Assignment 2 Report

Jui-yang Cheng

CPSC 424

Development Envirnoment

I used an IDE called *CodeRunner* to implement, test and debug the code on my local computer. The only module I loaded on *Grace* is the *intel* module. I loaded it with command `module load intel`.

Submit All Execution Slurm Script

Run `./submit.sh` to submit all the execution slurm script.
The output will be in `summary-X.out` where X should be replaced by the corresponding job number.

Task 1

Terminal Commands

- Compile and link the code: `make mandseq`
- Build and execute the code: `sbatch ./mandseq.sh`
- View and collect data: `vi mandseq-X.out` where X should be replaced by the corresponding job number.

Summary

The result I get from my serial program is consistently 1.521104 and the average wall clock runtime is 87.840906 seconds.

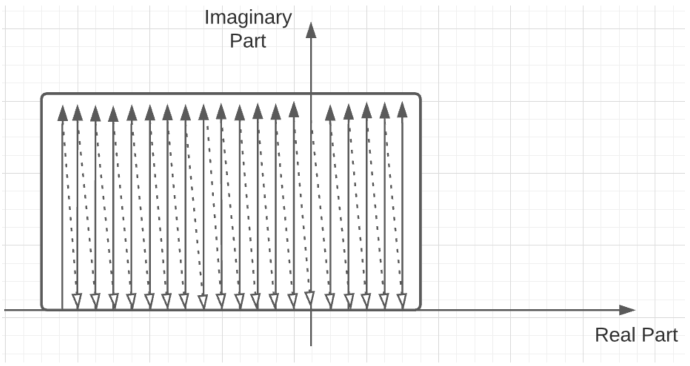
Sample Output

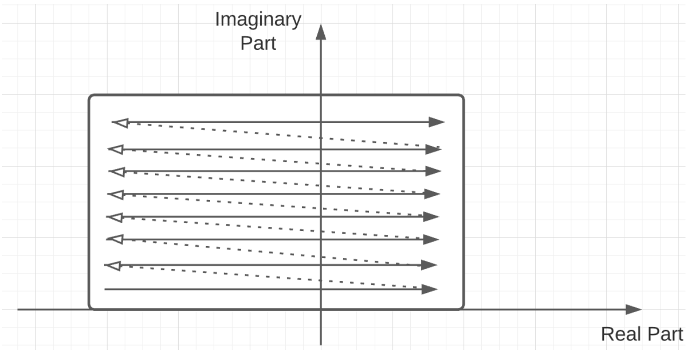
```
Seed = 12344. RAND_MAX = 2147483647.
Estimation of the area of the Mandelbrot set: 1.521104
Wall clock time used: 87.797007
CPU time used: 87.801289
```

Experiment

Originally, I programmed it to visit the cells column by column from left to right (shown in the first graph below and get the results in the *summary* section, which gives a consistent answer of 1.521104.

However, I noticed that the serial program gives a different consistent result of 1.521420 if I program it to visit the cells row by row from left to right.





Explanation

The reason of the program giving different results is the nature of the `drand()` function provided. Since we always start with seed `12345`, we will always have the same sequence of "random" number according to the algorithm. For example, the first two "random" number generated is always added to the base coordinates (e.g. the base coordinate for real part is -2 and imaginary part is 0 if we visit the cells row by row and from left to right)of the celling being visited (assuming one "random" number is added to the real part and the other is added to the imaginary part). Therefore, depending on the visiting order, the base coordinates of the first cells visited are different. Adding the same "random" numbers to base coordinates of different cells is going to generate different results. Some of them may go over the threshold and some of them may not, causing the numbers of cells that are determined to be inside or outside of the Mandelbrot Set to be different, which will eventually lead to different estimation of Mandelbrol Set.

Task 2.1

Terminal Commands

- Compile and link to the code: `make mandomp`
- Build and execute the code: `sbatch ./mandomp.sh`
- View and collect the data: `vi mandomp-X.out` where X should be replaced by the corresponding job number.

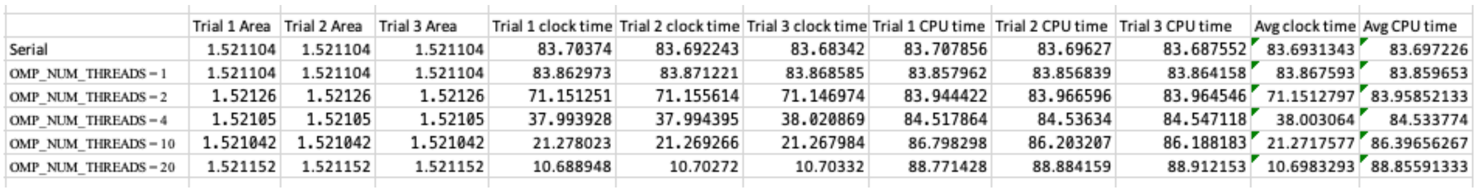
Problem

The reason for my answers not agree with each other is that the random number generator is not thread safe. When the program is executed serially, the "random" number generated is always the same for the same iteration due to the nature of provide `drand()`. However, when the program runs in parellel, the sequence of threads calling `drand()` become unpredictable and, since the value of `seed` changes every time a thread calls `drand()`, the value of global variable `seed` also become unpredictable.

Fix

To fix this problem, I make the value of `seed` private to each thread calling it. In this way, the value of `seed` won't be global and each thread can modify only the value of `seed` that is private to it. Avoiding the problem of unpredictable value change in `seed`.

Table



	Trial 1 Area	Trial 2 Area	Trial 3 Area	Trial 1 clock time	Trial 2 clock time	Trial 3 clock time	Trial 1 CPU time	Trial 2 CPU time	Trial 3 CPU time	Avg clock time	Avg CPU time
Serial	1.521104	1.521104	1.521104	83.70374	83.692243	83.68342	83.707856	83.69627	83.687552	83.6931343	83.697226
OMP_NUM_THREADS = 1	1.521104	1.521104	1.521104	83.862973	83.871221	83.868585	83.857962	83.856839	83.864158	83.867593	83.859653
OMP_NUM_THREADS = 2	1.52126	1.52126	1.52126	71.151251	71.155614	71.146974	83.944422	83.966596	83.964546	71.1512797	83.95852133
OMP_NUM_THREADS = 4	1.52105	1.52105	1.52105	37.993928	37.994395	38.020869	84.517864	84.53634	84.547118	38.003064	84.533774
OMP_NUM_THREADS = 10	1.521042	1.521042	1.521042	21.278023	21.269266	21.267984	86.798298	86.203207	86.188183	21.2717577	86.39656267
OMP_NUM_THREADS = 20	1.521152	1.521152	1.521152	10.688948	10.70272	10.70332	88.771428	88.884159	88.912153	10.6983293	88.85591333

Task 2.2

Terminal Commands

- Compile and link to the code: `make mandompts`
- Build and execute the code: `sbatch ./mandompts.sh`
- View and collect data: `vi mandompts-X.out` where X should be replaced by the corresponding job number.

Average Timing

- `OMP_NUM_THREADS = 2`
    - `OMP_SCHEDULE = static,1`
        - CPU time: 84.158017 seconds
        - Wall Clock time: 42.083378 seconds
    - `OMP_SCHEDULE = static,10`
        - CPU time: 84.140653 seconds
        - Wall Clock time: 42.075526 seconds

    - `OMP_SCHEDULE = dynamic`
        - CPU time: 84.093521 seconds
        - Wall Clock time: 42.050017 seconds
    - `OMP_SCHEDULE = dynamic,250`
        - CPU time: 84.231754 seconds
        - Wall Clock time: 44.302195 seconds
    - `OMP_SCHEDULE = guided`
        - CPU time: 84.156056 seconds
        - Wall Clock time: 42.200766 seconds
- `OMP_NUM_THREADS = 4`
    - `OMP_SCHEDULE = static,1`
        - CPU time: 84.876828 seconds
        - Wall Clock time: 21.222197 seconds
    - `OMP_SCHEDULE = static,10`
        - CPU time: 85.064917 seconds
        - Wall Clock time: 21.273288 seconds
    - `OMP_SCHEDULE = dynamic`
        - CPU time: 84.958424 seconds
        - Wall Clock time: 21.241671 seconds
    - `OMP_SCHEDULE = dynamic,250`
        - CPU time: 84.883263 seconds
        - Wall Clock time: 27.174229 seconds
    - `OMP_SCHEDULE = guided`
        - CPU time: 84.864473 seconds
        - Wall Clock time: 21.376983 seconds
- `OMP_NUM_THREADS = 10`
    - `OMP_SCHEDULE = static,1`

        - CPU time: 87.251738 seconds
        - Wall Clock time: 8.726911 seconds

    - `OMP_SCHEDULE = static,10`

        - CPU time: 87.769421 seconds
        - Wall Clock time: 8.789285 seconds

    - `OMP_SCHEDULE = dynamic`

        - CPU time: 87.372859 seconds
        - Wall Clock time: 8.739064 seconds

    - `OMP_SCHEDULE = dynamic,250`

        - CPU time: 86.492194 seconds
        - Wall Clock time: 21.285678 seconds

    - `OMP_SCHEDULE = guided`

        - CPU time: 86.525992 seconds
        - Wall Clock time: 8.657152 seconds

- `OMP_NUM_THREADS = 20`

    - `OMP_SCHEDULE = static,1`

        - CPU time: 93.192053 seconds
        - Wall Clock time: 4.683808 seconds

    - `OMP_SCHEDULE = static,10`

        - CPU time: 93.586879 seconds
        - Wall Clock time: 4.835152 seconds

    - `OMP_SCHEDULE = dynamic`

        - CPU time: 91.762464 seconds
        - Wall Clock time: 4.592875 seconds

    - `OMP_SCHEDULE = dynamic,250`

        - CPU time: 88.575462 seconds
        - Wall Clock time: 21.278675 seconds

- `OMP_SCHEDULE = guided`
    - CPU time: 90.311622 seconds
    - Wall Clock time: 4.523301 seconds

Task 2.3

Terminal Commands

- Compile and link the code: `make mand_collapse`
- Build and execute the code: `sbatch ./collapse.sh`
- View and collect data: `vi collapse-X.out` where X should be replaced by the corresponding job number.

Average Timing with `collapse(2)`

- `OMP_NUM_THREADS = 4`
  - `OMP_SCHEDULE = static,10`
    - CPU time: 84.982017 seconds
    - Wall Clock time: 21.249456 seconds
  - `OMP_SCHEDULE = dynamic`
    - CPU time: 86.681798 seconds
    - Wall Clock time: 21.673342 seconds
  - `OMP_SCHEDULE = guided`
    - CPU time: 84.920952 seconds
    - Wall Clock time: 21.483426 seconds
- `OMP_NUM_THREADS = 10`
  - `OMP_SCHEDULE = static,10`
    - CPU time: 88.309119 seconds
    - Wall Clock time: 9.016680 seconds
  - `OMP_SCHEDULE = dynamic`
    - CPU time: 92.133944 seconds

        - Wall Clock time: 9.216047 seconds
    - `OMP_SCHEDULE = guided`
        - CPU time: 86.463005 seconds
        - Wall Clock time: 8.649400 seconds
- `OMP_NUM_THREADS = 20`
    - `OMP_SCHEDULE = static,10`
        - CPU time: 95.476628 seconds
        - Wall Clock time: 4.861534 seconds
    - `OMP_SCHEDULE = dynamic`
        - CPU time: 101.458304 seconds
        - Wall Clock time: 5.084708 seconds
    - `OMP_SCHEDULE = guided`
        - CPU time: 90.039862 seconds
        - Wall Clock time: 4.507422 seconds

Performance Analysis

The clause `collapse()` can greatly improves the performance when the `for` loop followes right after it has fewer number of iterations to run than the number of thread created.
Therefore, the clause `collapse()` does not and should not make much of a performance difference in this case because the `for` loop follows right after the `#pragma` clauses has 2500 iterations need to perform. Since there is a maximum of 20 threads in this case, each threads has an average of 125 iteration from the outer-most loop to run, which is sufficient to keep them busy. It is very unlikely for any of the threads to be idle. Therefore, collasing the nesting loops won't make much difference in terms of performance.

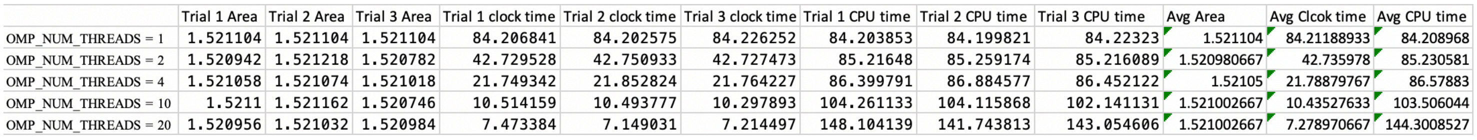
Task 3.1

Terminal Command

- Compile and link to the code: `make mandomptasks_part_one`
- Build and execute the code: `sbatch ./task_1.sh`
- View and collect data: `vi task_part_one-X` where X should be replaced by the corresponding job number.

Data Table

The following table contains the trial data, the average area, the average clock time and the average CPU time.



	Trial 1 Area	Trial 2 Area	Trial 3 Area	Trial 1 clock time	Trial 2 clock time	Trial 3 clock time	Trial 1 CPU time	Trial 2 CPU time	Trial 3 CPU time	Avg Area	Avg Clcok time	Avg CPU time
OMP_NUM_THREADS = 1	1.521104	1.521104	1.521104	84.206841	84.202575	84.226252	84.203853	84.199821	84.22323	1.521104	84.21188933	84.208968
OMP_NUM_THREADS = 2	1.520942	1.521218	1.520782	42.729528	42.750933	42.727473	85.21648	85.259174	85.216089	1.520980667	42.735978	85.230581
OMP_NUM_THREADS = 4	1.521058	1.521074	1.521018	21.749342	21.852824	21.764227	86.399791	86.884577	86.452122	1.52105	21.78879767	86.57883
OMP_NUM_THREADS = 10	1.5211	1.521162	1.520746	10.514159	10.493777	10.297893	104.261133	104.115868	102.141131	1.521002667	10.43527633	103.506044
OMP_NUM_THREADS = 20	1.520956	1.521032	1.520984	7.473384	7.149031	7.214497	148.104139	141.743813	143.054606	1.521002667	7.278970667	144.3008527

Task 3.2

Code Change

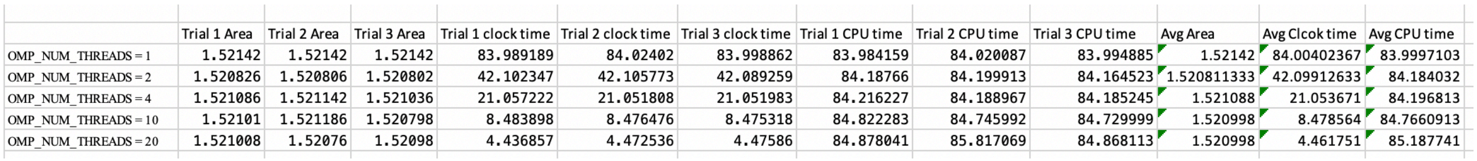
I moved the location of `#pragma omp task` from within the second `for` loop to the first `for` loop so that it treats each row of cells as a task.

Terminal Command

- Compile and link to the code: `make mandomptasks_part_two`
- Build and execute the code: `sbatch ./task_2.sh`
- View and collect data: `vi task_part_two-X` where X should be replaced by the corresponding job number.

Data Table

The following table contains the trial data, the average area, the average clock time and the average CPU time.



	Trial 1 Area	Trial 2 Area	Trial 3 Area	Trial 1 clock time	Trial 2 clock time	Trial 3 clock time	Trial 1 CPU time	Trial 2 CPU time	Trial 3 CPU time	Avg Area	Avg Clcok time	Avg CPU time
OMP_NUM_THREADS = 1	1.52142	1.52142	1.52142	83.989189	84.02402	83.998862	83.984159	84.020087	83.994885	1.52142	84.00402367	83.9997103
OMP_NUM_THREADS = 2	1.520826	1.520806	1.520802	42.102347	42.105773	42.089259	84.18766	84.199913	84.164523	1.520811333	42.09912633	84.184032
OMP_NUM_THREADS = 4	1.521086	1.521142	1.521036	21.057222	21.051808	21.051983	84.216227	84.188967	84.185245	1.521088	21.053671	84.196813
OMP_NUM_THREADS = 10	1.52101	1.521186	1.520798	8.483898	8.476476	8.475318	84.822283	84.745992	84.729999	1.520998	8.478564	84.7660913
OMP_NUM_THREADS = 20	1.521008	1.52076	1.52098	4.436857	4.472536	4.47586	84.878041	85.817069	84.868113	1.520998	4.461751	85.187741

Task 3.3

Code Change

I removed `#pragma omp single` from the code and put `for` directive back so that task creation is shared by all the threads.

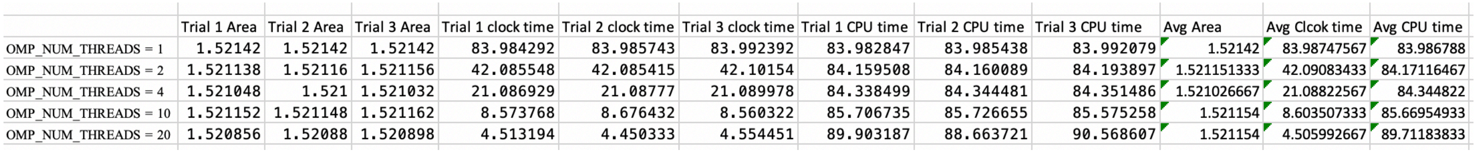
Terminal Command

- Compile and link to the code: `make mandomptasks_part_three`
- Build and execute the code: `sbatch ./task_3.sh`
- View and collect data: `vi task_part_three-X` where X should be replaced by the corresponding job

number.

Data Table

The following table contains the trial data, the average area, the average clock time and the average CPU time.



	Trial 1 Area	Trial 2 Area	Trial 3 Area	Trial 1 clock time	Trial 2 clock time	Trial 3 clock time	Trial 1 CPU time	Trial 2 CPU time	Trial 3 CPU time	Avg Area	Avg Clcok time	Avg CPU time
OMP_NUM_THREADS = 1	1.52142	1.52142	1.52142	83.984292	83.985743	83.992392	83.982847	83.985438	83.992079	1.52142	83.98747567	83.986788
OMP_NUM_THREADS = 2	1.521138	1.52116	1.521156	42.085548	42.085415	42.10154	84.159508	84.160089	84.193897	1.521151333	42.09083433	84.17116467
OMP_NUM_THREADS = 4	1.521048	1.521	1.521032	21.086929	21.08777	21.089978	84.338499	84.344481	84.351486	1.521026667	21.08822567	84.344822
OMP_NUM_THREADS = 10	1.521152	1.521148	1.521162	8.573768	8.676432	8.560322	85.706735	85.726655	85.575258	1.521154	8.603507333	85.66954933
OMP_NUM_THREADS = 20	1.520856	1.52088	1.520898	4.513194	4.450333	4.554451	89.903187	88.663721	90.568607	1.521154	4.505992667	89.71183833

Task 3.4

Observation

By comparing the results from Task 3.1, Task 3.2 and Task 3.3, I notice that the average performance is worst when treating each cell as a task and use only one thread to create task (Task 3.1), especially when there are multiple threads available, and the performances of Task 3.2 and Task 3.3 do not have much difference. All of them have a higher performance than the version from Task 2 using loop directive and default scheduling.

Task 4

Terminal Command

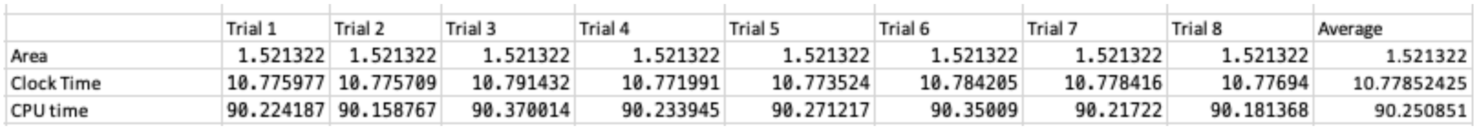
- Compile and link the code: `make mandompts_parallel`
- Build and execute the code: `sbatch ./parallel.sh`
- View and collect data: `vi parallel-X.out` where X should be replaced by the corresponding job number.

Approach

I searched online and found an method called ***Leapfrog***, and use the idea of it to implement the changes.

Result

The following table contains the data of running the program using 20 threads.



	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Average
Area	1.521322	1.521322	1.521322	1.521322	1.521322	1.521322	1.521322	1.521322	1.521322
Clock Time	10.775977	10.775709	10.791432	10.771991	10.773524	10.784205	10.778416	10.77694	10.77852425
CPU time	90.224187	90.158767	90.370014	90.233945	90.271217	90.35009	90.21722	90.181368	90.250851

Observation

The values of average area, average wall clock time and average CPU time do not have a significant difference than the unmodified version from Task 2.