** clause or a **`schedule`** clause. Run your program several times using 2 threads. (To control the number of OpenMP threads, set the environment variable **`OMP_NUM_THREADS`**, as in **`export OMP_NUM_THREADS=2`**.) Do your answers all agree with each other? If not, one possible cause is that the random number generator is not thread safe. Fix this problem and make several runs to demonstrate that you can reliably produce correct answers. In your report, explain how you fixed the problem and why your fix worked. Note: Depending on how you wrote your code, you may still see some slight differences as you vary the number of threads, or if you don't use a static assignment of iterations to threads. **After this step of Task 2, please use your thread-safe version of the random number generator for the rest of the assignment.**

   Run your corrected code for 1, 2, 4, 10, and 20 threads (a few times for each case). In your report, create a table containing average times and areas for these cases and also for the serial version from Task 1. Note: To reduce the tedium of making lots of runs, I have provided a sample Slurm batch script that you can modify to make the runs you need. It uses **`OMP_NUM_THREADS`** and **`OMP_SCHEDULE`** to make it simple to change the number of threads or the loop **`schedule`** clause. You can find it here: **`/home/cpsc424_ahs3/assignments/assignment2/slurmrun.sh`**

2. By default, for parallel loops, OpenMP uses static scheduling in which the total number of iterations is divided into **`OMP_NUM_THREADS`** contiguous blocks (sets of iterations), and each block is assigned to a single thread. Modify your code to try alternative schedule options and report average timings for each case you try using 2, 4, 10, and 20 threads. At the least, try:

   a. **`schedule(static,1)`**

   b. **`schedule(static,10)`**

   c. **`schedule(dynamic)`**

   d. **`schedule(dynamic,250)`**

   e. **`schedule(guided)`**

3. Experiment with the use of the **`collapse`** clause and report on a ***few*** experiments including at least one using the **`guided`** option. Should/does the **`collapse`** clause make much of a performance difference in this case? Explain your answer in the context of your particular source code, since the answer may vary depending on how you designed the code.

Task 3: OpenMP Program (Tasks) (35 points)

Modify your Task 2 program to use OpenMP **`tasks`**.

1. To begin with, create a code in which the processing of each cell constitutes a task, and one thread is dedicated to creating all the tasks. Run your code with 1, 2, 4, 10, and 20 threads and report the average area and average time for each case.

2. Now modify your program so that it treats each row of cells as a task. Again, run your code with 1, 2, 4, 10, and 20 threads and report the average area and average time for each case.

3. Finally, modify your program so that task creation is shared by all the threads. Again, run your code with 1, 2, 4, 10, and 20 threads and report the average area and average time for each case.

4. In your report, briefly discuss the observed performance for the various task-based implementations, and compare those versions with the versions from Task 2 using loop directives. (Be concise; just highlight the most important observations.)

Task 4: Parallel Random Number Generation (10 points)

A shortcoming of the random number generator I provided is that all the threads use the same sequence of random numbers. Search on line for *simple* approaches to cure this particular problem and modify **`drand.c`** to implement one of them in your best-performing program from Task 2. Run your modified code several times using 20 threads and compare the results (average area and time) to the unmodified version from Task 2. Does it make a significant difference? (Note: I'm not asking you to create a better sequence of pseudo-random numbers; all that's required is to ensure that each thread has a distinct sequence of numbers with essentially the same statistics as the original sequence.)

Procedures for Programming Assignments

For this class, we will use the Canvas website to submit solutions to programming assignments.

Remember: While you may discuss the assignment with me, a ULA, or your classmates, the source code you turn in must be yours alone and should not represent collaborations or the ideas of others!

What should you include in your solution package?

1. **All source code files, one (1) Makefile, and all scripts that you developed, used, or modified.** All source code files should contain proper attributions and suitable comments to explain your code. [NOTE: More details on this later.]
2. For this assignment, your package should include:
   a. A single Makefile that can build all of the programs you use in the assignment, using a different target for each program.
   b. One or more Slurm scripts, that build programs and run them using environment variables (such as OMP_NUM_THREADS or OMP_SCHEDULE) to set OpenMP options. You can probably build one really long script that allocates a full node (20 cores) and then uses environment variables to control the option for each run. Alternatively, you may find it easier to create a number of separate scripts for different cases. That's up to you; either way is fine. I have provided a sample script in **`/home/cpsc424_ahs3/assignments/assignment2`**.
   c. A single shell script to submit all of the execution Slurm scripts.
3. **A report in PDF format** containing:
   a. Your name, the assignment number, and course name/number.
   b. Information on building and running the code:
      i. A brief description of the software/development environment used. For example, you should list the module files you've loaded.
      ii. Steps/commands used to compile, link, and run the submitted code. Best is to use a Makefile for building the code and to submit an **sbatch** script for executing it. (If you ran your code interactively, then you'll need to list the commands required to run it.)
      iii. Outputs from executing your program.
   c. Any other information required for the assignment, including any questions you were asked to answer.

How should you submit your solution?

1. On the cluster, create a directory named "**NetID_ps2_cpsc424**". (For me, that would be "**ahs3_ps2_cpsc424**". Put into it all the files you need to submit.
2. Create a compressed tar file of your directory by running the following in its parent directory:

   tar -cvzf NetID_ps2_cpsc424.tar.gz NetID_ps2_cpsc424

3. To submit your solution, click on the "Assignments" button on the Canvas website and select this assignment from the list. Then click on the "Submit Assignment" button and upload your solution file **NetID_ps2_cpsc424.tar.gz**. You may add additional comments to your submission, but your report should be included in the attachment. You can use scp or rsync or various GUI tools (e.g., CyberDuck) to move files back and forth to Grace.

Due Date and Late Policy

Due Date: **Sunday, February 23, 2020 by 11:59 p.m.**

Late Policy: On time submission: Full credit

Up to 24 hours late: 90% credit

Up to 72 hours late: 75% credit

Up to 1 week late: 50% credit

More than 1 week late: 35% credit

General Statement on Collaboration

Unless instructed otherwise, all submitted assignments must be your own individual work. Neither copied work nor work done in groups will be permitted or accepted without prior permission.

However….

You may discuss questions about assignments and course material with anyone. You may always consult with the instructor or ULAs on any course-related matter. You may seek general advice and debugging assistance from fellow students, the instructor, ULAs, Piazza conversations, and Internet sites.

However, except when instructed otherwise, the work you submit (e.g., source code, reports, etc.) must be entirely your own. If you do benefit substantially from outside assistance, then you must acknowledge the source and nature of the assistance. (In rare instances, your grade for the work may be adjusted appropriately.) It is acceptable and encouraged to use outside resources to *inform* your approach to a problem, but *plagiarism is unacceptable* in any form and under any circumstances. If discovered, plagiarism will lead to adverse consequences for all involved.

DO NOT UNDER ANY CIRCUMSTANCES COPY ANOTHER PERSON'S CODE—to do so is a clear violation of ethical/academic standards that, when discovered, will be referred to the appropriate university authorities for disciplinary action. Modifying code to conceal copying only compounds the offense.