

# CPSC 524 Homework 2

Jonas Katona

October 15, 2023

## 1 Overall

For codes and scripts dealing with each exercise in particular, I will explain these in their respective sections below, but before doing that, I will describe my general environment/tools and also give a general overview of what each of the scripts do. Finally, I use the term “we” in this report in a manner which is inclusive for the reader, and it is *not* an indication that I have worked on this assignment with anybody. I attest that all of the words in this report are my own, and that all of the scripts provided (unless otherwise indicated, e.g., for the timing routines) were written entirely on my own.

### 1.1 Makefile

After writing the C codes for Exercise 1 (`exercise_1.c`), Exercise 2 (`exercise_2.c` and `exercise_2-*.c`), Exercise 3 (`exercise_3.c` and `exercise_3-*.c`), Exercise 4 (`exercise_4.c`) and the thread-safe random number generators (`drand-ts.c` and `drand-ts-2.c`), we can compile these all at once using the makefile entitled `Makefile`. `Makefile` was partially based on the default makefile provided in `/gpfs/gibbs/project/cpsc424/shared/assignments/assignment2` but modified by adding for loops for greater readability and a few slight formatting changes. The targets provided in `Makefile` are listed below. Targets 1-4 all use the original, non-thread-safe pseudorandom number generator already given to us in `/gpfs/gibbs/project/cpsc424/shared/assignments/assignment2`, `drand.c`, while targets 6-12 all use the first version of the thread-safe pseudorandom number generator I wrote, `drand-ts.c`. Target 13 uses the second version of the thread-safe pseudorandom number generator I wrote, `drand-ts-2.c`.

1. `mandseq`: Compiles `exercise_1.c`, which is the serial version of the Mandelbrot set area estimation code, using compiler options defined in `CFLAGS_SERIAL`, i.e., the “optimal” compiler options provided in HW 1.
2. `mandomp-critical`: Compiles `exercise_2-crit.c`, which is the first parallel version of the Mandelbrot set code using an `omp for pragma` with no `collapse` or `schedule` clauses, using compiler options defined in `CFLAGS_OMP`, i.e., the “optimal” compiler options provided in HW 1 plus the option `-qopenmp` (required for using OpenMP pragmas at compile time). Within the Mandelbrot set area estimation code, there is a section where `NI`, the counter for the number of points in the Mandelbrot set, must be preincremented; as a synchronization directive to make this operation thread-safe, `exercise_2-crit.c` puts the preincrementing within a critical region using `#pragma omp critical`.

3. **mandomp-atomic**: Same as **mandomp-critical**, except for compiling `exercise_2-atom.c`, which protects the preincrementing using `#pragma omp atomic`, since preincrementing is an atomic operation.
4. **mandomp-reduction**: Same as **mandomp-critical**, except for compiling `exercise_2-reduce.c`, which protects the preincrementing by applying a reduction at `#pragma omp for` before the for loop and adding `reduction(+:NI)`.
5. **ts-random**: Compiles `drand-ts.c`, the first version of a thread-safe version of the pseudorandom number generator that does *not* require one to change the original parallel version of the Mandelbrot set code. The way this code works will be explained later.
6. **mandomp-ts**: Compiles `exercise_2-reduce.c` in the same manner as for **mandomp-reduction**, except using the thread-safe pseudorandom number generator `drand-ts.c`.
7. **mandomp-loops**: Compiles `exercise_2-loops.c` using the thread-safe pseudorandom number generator `drand-ts.c`. `exercise_2-loops.c` is the same as `exercise_2-atom.c` (i.e., it uses `#pragma omp atomic`) except the script also adds a `schedule(runtime)` clause in `#pragma omp for` which allows one to specify different scheduling algorithms at runtime using environment variables, which we do.
8. **mandomp-collapse**: Compiles `exercise_2-collapse.c` using the thread-safe pseudorandom number generator `drand-ts.c`. `exercise_2-collapse.c` is the same as `exercise_2-loops.c` but also introduces the clause `collapse(2)` in `#pragma omp for`, which combines the two outermost nested for loops in the algorithm (those which are looping over different cells within the domain) into a single one which is then divided across different cores.
9. **mandomp-tasks**: Compiles `exercise_3.c` using the thread-safe pseudorandom number generator `drand-ts.c`. `exercise_3.c` treats every cell as a separate task via `#pragma omp task`, which is introduced at the start of the iteration for each cell; it no longer uses `#pragma omp for`, `schedule`, or `collapse`, since the outer two for loops are only needed for the master thread (which is only *one* master thread due to the use of `#pragma omp single`).
10. **mandomp-tasks-row**: Compiles `exercise_3-row.c` using the thread-safe pseudorandom number generator `drand-ts.c`. `exercise_3-row.c` treats every column (i.e., all cells corresponding to a given cell length in the reals) as a task. This is done by having the master thread called by `#pragma omp single` loop through each “real” cell length (i.e., each column), each of which is then given as a task to another thread via `#pragma omp task`. `firstprivate(i)` ensures that the column counter `i` is appropriately initialized within each task separately and privately, and even though doing so is probably unnecessary since all private variables defined before `#pragma omp task` implicitly become `firstprivate` within the task region, it does not hurt to make sure.
11. **mandomp-tasks-shared**: Compiles `exercise_3-tasks-shared.c` using the thread-safe pseudorandom number generator `drand-ts.c`. `exercise_3-tasks-shared.c` goes through the algorithm by having all threads go through the outer two for loops concurrently. For each unique pair of outer iteration counters `i` and `j`, precisely one of the threads will enter the `#pragma omp single` region while the rest continue to go through the double for loop. That thread will encounter `#pragma omp task` and assign

another thread the innermost iteration of the map  $z \leftarrow z^2 + c$ . Thus, assigning tasks is shared amongst the threads, as while one thread is assigning a task for one pair of  $i$  and  $j$ , another thread could already be doing the same for another pair of  $i$  and  $j$  as it will have encountered the next `#pragma omp single` region. Some overhead is involved as all of the threads begin by just going through the outer two for loops (sort of like busy-waiting), but eventually the process becomes quite productive as more and more threads are either working on tasks or assigning tasks. Or at least, that would be the intention.

12. `mandomp-tasks-row-shared`: Same as `mandomp-tasks-shared` except using `exercise_3-tasks-row-shared.c` in place of `exercise_3-tasks-shared.c`, which is the same as `exercise_3-tasks-shared.c` except only using the outermost for loop to share task creation (i.e., **despite the name**,<sup>1</sup> looping through the columns) and assigning each column of cells as a task.
13. `mandomp-collapse-ts`: Compiles `exercise_4.c` using the thread-safe pseudorandom number generator `drand-ts-2.c` to answer Exercise 4. I will explain what each of these scripts are when going into my response for Exercise 4, but `exercise_4.c` is based on `exercise_2-collapse.c` with some modifications necessary to use `drand-ts-2.c` in a thread-safe manner.
14. `ts-random-2`: Compiles `drand-ts-2.c`, the second version of a thread-safe pseudorandom number generator. `drand-ts-2.c` *does* require one to change the parallel version of the Mandelbrot set code but in a way which is decidedly more elegant.
15. `clean`: Erases all executables and object files which could be generated by `Makefile`.
16. `all`: Runs all targets in `Makefile`.

## 1.2 Slurm build

`slurmrun.sh` is partially based on the sample batch script with the same name provided in `/gpfs/gibbs/project/cpsc424/shared/assignments/assignment2`, but I have changed the locations of a few spaces, changed some descriptions, added a few extra makes than were originally given (e.g., scripts which test different loop synchronization directives in Exercise 2 and trying shared task creation over rows and cells in Exercise 3), and put the multiple runs over varying numbers of threads into for loops (which are nested if necessary). However, since running everything involved for this assignment in `slurmrun.sh` would take well over an hour (which proved not to be super helpful when trying to debug and rerun parts of the assignment), I have also split `slurmrun.sh` into four separate scripts, organized according to the relevant Exercise in question: `slurmrun_i.sh` for  $i=1,2,3,4$ . The results/output for running each of these scripts separately are found in `MandelbrotFinalTest-27030939.out`, `MandelbrotFinalTest-27067660.out`, `MandelbrotFinalTest-27070337.out`, and `MandelbrotFinalTest-27072158.out`, respectively. These are provided in the report, but for sake of organization, they are located in the Appendix.

`rand_test.c` is a script which I used to test out both the range and randomness of `drand-ts.c`; it takes the number of parallel threads as a user input. `run_test.sh` was used to run these tests and output the results in a text file, `rand_samples.txt`. I plotted the

---

<sup>1</sup>`exercise_3-tasks-row-shared.c` was the name of the corresponding script in the `Makefile`, so I kept the name even though it involved looping through columns, not rows.

results from `rand_samples.txt` in a histogram using an external Python script to check that they looked sufficiently close to  $U(0, 1)$ , a uniform distribution on the unit interval in  $\mathbb{R}$ .

As for the text editor I used while working in the Grace OOD virtual desktop to write all my scripts, I used Emacs, which can be accessed by running `emacs` from the command line.

### 1.3 C compiler and modules

Regardless of the compiler options, running `icc --version` (after loading the Intel compiler module using `module load intel/2022b`) gives the C compiler I used:

```
icc (ICC) 2021.7.1 20221019
Copyright (C) 1985-2022 Intel Corporation. All rights reserved.
```

and running `which icc` gives the location:

```
$DIR$/intel-compilers/2022.2.1/compiler/2022.2.1/linux/bin/intel64/icc
```

where `DIR=/vast/palmer/apps/avx2/software`. We also have the module list after running `module list`:

Currently Loaded Modules:

- 1) StdEnv (S)
- 2) GCCcore/12.2.0
- 3) zlib/1.2.12-GCCcore-12.2.0
- 4) binutils/2.39-GCCcore-12.2.0
- 5) intel-compilers/2022.2.1
- 6) numactl/2.0.16-GCCcore-12.2.0
- 7) UCX/1.13.1-GCCcore-12.2.0
- 8) impi/2021.7.1-intel-compilers-2022.2.1
- 9) imkl/2022.2.1
- 10) iimpi/2022b
- 11) imkl-FFTW/2022.2.1-iimpi-2022b
- 12) intel/2022b

Where:

S: Module is Sticky, requires `--force` to unload or purge

## 2 Exercise 1

### 2.1 Codes

By describing the serial version of our Mandelbrot set area algorithm, `exercise_1.c`, we describe virtually the same code used for the rest of the homework, as every other script merely features parallelization of this baseline script. The script starts by importing the following header files: `stdlib.h` (for reading and clearing inputs from the command line), `math.h` (probably unnecessary, but imported just in case), `timing.h` (the timing routine provided from the first homework), and `drand.h` (the pseudorandom number generator of choice at compile time). From here, we declare some initial variables and then take in user input for the number of iterations for each  $c$ ,  $N$ , and the side length for each of the

Run #	Wallclock time [seconds]	Estimated area
1	64.938541	1.506632
2	64.930869	1.506632
3	64.930575	1.506632
Average	64.933328	1.506632

Table 1: Results from the serial version of the Mandelbrot set code in Problem 1, based on `exercise_1.c`.

cells, **h**. After initializing our random seed,<sup>2</sup> we use  $h$  to compute the number of horizontal and vertical lengths of the cells in the domain, i.e., how many different  $x$ - (real) and  $y$ - (imaginary) coordinates we have to define, where one pair of coordinates represents one cell, e.g., the center of that cell, the bottom-left corner, etc.—in the end, it does not matter. These numbers are **a** and **b**, respectively. While one could define a virtual 2D coordinate array here of dimensions  $\mathbf{a} \times \mathbf{b}$ , doing so is totally unnecessary, as when we loop through each coordinate in this array, we only need to evaluate the map at the value of  $c$  within that array and iterate that to see if the iterates stay within the set or leave. Thus, **a** and **b** only need to be used as the maximum bounds for the two iteration counters that correspond to the real- and imaginary-coordinates, respectively.

From here, we run the double for loop across the aforementioned two coordinates, starting the timer immediately before. For each coordinate  $c$  (randomly chosen in the given cell using `drand()`), we iterate  $z \leftarrow z^2 + c$  using the initial condition  $z = 0$  and carefully separating out the real and imaginary components of the map, denoted by **zx** and **zy** in the script, whilst similarly separating out the components of  $c$  in the same fashion. If  $|z| > 2$  (or equivalently,  $|z|^2 = (\Re z)^2 + (\Im z)^2 > 4$ , as we have in the script), then we stop iterating  $z \leftarrow z^2 + c$  and continue. Otherwise, if  $|z| > 2$  is never reached after **N** iterations, then we increment **NI**, the counter for the number of cells which yielded a value of  $c$  that resulted in iterations  $z \leftarrow z^2 + c$  that remained bounded within the region  $\{z \in \mathbb{C} : |z| \leq 2\}$  after **N** iterations.

After counting **NI** through the three loops aforementioned, we stop the timing routine; compute the wallclock time elapsed, **wcTime**; compute **NO** (the number of points which are outside of the Mandelbrot set) by noting that  $\mathbf{NI} + \mathbf{NO} = \mathbf{a} \times \mathbf{b}$ ; and then finally estimate the area of the interior<sup>3</sup> of the Mandelbrot set, **area**, using the provided formula. Both **wcTime** and **area** are printed as standard outputs.

## 2.2 Discussion

As shown in Table 1, the good news is that the estimated area produced by `exercise_1.c` is *precisely* the one which Professor Sherman said that he also got from whatever script he used: 1.506632. It should be important to note that the precision of the estimated area is limited by the size of  $\mathbf{a} \times \mathbf{b}$ , as we can only get as precise as **area** provides as a rational approximation, and twice the area of the region, **NI**, and **NO** are all multiples of  $1/4$ . However, the bad news is that our wallclock time is nearly 15 seconds over his! Not a problem; parallelization will make that decrease heavily.

<sup>2</sup>The only script this changes for is in `exercise_4.c`; this will be talked about later.

<sup>3</sup>There is an infinite set of measure zero relative to  $\mathbb{R}^2$  for Mandelbrot set; we call this a Julia set. It is a fractal, and its measure is only well-defined in a fractional-dimension between 1 and 2. We obviously are not counting that here, and it does not lie in the interior of the Mandelbrot set.

## 3 Exercise 2

Each part of this exercise is quite different from the rest in terms of what we do and test out, so we separate our results.

### 3.1 Part 1

#### 3.1.1 Codes

The relevant scripts I wrote for this part are `exercise_2-crit.c`, `exercise_2-atom.c`, `exercise_2-reduce.c`, and `drand-ts.c`. The first three feature the same serial process from Exercise 1, except the iterations all lie under a parallel region `#pragma omp parallel` and the for loops (note that the outer two loops are essentially independent of one another; embarrassingly parallel basically) are encased in a `#pragma omp for` region to ensure that they are appropriately divided amongst the provided number of threads, set according to the environment variable `OMP_NUM_THREADS` at runtime. The differences between them are outlined under Section 1.1 above; they each involve different ways of ensuring that preincrementing `NI` for each cell remains thread-safe and that multiple threads are not stepping on each other when that occurs.

`drand-ts.c` deserves more explanation. Intuitively speaking, to make the pseudorandom number generator thread safe, it would make sense to set a different seed separately for each thread and then ensure that each has its own privately-generated sequence of pseudorandom numbers to use—this is what I do for Exercise 4. However, since the default Makefile for ensuring a thread-safe pseudorandom number generation involves the same C script from before but just changes `drand.c`, it seems like that was not the intention for this problem. Hence, I do something else! `drand-ts.c` starts with declaring two static variables: The seed `seed` as before and a flag `key` (initialized to 1), both of which we set to be `threadprivate` to ensure that each thread has its own private copy, the privateness is global, and the privateness will pass over when we call `dsrand` and `drand` in our main scripts. `dsrand` remains unchanged, since `seed` is initialized before the for loop even starts.

`drand`, however, is changed drastically. Each thread passes through a critical section (so separately) which checks each of their copies of `key`. If `key` still has its initial value, this means that `drand` is being called for the first time. In that case, we shift the initial value of `seed` by the thread number to ensure that every thread starts with a different value of `seed`. But how do we ensure that two different initial seeds do not eventually converge to the same sequence of numbers, or remain close? This is because of the mathematics behind the provided pseudorandom number generator; it involves a linear transformation composed with a shift map, and shift maps are generally known to be chaotic.<sup>4</sup> In particular, chaotic systems are sensitive to initial conditions and neighboring trajectories diverge exponentially with time, which means that even though two nearby threads have initial values of `seed` which are only one apart, the sequences they generate should quickly (if not immediately, which is generally what we see in practice) diverge from each other and produce statistically independent sequences.

After ensuring that every thread has a different value of `seed`, `key` gets set back to zero, which unfortunately adds a serial overhead of having to recheck `key` serially every time each thread calls `drand` again. But anyways, from here, we essentially run the pseudorandom number generator as before in `drand.c`, except while protecting the linear part using `#pragma omp atomic` just in case, although that should be completely unnecessary since `seed` is `threadprivate`.

---

<sup>4</sup>See, for instance, [https://en.wikipedia.org/wiki/Dyadic\\_transformation](https://en.wikipedia.org/wiki/Dyadic_transformation) as a reference.

Run #	Critical <sup>(wallclock time in seconds ↓)</sup>	Atomic (‘’)	Reduction (‘’)
1	55.611877	55.566611	55.563506
2	55.607717	55.571660	55.564803
3	55.607665	55.578468	55.570601
Average	55.609086	55.572246	55.566303

Run #	Critical <sup>(approximated area ↓)</sup>	Atomic (‘’)	Reduction (‘’)
1	1.506736	1.506746	1.506636
2	1.506776	1.506706	1.506742
3	1.506646	1.506808	1.506842
Average	1.506719	1.506753	1.506740

Table 2: Results from multiple runs of `exercise_2-crit.c`, `exercise_2-atom.c`, and `exercise_2-reduce.c`, while using the non-thread safe pseudorandom number generator `drand.c` provided.

### 3.1.2 Discussion

Before talking about looking at the thread-safe pseudorandom number generator, you might be asking why I tested three different ways to protect the incrementing of `NI`. Generally speaking,<sup>5</sup> the order of efficiency (where “ $\leq$ ” means “less efficient than”) should generally be `critical`  $\leq$  `atomic`  $\leq$  `reduction`, since `critical` protects entire sections of code regardless (which completely serializes a process), `atomic` only protects atomic operations (which allows for some leeway since it essentially only serializes the memory access and writing part required by the process), and `reduction` uses a variety of so-called “reduction algorithms” which take some a number of private copies of the variable in question being “reduced” across the different threads and does the indicated operation. As you may have seen on Canvas, I asked Professor Sherman which one would be best to use, given that I could see a difference in performance due to the fact that each, while meeting the same end, work very differently from a technical standpoint. He told me that he does not expect a detectable difference, but told me to try it nonetheless and report on the results in my write-up; these results can be seen in Table 2. Table 2 does show that, on average, using `#pragma omp atomic` *does* result in a very, very slight improvement in runtime over `#pragma omp critical`, with even a smaller improvement over `#pragma omp for reduction(+:NI)`, but the apparent “improvement” is so negligibly small and is totally clouded out by the variance in runtimes across different runs that it is statistically insignificant.

That being said, the more pressing issue shown in Table 2 is the differences in the approximated area across different runs, which indicates a lack of thread-safety in the program. Fortunately, as shown in Table 3, the problem apparently was caused by the provided pseudorandom number generator, as using `drand-ts.c` caused the approximated area returned to be *exactly* the same across all runs. However, it *did* vary by around the same variance across different numbers of threads (although they were all equal within four significant digits of each other), even though within the same number of threads, the result remained deterministic. This probably is due to the pseudorandom number generator, which is not actually random. Depending on the initial seed, the generator will generate a different sequence of

<sup>5</sup>This was covered briefly in lecture, but there have been a number of questions related to this on Stack Exchange, e.g., <https://stackoverflow.com/questions/54186268/why-should-i-use-a-reduction-rather-than-an-atomic-variable>.

Run #	1 thread (wallclock time in seconds ↓)	2 threads (“”)	4 “” (“”)	12 “” (“”)	24 “” (“”)
1	65.555579	55.885873	29.883067	13.050817	7.648511
2	65.547570	55.885696	29.883321	13.051842	7.699972
3	65.555749	55.890747	29.882083	13.052164	7.622029
Average	65.552966	55.887439	29.882824	13.051608	7.656837

  

Run #	1 thread (approximated area ↓)	2 threads (“”)	4 “” (“”)	12 “” (“”)	24 “” (“”)
1	1.506632	1.506684	1.506710	1.506800	1.506836
2	1.506632	1.506684	1.506710	1.506800	1.506836
3	1.506632	1.506684	1.506710	1.506800	1.506836
Average	1.506632	1.506684	1.506710	1.506800	1.506836

Table 3: Results after running `exercise_2-reduce.c` with the thread-safe pseudorandom number generator `drand-ts.c`.

numbers which has roughly random statistics amongst itself (i.e., across the numbers in that sequence), but with the same seed, the generator still will give the same sequence of numbers. However, as we change the number of threads, the sequences used will be different, as each thread will use a different sequence of pseudorandom numbers, each with length and associated cell (i.e., which cell is using that pseudorandom number?) dependent on the number of threads. Thus, the only way to introduce any true “randomness” to our algorithm would actually be to vary the seed and/or the number of threads, but we did not want to do this in the first place since we wanted to check for thread-safety first.

## 3.2 Part 2

### 3.2.1 Codes

We just take `exercise_2-atom.c` from the previous part (as we showed, it does not really matter whether we use this one, `exercise_2-critical.c`, or `exercise_2-reduce.c`) and add a schedule at runtime option to `#pragma omp for`, thereby giving us `exercise_2-loops.c`.

### 3.2.2 Discussion

See Table 4 for the results of varying the scheduling options at runtime. The average approximated areas were all equal within four significant figures in all cases but one (12 threads using `schedule(static, 1)`), but even that case barely missed being rounded up for the third decimal places. As for the wallclock time in each case, `schedule(dynamic, 250)` was the slowest by far, with `schedule(static, 100)` being next slowest. The rest were all roughly equal regardless of the number of threads, but `schedule(guided)` was consistently a bit faster than both `schedule(static, 1)` and `schedule(dynamic)`.

We can briefly explain what each scheduling option does to understand the disparities in performance. Static scheduling involves already planning the way in which the threads will share the taskload before runtime, generally using a round-robin algorithm.<sup>6</sup> Every thread shares the work with equal opportunity and size. Dynamic scheduling does this on a first-come, first-serve basis—the work queue gives a chunk to each thread, and then when one is finished, it grabs the next from the queue. Finally, guided scheduling is similar to

<sup>6</sup><https://stackoverflow.com/questions/10850155/whats-the-difference-between-static-and-dynamic-schedule-in-openssl>

Schedule option	2 threads (average wallclock time over three runs in seconds ↓)	4 “” (“”)	12 “” (“”)	24 “” (“”)
<code>schedule(static, 1)</code>	33.304340	17.332725	6.630209	4.209467
<code>schedule(static, 100)</code>	33.341971	19.729973	8.590298	7.648828
<code>schedule(dynamic)</code>	33.313329	17.360312	6.658384	4.238770
<code>schedule(dynamic, 250)</code>	35.023132	21.629599	16.838498	16.832024
<code>schedule(guided)</code>	33.010117	17.097930	6.579577	4.163306

Schedule option	2 threads (average approximated area over three runs ↓)	4 “” (“”)	12 “” (“”)	24 “” (“”)
<code>schedule(static, 1)</code>	1.506636	1.506886	1.506466	1.506796
<code>schedule(static, 100)</code>	1.506540	1.506686	1.506690	1.506662
<code>schedule(dynamic)</code>	1.506719	1.506747	1.506774	1.506709
<code>schedule(dynamic, 250)</code>	1.506600	1.506778	1.506672	1.506794
<code>schedule(guided)</code>	1.506804	1.506621	1.506641	1.506697

Table 4: Results after running `exercise_2-loops.c` (uses `#pragma omp atomic` to protect the counter for points in the Mandelbrot set, NI) while varying scheduling options and the number of threads.

`dynamic`,<sup>7</sup> except it starts with the chunk size large and then gradually decreases it to better handle load imbalance across different iterations. Hence, in a case where each iteration takes roughly the same work, we might expect static scheduling to outperform both dynamic and guided scheduling, because the extra overhead introduced by the latter two approaches is not compensated by the imbalance of work between iterations. However, the converse should also be true; ours appears to be one such case. Although, the imbalance of work between iteration is still not enough for dynamic scheduling to clearly outperform static scheduling, we see the outperformance of guided scheduling.

To understand why `schedule(dynamic, 250)` and `schedule(static, 100)` both underperformed, we have to understand the number next to them. That number is the chunk size. When dividing the iterations of a for loop, these are decided based on this size. Hence, if the chunk size is too large, then relative to the number of iterations, the ratio of work per thread goes up, which ruins the efficiency of the process. This could also explain why the runtime actually *saturated* for `schedule(dynamic, 250)` between 12 and 24 threads, as the chunks being allocated to each thread were so large that there was hardly a need for that many more threads.

Finally, this all explains the differences in estimated area across runs for dynamic and guided scheduling. Our script did not suddenly lose its thread-safety—rather, the scheduling across different threads will change depending on many external factors to the CPU(s) being used, as it is determined dynamically during runtime.

### 3.3 Part 3

#### 3.3.1 Codes

We introduced a collapse clause, `collapse(2)`, to ensure that both of the outer two loops (hence the 2) were being appropriately parallelized rather than just the outermost one. This resulted in `exercise_2-collapse.c` being created from `exercise_2-loops.c`.

<sup>7</sup><https://stackoverflow.com/questions/42970700/openmp-dynamic-vs-guided-scheduling>

Schedule option	24 threads (average wallclock time over three runs in seconds ↓)	24 threads (average approximated area over three runs ↓)
<code>schedule(static, 100)</code>	4.202319	1.506652
<code>schedule(dynamic, 250)</code>	4.220878	1.506643
<code>schedule(guided)</code>	4.134117	1.506791

Table 5: Results after running `exercise_2-collapse.c`, which uses both `#pragma omp atomic` to protect the counter for points in the Mandelbrot set, `NI`, and the clause `collapse(2)` to collapse the two outermost nested for loops in the algorithm into a single one to be divided across different threads. These are done across three different schedule options to compare the results.

# of threads → Part # ↓	1 (average wallclock time over three runs in seconds ↓)	2 (″)	4 (″)	12 (″)	24 (″)
1 (cell tasks)	66.095441	33.508349	17.495800	6.672314	4.026237
2 (row tasks)	50.139497	25.302024	13.345016	5.251697	3.591382
3 (shared cell tasks)	66.164702	34.080561	18.921334	7.895722	5.271672
3 (shared row tasks)	51.309117	26.124081	13.270310	5.124749	3.456234

  

# of threads → Part # ↓	1 (average approximated area over three runs ↓)	2 (″)	4 (″)	12 (″)	24 (″)
1 (cell tasks)	1.506632	1.506675	1.506735	1.506746	1.506638
2 (column tasks)	1.506638	1.506780	1.506705	1.506687	1.506716
3 (shared cell tasks)	1.506632	1.506762	1.506679	1.506735	1.506655
3 (shared column tasks)	1.506638	1.506823	1.506763	1.506733	1.506751

Table 6: Results after running each of the scripts from Exercise 3.

### 3.3.2 Discussion

Understandably, as Table (5) demonstrates, there was a noticeable speed-up in the efficiency of the code once we parallelized across both the real and imaginary coordinates rather than just one. However, this speed-up was especially noticeable for our formerly underperforming scheduling options `schedule(static, 100)` and `schedule(dynamic, 250)`. This is because, after collapsing the outer two loops, the number of chunks has been multiplied by a factor of `b` from the previous part (i.e., without collapsing). Thus, using large chunks from the start is no longer as limiting and actually appears to help with the overhead of assigning new chunks. However, `guided` already chooses large chunks from the start, so the speed-up there was miniscule compared to Part 2 above.

## 4 Exercise 3

### 4.1 Codes

The relevant codes are `exercise_3.c` for Part 1, `exercise_3-row.c` for Part 2, and `exercise_3-tasks-s` and `exercise_3-tasks-row-shared.c` for Part 3. The contents of these was already sufficiently explained in Section 1.1.

## 4.2 Discussion

**Note:** For Part 3 of this exercise, I was not sure if Professor Sherman wanted us to write a script for all threads to share tasks across the cells or columns, although the names of the scripts in the Makefile template provided appear to suggest the latter (with “row” mistakenly used instead of “column”). Hence, I wrote two scripts which do both: `exercise_3-tasks-shared.c` and `exercise_3-tasks-row-shared.c`.

As shown in Table 6, the worst performers in either case were those which involved assigning tasks for each cell separately. Unlike when working with `#pragma omp for`, one has to consider the additional overhead of assigning and closing out tasks, and we added a lot of that unnecessarily when we assigned a task based on each cell. We also have to consider the added overhead of how either one thread (`exercise_3.c`) or all threads (`exercise_3-tasks-shared.c`) have to go through the entire loop, and in the latter, as mentioned earlier in this report, it ends up acting like a busy-wait.

However, conversely, if we create only one task per column, then we reduce the overhead of dealing with that many tasks being assigned, which enhances the performance. That being said, perhaps because all of the threads have to deal with assigning tasks rather than just one (which perhaps adds some latency into the process), while one would think that maybe, sharing the task creation might help the program, it only does so very slightly. But nonetheless, for the same number of threads, using tasks actually leads to quicker programs than using `#pragma omp for` as in Exercise 2. But of course, the disadvantage (or perhaps advantage, in light of the inherent and expected randomness to the Monte Carlo algorithm being used) is that the program is no longer deterministic due to the inherently unpredictable nature of deciding which thread assigns tasks to which thread(s).

## 5 Exercise 4

### 5.1 Codes

Our two best performing codes from Exercise 2 involved both the collapse clause `collapse(2)` and either one of the static scheduling options or `schedule(guided)`, depending on whether or not one wants to take determinism into account. `exercise_4.c` is a modified version of `exercise_2-collapse.c`, except in this case, `dsrand` is called privately within each thread separately. By doing this, `drand-ts-2.c` does not need a threadprivate flag variable like `drand-ts.c` had, and instead, one only needs to ensure that `seed` is threadprivate and updates in a thread-safe fashion. Why does `seed` still need to be threadprivate? We will answer that below.

### 5.2 Discussion

If `seed` were not threadprivate, then we would run into the shortcoming brought up in the question: “Multiple threads may wind up using the same sequence of random numbers.” Technically, my original corrected, thread-safe pseudorandom number generator, `drand-ts.c`, already accomplishes this by defining `seed` as a threadprivate variable, since this causes every thread *globally* to have its own copy of `seed`.<sup>8</sup> However, I did not want to get points docked off for not doing Exercise 4 separately, so I merely thought I would introduce

---

<sup>8</sup>By global here, we mean that, if there were another parallel region in `exercise_4.c`, then each thread in that parallel region would also have a private copy of `seed` with the same value it had immediately before entering the parallel region.

Run #	(wallclock time in seconds ↓)	(approximated area ↓)	Run #	(wallclock time in seconds ↓)	(approximated area ↓)
1	4.104567	1.506830	1	4.195226	1.506642
2	4.118794	1.506780	2	4.230573	1.506642
3	4.099058	1.506678	3	4.204444	1.506642
Average	4.107473	1.506763	Average	4.210081	1.506642

Table 7: The results of the program generated by compiling `exercise_4.c` with the second version of the thread-safe pseudorandom number generator, `drand-ts-2.c`. The left table corresponds to `schedule` option `guided` and the right one to `static,1`. Both cases were ran using 24 threads.

here what I had originally intended to do in Exercise 2, which was to assign a different seed using `dsrand` instead to each thread. However, this must be done in `exercise_4.c`, and this is done via defining a critical region to ensure that the function call `omp_get_thread_num()` is thread-safe.

The results are shown in Table 7. As shown there, `drand-ts-2.c` is also thread-safe, since running the `schedule` option `static` gives us the exact same approximated area for every run. As for the runtime, we get roughly the same results as we did for Exercise 2, albeit maybe with the slightest improvement. We would expect at least some improvement, since the threads no longer have to run through a critical region and check the flag variable `key` each time `drand` is called.

## 6 Appendix

Some of the lines below are too long for the page, but these lines are not relevant for what will be actually used in the report, so if you must, you can check them directly by looking at the corresponding files in the submitted `.gz` file.

### 6.1 MandelbrotFinalTest-27030939.out

Currently Loaded Modules:

```

1) StdEnv                               (S)  4) binutils/2.35-GCCcore-10.2.0      7) UCX/1.9.0-C
2) GCCcore/10.2.0                      5) iccifort/2020.4.304              8) impi/2019.9
3) zlib/1.2.11-GCCcore-10.2.0          6) numactl/2.0.13-GCCcore-10.2.0    9) iimpi/2020b

```

Where:

S: Module is Sticky, requires `--force` to unload or purge

```

rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma
make mandseq
icc -g -O3 -xHost -fno-alias -std=c99 -I/gpfs/gibbs/project/cpsc424/shared/utils/timing

```

Serial version

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 64.938541 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 64.930869 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 64.930575 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

## 6.2 MandelbrotFinalTest-27067660.out

Currently Loaded Modules:

1) StdEnv	(S)	4) binutils/2.35-GCCcore-10.2.0	7) UCX/1.9.0-C
2) GCCcore/10.2.0		5) iccifort/2020.4.304	8) impi/2019.9
3) zlib/1.2.11-GCCcore-10.2.0		6) numactl/2.0.13-GCCcore-10.2.0	9) iimpi/2020b

Where:

S: Module is Sticky, requires --force to unload or purge

OpenMP version with original drand.c

```
rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma
make mandomp-critical
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
make mandomp-atomic
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
make mandomp-reduction
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
Number of threads = 2
OMP_SCHEDULE =
```

Run 1:

Using critical:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 55.611877 s.  
Approximated area of the Mandelbrot Set is 1.5067360000000000.

Using atomic:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 55.566611 s.  
Approximated area of the Mandelbrot Set is 1.5067460000000000.

Using reduction:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 55.563506 s.  
Approximated area of the Mandelbrot Set is 1.5066360000000000.

Run 2:

Using critical:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 55.607717 s.

Approximated area of the Mandelbrot Set is 1.5067760000000000.

Using atomic:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 55.571660 s.

Approximated area of the Mandelbrot Set is 1.5067060000000000.

Using reduction:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 55.564803 s.

Approximated area of the Mandelbrot Set is 1.5067420000000000.

Run 3:

Using critical:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 55.607665 s.

Approximated area of the Mandelbrot Set is 1.5066460000000000.

Using atomic:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 55.578468 s.

Approximated area of the Mandelbrot Set is 1.5068080000000000.

Using reduction:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 55.570601 s.

Approximated area of the Mandelbrot Set is 1.5068420000000000.

```
rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma
make ts-random
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
make mandomp-ts
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
```

OpenMP version with threadsafe drand

Number of threads = 1

OMP\_SCHEDULE =

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 65.555579 s.

Approximated area of the Mandelbrot Set is 1.5066320000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 65.547570 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 65.555749 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Number of threads = 2  
OMP\_SCHEDULE =  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 55.885873 s.  
Approximated area of the Mandelbrot Set is 1.5066840000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 55.885696 s.  
Approximated area of the Mandelbrot Set is 1.5066840000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 55.890747 s.  
Approximated area of the Mandelbrot Set is 1.5066840000000000.

Number of threads = 4  
OMP\_SCHEDULE =  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 29.883067 s.  
Approximated area of the Mandelbrot Set is 1.5067100000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 29.883321 s.  
Approximated area of the Mandelbrot Set is 1.5067100000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 29.882083 s.  
Approximated area of the Mandelbrot Set is 1.5067100000000000.

Number of threads = 12  
OMP\_SCHEDULE =  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 13.050817 s.  
Approximated area of the Mandelbrot Set is 1.5068000000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 13.051842 s.  
Approximated area of the Mandelbrot Set is 1.5068000000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 13.052164 s.  
Approximated area of the Mandelbrot Set is 1.5068000000000000.

Number of threads = 24  
OMP\_SCHEDULE =

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 7.648511 s.  
Approximated area of the Mandelbrot Set is 1.5068360000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 7.699972 s.  
Approximated area of the Mandelbrot Set is 1.5068360000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 7.622029 s.  
Approximated area of the Mandelbrot Set is 1.5068360000000000.

```
rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma
make ts-random
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
make mandomp-loops
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
```

Performance runs for thread-safe OpenMP with loops, Part 2a: using schedule(static, 1)

Number of threads = 2  
OMP\_SCHEDULE = static,1

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.315344 s.  
Approximated area of the Mandelbrot Set is 1.5066360000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.293671 s.  
Approximated area of the Mandelbrot Set is 1.5066360000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.304005 s.  
Approximated area of the Mandelbrot Set is 1.5066360000000000.

Number of threads = 4  
OMP\_SCHEDULE = static,1

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.339365 s.  
Approximated area of the Mandelbrot Set is 1.5068860000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.312522 s.  
Approximated area of the Mandelbrot Set is 1.5068860000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.346287 s.  
Approximated area of the Mandelbrot Set is 1.5068860000000000.

Number of threads = 12  
OMP\_SCHEDULE = static,1

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.613174 s.  
Approximated area of the Mandelbrot Set is 1.5064660000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.648118 s.  
Approximated area of the Mandelbrot Set is 1.5064660000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.629335 s.  
Approximated area of the Mandelbrot Set is 1.5064660000000000.

Number of threads = 24  
OMP\_SCHEDULE = static,1

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.212173 s.  
Approximated area of the Mandelbrot Set is 1.5067960000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.190741 s.  
Approximated area of the Mandelbrot Set is 1.5067960000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.225488 s.  
Approximated area of the Mandelbrot Set is 1.5067960000000000.

Performance runs for thread-safe OpenMP with loops, Part 2b: using schedule(static,100)

Number of threads = 2  
OMP\_SCHEDULE = static,100  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.325668 s.  
Approximated area of the Mandelbrot Set is 1.5065400000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.353241 s.  
Approximated area of the Mandelbrot Set is 1.5065400000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.347004 s.  
Approximated area of the Mandelbrot Set is 1.5065400000000000.

Number of threads = 4  
OMP\_SCHEDULE = static,100  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 19.737806 s.  
Approximated area of the Mandelbrot Set is 1.5066860000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 19.717525 s.  
Approximated area of the Mandelbrot Set is 1.5066860000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 19.734588 s.  
Approximated area of the Mandelbrot Set is 1.5066860000000000.

Number of threads = 12  
OMP\_SCHEDULE = static,100  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 8.583702 s.

Approximated area of the Mandelbrot Set is 1.5066900000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 8.590415 s.

Approximated area of the Mandelbrot Set is 1.5066900000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 8.596776 s.

Approximated area of the Mandelbrot Set is 1.5066900000000000.

Number of threads = 24

OMP\_SCHEDULE = static,100

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 7.652124 s.

Approximated area of the Mandelbrot Set is 1.5066620000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 7.644691 s.

Approximated area of the Mandelbrot Set is 1.5066620000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 7.649670 s.

Approximated area of the Mandelbrot Set is 1.5066620000000000.

Performance runs for thread-safe OpenMP with loops, Part 2c: using schedule(dynamic)

Number of threads = 2

OMP\_SCHEDULE = dynamic

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 33.318209 s.

Approximated area of the Mandelbrot Set is 1.5067080000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 33.307215 s.

Approximated area of the Mandelbrot Set is 1.5067160000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 33.314562 s.

Approximated area of the Mandelbrot Set is 1.5067320000000000.

Number of threads = 4  
OMP\_SCHEDULE = dynamic  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.354504 s.  
Approximated area of the Mandelbrot Set is 1.5068080000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.362222 s.  
Approximated area of the Mandelbrot Set is 1.5067280000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.364210 s.  
Approximated area of the Mandelbrot Set is 1.5067040000000000.

Number of threads = 12  
OMP\_SCHEDULE = dynamic  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.703087 s.  
Approximated area of the Mandelbrot Set is 1.5067060000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.614807 s.  
Approximated area of the Mandelbrot Set is 1.5069220000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.657258 s.  
Approximated area of the Mandelbrot Set is 1.5066940000000000.

Number of threads = 24  
OMP\_SCHEDULE = dynamic  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.229479 s.  
Approximated area of the Mandelbrot Set is 1.5066080000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.257172 s.  
Approximated area of the Mandelbrot Set is 1.5067000000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 4.229658 s.  
Approximated area of the Mandelbrot Set is 1.5068200000000000.

Performance runs for thread-safe OpenMP with loops, Part 2d: using schedule(dynamic,250)

Number of threads = 2  
OMP\_SCHEDULE = dynamic,250

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 35.058941 s.  
Approximated area of the Mandelbrot Set is 1.5066000000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 34.997060 s.  
Approximated area of the Mandelbrot Set is 1.5066000000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 35.013396 s.  
Approximated area of the Mandelbrot Set is 1.5066000000000000.

Number of threads = 4  
OMP\_SCHEDULE = dynamic,250

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 21.628134 s.  
Approximated area of the Mandelbrot Set is 1.5067620000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 21.632976 s.  
Approximated area of the Mandelbrot Set is 1.5067860000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 21.627687 s.  
Approximated area of the Mandelbrot Set is 1.5067860000000000.

Number of threads = 12  
OMP\_SCHEDULE = dynamic,250

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 16.841268 s.  
Approximated area of the Mandelbrot Set is 1.5067060000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 16.836342 s.  
Approximated area of the Mandelbrot Set is 1.5066180000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 16.837885 s.  
Approximated area of the Mandelbrot Set is 1.5066920000000000.

Number of threads = 24  
OMP\_SCHEDULE = dynamic,250

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 16.833312 s.  
Approximated area of the Mandelbrot Set is 1.5068120000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 16.835389 s.  
Approximated area of the Mandelbrot Set is 1.5068980000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 16.827372 s.  
Approximated area of the Mandelbrot Set is 1.5066720000000000.

Performance runs for thread-safe OpenMP with loops, Part 2e: using schedule(guided)

Number of threads = 2  
OMP\_SCHEDULE = guided

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 32.999778 s.  
Approximated area of the Mandelbrot Set is 1.5068040000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.014064 s.  
Approximated area of the Mandelbrot Set is 1.5068040000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.016508 s.  
Approximated area of the Mandelbrot Set is 1.5068040000000000.

Number of threads = 4  
OMP\_SCHEDULE = guided

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.095655 s.  
Approximated area of the Mandelbrot Set is 1.5066060000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.102738 s.  
Approximated area of the Mandelbrot Set is 1.5066260000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.095396 s.  
Approximated area of the Mandelbrot Set is 1.5066300000000000.

Number of threads = 12  
OMP\_SCHEDULE = guided

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.558675 s.  
Approximated area of the Mandelbrot Set is 1.5065900000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.597257 s.  
Approximated area of the Mandelbrot Set is 1.5066200000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.582799 s.  
Approximated area of the Mandelbrot Set is 1.5067140000000000.

Number of threads = 24  
OMP\_SCHEDULE = guided

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.160366 s.  
Approximated area of the Mandelbrot Set is 1.5067780000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.144926 s.  
Approximated area of the Mandelbrot Set is 1.5066020000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.184627 s.  
Approximated area of the Mandelbrot Set is 1.5067120000000000.

```
rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma
make ts-random
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
make mandomp-collapse
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
```

Performance runs for thread-safe OpenMP with collapse

```
Number of threads = 24
OMP_SCHEDULE = static,100
```

Run 1:

```
Seed = 12344. RAND_MAX = 2147483647.
Wallclock time elapsed for algorithm was 4.247929 s.
Approximated area of the Mandelbrot Set is 1.5066520000000000.
```

Run 2:

```
Seed = 12344. RAND_MAX = 2147483647.
Wallclock time elapsed for algorithm was 4.198158 s.
Approximated area of the Mandelbrot Set is 1.5066520000000000.
```

Run 3:

```
Seed = 12344. RAND_MAX = 2147483647.
Wallclock time elapsed for algorithm was 4.160871 s.
Approximated area of the Mandelbrot Set is 1.5066520000000000.
```

```
Number of threads = 24
OMP_SCHEDULE = dynamic,250
```

Run 1:

```
Seed = 12344. RAND_MAX = 2147483647.
Wallclock time elapsed for algorithm was 4.190965 s.
Approximated area of the Mandelbrot Set is 1.5065660000000000.
```

Run 2:

```
Seed = 12344. RAND_MAX = 2147483647.
Wallclock time elapsed for algorithm was 4.227631 s.
Approximated area of the Mandelbrot Set is 1.5066180000000000.
```

Run 3:

```
Seed = 12344. RAND_MAX = 2147483647.
Wallclock time elapsed for algorithm was 4.244038 s.
Approximated area of the Mandelbrot Set is 1.5067460000000000.
```

```
Number of threads = 24
OMP_SCHEDULE = guided
```

Run 1:

```
Seed = 12344. RAND_MAX = 2147483647.
Wallclock time elapsed for algorithm was 4.155485 s.
Approximated area of the Mandelbrot Set is 1.5067580000000000.
```

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.143925 s.  
Approximated area of the Mandelbrot Set is 1.5067880000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.102942 s.  
Approximated area of the Mandelbrot Set is 1.5068260000000000.

### 6.3 MandelbrotFinalTest-27070337.out

The following have been reloaded with a version change:

- |   |                   |
|---|-------------------|
| 1) GCCcore/12.2.0 => GCCcore/10.2.0                             | 6) impi/2021.7.1- |
| 2) UCX/1.13.1-GCCcore-12.2.0 => UCX/1.9.0-GCCcore-10.2.0        | 7) intel/2022b => |
| 3) binutils/2.39-GCCcore-12.2.0 => binutils/2.35-GCCcore-10.2.0 | 8) numactl/2.0.16 |
| 4) iimpi/2022b => iimpi/2020b                                   | 9) zlib/1.2.12-GC |
| 5) imkl/2022.2.1 => imkl/2020.4.304-iimpi-2020b                 |                   |

Currently Loaded Modules:

- |                               |     |                                  |                |
|-------------------------------|-----|----------------------------------|----------------|
| 1) StdEnv                     | (S) | 4) binutils/2.35-GCCcore-10.2.0  | 7) UCX/1.9.0-C |
| 2) GCCcore/10.2.0             |     | 5) iccifort/2020.4.304           | 8) impi/2019.9 |
| 3) zlib/1.2.11-GCCcore-10.2.0 |     | 6) numactl/2.0.13-GCCcore-10.2.0 | 9) iimpi/2020b |

Where:

S: Module is Sticky, requires --force to unload or purge

```
rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma
make ts-random
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
make mandomp-tasks
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
```

Performance runs for thread-safe OpenMP with tasks

Number of threads = 1  
OMP\_SCHEDULE = dynamic,250

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 66.111314 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 66.071666 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 66.103343 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Number of threads = 2  
OMP\_SCHEDULE = dynamic,250  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.501829 s.  
Approximated area of the Mandelbrot Set is 1.5066940000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.517486 s.  
Approximated area of the Mandelbrot Set is 1.5066440000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 33.505733 s.  
Approximated area of the Mandelbrot Set is 1.5066880000000000.

Number of threads = 4  
OMP\_SCHEDULE = dynamic,250  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.578185 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.552532 s.  
Approximated area of the Mandelbrot Set is 1.5067640000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 17.356684 s.  
Approximated area of the Mandelbrot Set is 1.5068100000000000.

Number of threads = 12  
OMP\_SCHEDULE = dynamic,250  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.672569 s.  
Approximated area of the Mandelbrot Set is 1.5067460000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.674652 s.  
Approximated area of the Mandelbrot Set is 1.5066720000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 6.669721 s.  
Approximated area of the Mandelbrot Set is 1.5068200000000000.

Number of threads = 24  
OMP\_SCHEDULE = dynamic,250

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.022631 s.  
Approximated area of the Mandelbrot Set is 1.5065700000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.024561 s.  
Approximated area of the Mandelbrot Set is 1.5066140000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.031518 s.  
Approximated area of the Mandelbrot Set is 1.5067300000000000.

```
rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma
make ts-random
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
make mandomp-tasks-row
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
```

Performance runs for thread-safe OpenMP with row tasks

Number of threads = 1  
OMP\_SCHEDULE = dynamic,250  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 50.180743 s.  
Approximated area of the Mandelbrot Set is 1.5066380000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 50.306754 s.  
Approximated area of the Mandelbrot Set is 1.5066380000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 49.930994 s.  
Approximated area of the Mandelbrot Set is 1.5066380000000000.

Number of threads = 2  
OMP\_SCHEDULE = dynamic,250

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 25.231568 s.  
Approximated area of the Mandelbrot Set is 1.5067200000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 25.330118 s.  
Approximated area of the Mandelbrot Set is 1.5068740000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 25.344386 s.  
Approximated area of the Mandelbrot Set is 1.5067460000000000.

Number of threads = 4  
OMP\_SCHEDULE = dynamic,250

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 13.331219 s.  
Approximated area of the Mandelbrot Set is 1.5067480000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 13.371712 s.  
Approximated area of the Mandelbrot Set is 1.5067280000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 13.332118 s.  
Approximated area of the Mandelbrot Set is 1.5066400000000000.

Number of threads = 12  
OMP\_SCHEDULE = dynamic,250

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 5.279719 s.  
Approximated area of the Mandelbrot Set is 1.5066840000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 5.216143 s.  
Approximated area of the Mandelbrot Set is 1.5066820000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 5.259230 s.  
Approximated area of the Mandelbrot Set is 1.5066960000000000.

Number of threads = 24  
OMP\_SCHEDULE = dynamic,250

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 3.584152 s.  
Approximated area of the Mandelbrot Set is 1.5067820000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 3.590570 s.  
Approximated area of the Mandelbrot Set is 1.5067380000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 3.599423 s.  
Approximated area of the Mandelbrot Set is 1.5066280000000000.

```
rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma
make ts-random
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
make mandomp-tasks-shared
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
```

Performance runs for thread-safe OpenMP with shared task creation

Number of threads = 1  
OMP\_SCHEDULE = dynamic,250

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 66.157297 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 66.167060 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 66.169748 s.  
Approximated area of the Mandelbrot Set is 1.5066320000000000.

Number of threads = 2  
OMP\_SCHEDULE = dynamic,250  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 34.054638 s.  
Approximated area of the Mandelbrot Set is 1.5068260000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 34.059233 s.  
Approximated area of the Mandelbrot Set is 1.5068000000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 34.127812 s.  
Approximated area of the Mandelbrot Set is 1.5066600000000000.

Number of threads = 4  
OMP\_SCHEDULE = dynamic,250  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 18.935712 s.  
Approximated area of the Mandelbrot Set is 1.5065820000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 18.923550 s.  
Approximated area of the Mandelbrot Set is 1.5066440000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 18.904741 s.  
Approximated area of the Mandelbrot Set is 1.5068100000000000.

Number of threads = 12  
OMP\_SCHEDULE = dynamic,250  
Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 7.764758 s.  
Approximated area of the Mandelbrot Set is 1.5067200000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 7.972390 s.  
Approximated area of the Mandelbrot Set is 1.5067660000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 7.950017 s.

Approximated area of the Mandelbrot Set is 1.5067180000000000.

Number of threads = 24

OMP\_SCHEDULE = dynamic,250

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 5.270970 s.

Approximated area of the Mandelbrot Set is 1.5066220000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 5.249984 s.

Approximated area of the Mandelbrot Set is 1.5065500000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 5.294062 s.

Approximated area of the Mandelbrot Set is 1.5067920000000000.

rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma

make ts-random

icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti

make mandomp-tasks-row-shared

icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti

Performance runs for thread-safe OpenMP with row tasks and shared task creation

Number of threads = 1

OMP\_SCHEDULE = dynamic,250

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 51.326269 s.

Approximated area of the Mandelbrot Set is 1.5066380000000000.

Run 2:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 51.301611 s.

Approximated area of the Mandelbrot Set is 1.5066380000000000.

Run 3:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 51.299470 s.

Approximated area of the Mandelbrot Set is 1.5066380000000000.

Number of threads = 2

OMP\_SCHEDULE = dynamic,250

Run 1:

Seed = 12344. RAND\_MAX = 2147483647.

Wallclock time elapsed for algorithm was 26.143040 s.  
Approximated area of the Mandelbrot Set is 1.5068760000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 26.111609 s.  
Approximated area of the Mandelbrot Set is 1.5068180000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 26.117595 s.  
Approximated area of the Mandelbrot Set is 1.5067740000000000.

Number of threads = 4  
OMP\_SCHEDULE = dynamic,250

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 13.146293 s.  
Approximated area of the Mandelbrot Set is 1.5067480000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 13.252018 s.  
Approximated area of the Mandelbrot Set is 1.5067140000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 13.412620 s.  
Approximated area of the Mandelbrot Set is 1.5068260000000000.

Number of threads = 12  
OMP\_SCHEDULE = dynamic,250

Run 1:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 5.326563 s.  
Approximated area of the Mandelbrot Set is 1.5067900000000000.

Run 2:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 4.708591 s.  
Approximated area of the Mandelbrot Set is 1.5067320000000000.

Run 3:  
Seed = 12344. RAND\_MAX = 2147483647.  
Wallclock time elapsed for algorithm was 5.339092 s.  
Approximated area of the Mandelbrot Set is 1.5066780000000000.

Number of threads = 24  
OMP\_SCHEDULE = dynamic,250

```
Run 1:
Seed = 12344. RAND_MAX = 2147483647.
Wallclock time elapsed for algorithm was 3.280100 s.
Approximated area of the Mandelbrot Set is 1.5067320000000000.
```

```
Run 2:
Seed = 12344. RAND_MAX = 2147483647.
Wallclock time elapsed for algorithm was 3.402947 s.
Approximated area of the Mandelbrot Set is 1.5067740000000000.
```

```
Run 3:
Seed = 12344. RAND_MAX = 2147483647.
Wallclock time elapsed for algorithm was 3.685654 s.
Approximated area of the Mandelbrot Set is 1.5067480000000000.
```

```
rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma
```

## 6.4 MandelbrotFinalTest-27072158.out

The following have been reloaded with a version change:

- |   |                   |
|---|-------------------|
| 1) GCCcore/12.2.0 => GCCcore/10.2.0                             | 6) impi/2021.7.1- |
| 2) UCX/1.13.1-GCCcore-12.2.0 => UCX/1.9.0-GCCcore-10.2.0        | 7) intel/2022b => |
| 3) binutils/2.39-GCCcore-12.2.0 => binutils/2.35-GCCcore-10.2.0 | 8) numactl/2.0.16 |
| 4) iimpi/2022b => iimpi/2020b                                   | 9) zlib/1.2.12-GC |
| 5) imkl/2022.2.1 => imkl/2020.4.304-iimpi-2020b                 |                   |

Currently Loaded Modules:

- |                               |     |                                  |                |
|-------------------------------|-----|----------------------------------|----------------|
| 1) StdEnv                     | (S) | 4) binutils/2.35-GCCcore-10.2.0  | 7) UCX/1.9.0-C |
| 2) GCCcore/10.2.0             |     | 5) iccifort/2020.4.304           | 8) impi/2019.9 |
| 3) zlib/1.2.11-GCCcore-10.2.0 |     | 6) numactl/2.0.13-GCCcore-10.2.0 | 9) iimpi/2020b |

Where:

S: Module is Sticky, requires --force to unload or purge

```
rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma
make ts-random-2
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
make mandomp-collapse-ts
icc -g -O3 -xHost -fno-alias -std=c99 -qopenmp -I/gpfs/gibbs/project/cpsc424/shared/uti
```

OpenMP version with second version of threadsafe drand

Number of threads = 24

OMP\_SCHEDULE = guided

Run 1:

Wallclock time elapsed for algorithm was 4.104567 s.  
Approximated area of the Mandelbrot Set is 1.5068300000000000.

Run 2:  
Wallclock time elapsed for algorithm was 4.118794 s.  
Approximated area of the Mandelbrot Set is 1.5067800000000000.

Run 3:  
Wallclock time elapsed for algorithm was 4.099058 s.  
Approximated area of the Mandelbrot Set is 1.5066780000000000.

Number of threads = 24  
OMP\_SCHEDULE = static,1

Run 1:  
Wallclock time elapsed for algorithm was 4.195226 s.  
Approximated area of the Mandelbrot Set is 1.5066420000000000.

Run 2:  
Wallclock time elapsed for algorithm was 4.230573 s.  
Approximated area of the Mandelbrot Set is 1.5066420000000000.

Run 3:  
Wallclock time elapsed for algorithm was 4.204444 s.  
Approximated area of the Mandelbrot Set is 1.5066420000000000.

rm -f mandseq mandomp-critical mandomp-atomic mandomp-reduction ts-random mandomp-ts ma