CPSC 424/524 Spring 2023
Assignment 2: Area of the Mandelbrot Set
Due Date: Thursday, October 12, 2023 (11:59 p.m.)

The Problem: The Mandelbrot Set is the set of complex numbers c for which the values z computed by the infinite iteration $z \leftarrow z^2 + c$ never exceed 2.0, starting from the initial condition $z = 0$. To determine (approximately) whether a point c lies in the Set, a finite number of iterations is performed. If the threshold condition $|z| > 2$ is satisfied after any iteration, the point is deemed to be outside the Mandelbrot Set. Points that complete all the iterations without satisfying the threshold condition are deemed to be inside the Set. Your task 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 below.

The method used in this assignment is rather simple:

(a) Generate a "virtual" grid of equal-size square cells covering a rectangular region **R** in the complex plane that contains 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, you 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. The grid is "virtual" because you should not create large coordinate arrays. Instead, you can easily (and cheaply) compute the cells' corner coordinates, as needed.

(b) Carry out the iteration above using as c, in turn, a randomly selected point in each cell. For this assignment, compute up to 25,000 iterations for each c. If the threshold condition $|z| > 2$ is satisfied at the end of any iteration, then immediately terminate the computation for c and mark its cell as outside the Set. Otherwise, mark its 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 Set.

(c) Finally, to estimate the area of the Mandelbrot Set, use

$$A = 2.0 \times \frac{N_I}{(N_I + N_O)} \times \text{Area of } \mathbf{R}$$

Task 1: Serial Program (20 points)

Create a serial program that estimates the area of the Mandelbrot Set using the method described above. Use the file **`/gpfs/gibbs/project/cpsc424/shared/assignments/assignment2/Makefile`** as a template for creating your own Makefile that can be used to build this serial program and the parallel OpenMP programs described below. (You may change the program names if you wish.) Use a batch job to build and run your serial program, using a single Slurm task and 1 core. As in Assignment 1, you may use the routines in **`/gpfs/gibbs/project/cpsc424/shared/utils/timing`** to time your code.

For this assignment, please use the 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 **`/gpfs/gibbs/project/cpsc424/shared/assignments/assignment2`**. To initialize the random number generator, call **`dsrand(12345)`** to set an initial seed (which will actually be **`12344`**).

Run your serial program several times to verify that it consistently produces the same answer and to measure its performance. When I ran my serial program, I found the computed area was around 1.506632, and the elapsed time was around 50.78 seconds. Your results may vary from mine depending on the order in which you process the grid cells. (Your report should explain why that might happen.)

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? If not, one possible cause is that the random number generator is not thread safe. Fix this problem with a simple code change and make several runs to demonstrate that you can reliably produce correct answers. In your report, describe and explain what the problem was, how you fixed it, 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. (In your report, you should explain why there might be variations.) **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, 12, and 24 cores with one thread per core (a few times for each case). In your report, include a table showing average times and areas for each case and also for the serial version from Task 1. Note: To reduce the tedium of making many runs, you may use/modify a sample Slurm batch script to make the runs you need. It uses the environment variables **`OMP_NUM_THREADS`** and **`OMP_SCHEDULE`** to make it simple to change the number of threads or the loop **`schedule`** clause. The script covers building and running most of the program variants that you need to create and run for this assignment. You can find the script here:

   `/gpfs/gibbs/project/cpsc424/shared/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, 12, and 24 cores with one thread per core. At the least, try:

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

   b. **`schedule(static,100)`**

   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, 12, and 24 cores with one thread per core and report the average area and average time for each case.
2. Now modify your program so that it treats each column of cells as a task. Again, run your code with 1, 2, 4, 12, and 24 cores with one thread per core 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, 12, and 24 cores with one thread per core 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)

One (of many) possible shortcomings of the random number generator I provided is that multiple threads may wind up using the same sequence of random numbers. Search on line for a *simple* approach to cure this particular problem and modify **`drand.c`** to implement it in your best-performing program from Task 2. Run your modified code several times using 24 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 "statistically better" sequence of pseudo-random numbers. The aim is simply to try to ensure that each thread uses a distinct sequence of numbers with statistics that are essentially the same as the corresponding serial sequence. One useful reference/source for Task 4 may be the material by Ian Foster available at https://www.mcs.anl.gov/~itf/dbpp/text/node116.html. (If you choose to use a version of Foster's approach, be sure to properly acknowledge it in your report.)

Additional Notes: In addition to Chapters 4 and 5 of the Pacheco and Malensek text, You might be interested in the following:

1. For background on parallelization, you may wish to read Chapter 5 of the Hager text.
2. For information about multithreading and OpenMP, you may wish to read Chapter 6 in the Hager text and/or Chapter 7 of the Robey & Zamora e-book (if you have access to it).

Procedures for Programming Assignments

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

Remember: While you may discuss the assignment with the instructor, a ULA or your classmates, the work you turn in must be yours alone!

What should you include in your solution package?

1. **All source code files, Makefiles, and scripts that you developed or modified.** All source code files should contain proper attributions, as needed. In addition, make sure that your source code contains suitable documentation (e.g., comments) so that someone looking at your code can understand what it's trying to do. In part, your grade will depend on the documentation, code organization, and comments for your code.
2. **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 Slurm script that you submit using `sbatch`. The script could use a Makefile to build the code and would then run it multiple times so that you can report average timings. (If you ran your code interactively, then your report will need to list the commands required to run it.)
      iii. Outputs from executing your program(s).
   c. Any other information required for the assignment (e.g., answers to questions that may be asked in the assignment).

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, including your report.
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`. (We will set Canvas to accept only files with a "`gz`" or "`tgz`" extension.) You may add additional comments to your submission, but your report, including the plot, should be included in the attachment. You can use OOD, tools like 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: **Thursday, October 12, 2023 by 11:59 p.m.**

Late Policy: On time submission: Full credit

Up to 24 hours late: 85% credit

Up to 72 hours late: 50% credit

Up to 1 week late: 25% credit

More than 1 week late: No 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, Canvas ED discussions, and Internet sites, including https://ask.cyberinfrastructure.org/c/yale/40 or similar 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.