CPSC 524 Assignment 2: Area of the Mandelbrot Set

Rami Pellumbi*

October 15, 2023

*M.S., Statistics & Data Science

1 Introduction

This assignment focuses on assessing the performance gains achievable through the application of loop directives and tasks in parallelizing the computation of the Mandelbrot set area. The basis for this study is a serial algorithm designed for this task, detailed as follows:

1. We begin by defining a rectangular region in the complex plane, denoted as $\mathbf{R}$. This region is bounded by the lower-left corner $-2.0 + 0.0i$ and the upper-right corner $0.5 + 1.25i$.

2. This region $\mathbf{R}$ is then discretized into a grid of cells, each having a side length of 0.001 units. The set of all possible lower-left corner coordinates of these cells is represented by T, defined as:

$$T := \{(x, y) : x \in X, y \in Y\},$$

   where

$$X = \{x \in \mathbb{R} : x = -2.0 + n * 0.001, -2.0 \leq x < 0.5, n \in \mathbb{N}\},$$
$$Y = \{y \in \mathbb{R} : y = 0.0 + n * 0.001, 0.0 \leq y < 1.25, n \in \mathbb{N}\}.$$

   A cell is uniquely defined by its lower-left corner coordinate $(x, y) \in X \times Y$. We denote such a cell by $C_{x,y}$, and the set of all such cells is $C := \{C_{x,y} : (x, y) \in T\}$.

   A few clarifying remarks:

   - The cardinality of X is
$$|X| = \frac{0.5 - (-2.0)}{0.001} = 2500.$$
   - The cardinality of Y is
$$|Y| = \frac{1.25 - 0.0}{0.001} = 1250.$$
   - The cardinality of T is
$$|T| = |X| \cdot |Y| = 2500 \cdot 1250 = 3125000.$$

3. For each cell $C_{x,y} \in C$, a random point c within the cell is generated.

4. The Mandelbrot update $z \leftarrow z^2 + c$ is iteratively computed for this random point c, for up to 25,000 iterations.

   (a) If the condition $|z|^2 > 4$ is met at any iteration, the point c is marked as being outside the Mandelbrot set.

   (b) Conversely, if all 25,000 iterations are completed without meeting this condition, the point c is considered to be inside the Mandelbrot set.

   Let N_I and N_O denote the total number of points that are inside and outside the Mandelbrot set, respectively.

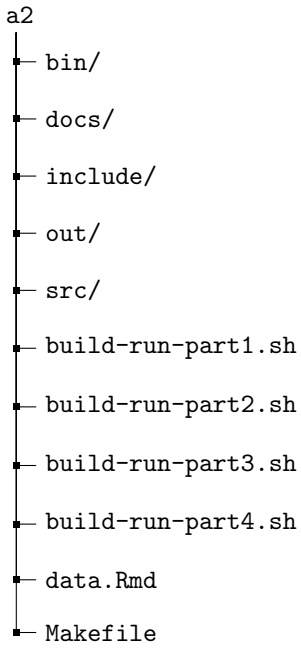
5. The area estimate for the Mandelbrot set is computed as:

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

   where $\text{Area}(\mathbf{R}) = 2.5 \times 1.25 = 3.125$.

2 Project Organization

The project is laid out to provide a clean separation of source files, header files, compiled objects, and documentation.



- **bin/**: The `bin` folder holds compiled objects and executable files, centralizing the output of the compilation process.
- **docs/**: This folder contains LaTeX files and other documentation materials that pertain to the report.
- **include/**: Here, all the header files (`.h`) are stored. These headers contain function prototypes and static inline functions used across different benchmarks.
- **out/**: The `out` folder stores the outputs from each task. Each subdirectory is of the relevant task. It also houses the csv files generated by the programs.
- **src/**: This directory houses the source files (`.c`) that make up the benchmarks.
- **Shell Scripts**: The shell scripts are at the root directory and are used to submit the job for the relevant task to slurm via `sbatch`.
- **data.Rmd**: This R Markdown file is used to generate the tables and plots from the output csv files for the report.
- **Makefile**: This Makefile has undergone extensive modifications to support the project hierarchy. It is engineered to handle the appropriate linking of shared libraries and integration of common code components, ensuring a seamless compilation process.

3 Code Explanation, Compilation, and Execution

This section outlines the steps required to build and execute the code. The provided Bash scripts automate the entire process, making it straightforward to compile and run the code. All the below steps assume you are in the root of the project directory.

3.1 Automated Building and Execution

- **Part 1: Serial Program** There are two implementation of the serial program (to be elaborated in Section 4). The command `sbatch build-run-part1.sh` will compile and run the files:
  - `mandseq.c`: standard, slightly optimized, serial implementation of the algorithm.
  - `mandseq-avx.c` more advanced implementation with AVX 512 instructions.

  The output of the experiments will be in `out/serial.csv`.
- **Part 2: OMP Loop Directives** There are multiple programs run in Part 2. The programs associated with part 2 are:
  - `mandomp.c`: extending `mandseq.c` to use OMP loop directives.
  - `mandomp-avx.c`: extending `manseq-avx.c` to use OMP loop directives.

- `mandomp-ts.c`: a copy of `mandomp.c` using `drand-ts.c` for thread safe random number generation (RNG).
- `mandomp-ts-avx.c` a copy of `mandomp-avx.c` using `drand-ts.c` for thread safe RNG.
- `mandomp-collapse.c` a modification of `mandomp-ts.c` to support the collapse clause.
- `mandomp-collapse-avx.` a modification of `mandomp-ts-avx.c` to support the collapse clause.

The script `build-run-part2.sh` validates the weird behavior of the non thread safe RNG, shows how the fix in `drand-ts.c` fixes the issue, and assesses the thread safe code against multiple thread counts and multiple schedules. To run all the experiments, execute `sbatch build-run-part2.sh`. The results are stored in `out/omp.csv`.

- **Part 3: OMP Tasks** In part 3, only the AVX version of the program is adopted to OMP Tasks. There are multiple programs run in Part 3. The programs associated with part 3 are:
  - `mandomp-tasks.c`: extending `mandseq-avx.c` to use OMP tasks. One thread creates tasks and one task is created per batch of cells.
  - `mandomp-tasks-columns.c`: extending `mandomp-tasks.c` to instead create one task per column. One thread creates tasks.
  - `mandomp-ts-columns-shared.c`: a copy of `mandomp-tasks-columns.c` but now all threads create tasks.

  The script `build-run-part3.sh` assesses the area and wall clock time of the above progrmas. To run all the experiments, execute `sbatch build-run-part3.sh`. The results are stored in `out/tasks.csv`.
- **Part 4: Parallel RNG** In part 4, `mandomp-ts-avx.c` is modified into `mandomp-ts-avx-parallel.c`. It makes use of the parallel random number generator in `drand-ts.c`. To run all the experiments, execute `sbatch build-run-part4.sh`. The results are stored in `out/omp-parallel.csv`.

3.2 Post-Build Objects and Executables

Upon successful compilation and linking, an `obj/` directory will be generated within the root directory. This directory will contain the compiled output files. Additionally, the executable files for running each part will be situated in the `bin/` directory.

3.3 Output Files From `sbatch`

The output files generated from running the code by submitting the relevant Bash script via `sbatch` will be stored in the relevant subdirectory of the `out` directory.

4 Serial Implementation

The serial implementation was first written in the standard way to achieve a runtime of approximately 49.2 seconds. After some searching, an optimized implementation using AVX 256 instructions was found.[1] This source was rewritten to the assignment at hand using AVX 512 instructions found in the intel intrinsics guide. The serial runtime was improved to approximately 17.7 seconds using AVX 512 instructions. The serial implementation and all subsequent implementations make use of the provided random number generator seeded at value 12345.

[1]Source: https://polarnick.com/blogs/other/cpu/gpu/sse/opencl/openmp/2016/10/01/mandelbrot-set-sse-opencl.html

4.1 Standard Implementation

4.1.1 Implementation

First, we iterate over the cells in $\mathbf{R}$ by a double for loop through the sets X and Y. Concretely, the below loops iterate over all $(x, y) \in T$. For each cell $C_{x,y}$, we generate a random coordinate inside the cell and perform the mandelbrot iteration on it. Since we are performing $N = 3125000$ iterations total, we can simply count the number of cells inside the mandelbrot set and then compute $N_O = N - N_I$. This gets rid of any conditional checks in the loop helping optimize. The resulting for loop is:

```
int N = 3125000, N_I = 0;
for (size_t n = 0; n < 2500; n++)
{
    double current_bottom_left_x = -2.0 + CELL_SIDE_LENGTH * n;
    double max_x = current_bottom_left_x + CELL_SIDE_LENGTH;
    for (size_t m = 0; m < 1250; m++)
    {
        double current_bottom_left_y = 0.0 + CELL_SIDE_LENGTH * m;
        double max_y = current_bottom_left_y + CELL_SIDE_LENGTH;
        // Get a random x and y inside of the cell
        double random_x = current_bottom_left_x + (max_x - current_bottom_left_x) * drand();
        double random_y = current_bottom_left_y + (max_y - current_bottom_left_y) * drand();
        // mandelbrot_iteration returns 0 if outside the set, else 1
        N_I += mandelbrot_iteration(random_x, random_y);
    }
}
int N_O = N - N_I;
```

The `mandelbrot_iteration` function checks if the complex number with real part `random_x` and imaginary part `random_y`. The implementation is found in `mandelbrot.h`. The function is static inlined to allow for portability between the multiple programs and is elaborated below:

```
static inline int mandelbrot_iteration(double c_re, double c_im, size_t max_iterations)
{
    double z_re = 0.0, z_im = 0.0; // start at 0
    double magnitude_squared = 0.0;

    for (size_t i = 0; i < max_iterations; i++)
    {
        // compute the real part of z in the update
        double temp_re = z_re * z_re - z_im * z_im + c_re;
        // compute the imaginary part of z in the update
        z_im = (z_re + z_re) * z_im + c_im;
        z_re = temp_re;
        magnitude_squared = z_re * z_re + z_im * z_im;
        // break if we diverged
        if (magnitude_squared > 4.0) { return 0; }
    }
    return 1;
}
```

The update rule was done more optimally than just doing $z^2 + c$ via the pseudocode found in Wikipedia. It is worth noting that complex numbers were manually handled via tracking two real numbers rather than using the `double complex` type from the `<complex.h>`. Using the complex library resulted in atrociously bad performance relative to using just doubles. This seems to be a a common issue.

4.1.2 Results

Running the standard implementation 3 times, the following results were obtained:

Table 1: Serial Wall Clock Time and Area - Standard Algorithm

num_threads	seed	program	wc_time	area
1	12345	serial	49.25816	1.506648
1	12345	serial	49.25670	1.506648
1	12345	serial	49.25700	1.506648

The wall clock time is roughly 49.2 seconds. Each execution yielded a stable area estimate of 1.506648, which is a minor deviation from the 1.506632 value cited in the assignment guidelines. In our algorithm, we adopted a column-wise selection strategy for iterating through the cells $C_{x,y}$. Specifically, we looped over all available y values for a given fixed x. The state of the random number generator evolves as we traverse the cells, occupying distinct positions in its sequence. This results in different sampling points within each cell for each traversal scheme and thus will result in different points being selected in each $C_{x,y}$, potentially yielding a different area estimate.

4.2 AVX 512 Implementation

The Intel® Xeon® Platinum 8268 processor supports AVX-512, which allows for simultaneous operations on eight double-precision floating-point numbers. The AVX-512 version of this algorithm is implemented using the Intel AVX-512 instruction set, as detailed in Intel's Intrinsics Guide[2]. This capability enables the concurrent sampling of points for eight $C_{x,y}$ values and checks whether they belong to the Mandelbrot set.

4.2.1 Implementation

We still iterate over the cells in $\mathbf{R}$ by a double loop through the sets X and Y, but now we process 8 y values at a time. Since 1250 is not a multiple of 8, the remaining values are processed in a cleanup loop that is the standard serial implementation:

```
for (size_t n = 0; n < 2500; n++)
{
    double current_bottom_left_x = -2.0 + CELL_SIDE_LENGTH * n;
    double max_x = current_bottom_left_x + CELL_SIDE_LENGTH;

    // compute 8 y-values simultaneously
    for (size_t m = 0; m < 1250 / 8 * 8; m += 8)
    {
        // need to generate the 16 random numbers for the 8 cells
        for (int i = 0; i < 8; ++i)
        {
            random_x[i] = drand(); // pre-allocated arrays for 8 doubles
            random_y[i] = drand();
        }
        // load the random numbers into a 512 bit register via an unaligned load
        __m512d random_numbers_x = _mm512_loadu_pd(random_x);
        __m512d random_numbers_y = _mm512_loadu_pd(random_y);

        // get the 8 bottom left and top left y values for this iteration of the loop
        __m512d bottom_left_y_values = _mm512_add_pd(
            _mm512_set1_pd(0.0),
            _mm512_add_pd(_mm512_set1_pd(m * CELL_SIDE_LENGTH), pxs_deltas512));
        __m512d top_left_y_values = _mm512_add_pd(
```

[2]Source: Intel Intrinsics Guide

```
            _mm512_set1_pd(CELL_SIDE_LENGTH),
            bottom_left_y_values);

        // compute the random coordinates for the 8 cells
        __m512d x_values = _mm512_fmadd_pd(
            random_numbers_x,
            _mm512_set1_pd(max_x - current_bottom_left_x),
            _mm512_set1_pd(current_bottom_left_x));
        __m512d y_values = _mm512_fmadd_pd(
            random_numbers_y,
            _mm512_sub_pd(top_left_y_values, bottom_left_y_values),
            bottom_left_y_values);

        // Assess the 8 c values concurrently
        __mmask8 diverged_indices = mandelbrot_iteration_avx(
            x_values,
            y_values,
            MAX_ITERATIONS);

        // the 1's in this mask are the iterations that did NOT diverge
        __mmask8 indices_in_set = ~diverged_indices;
        int count = _popcnt32((unsigned int)indices_in_set);
        number_of_cells_inside_mandelbrot_set += count;

        total_iterations += 8;
    }

    // cleanup by performing the standard serial implementation
    for (size_t m = 1250 / 8 * 8; m < 1250; m++)
    {
        // standard serial implementation
    }
}
```

The `mandelbrot_iteration_avx` assesses the 8 sampled points from the 8 $C_{x,y}$ values simultaneously as follows:

```
static inline __mmask8 mandelbrot_iteration_avx(__m512d x_values,
                                                __m512d y_values,
                                                size_t max_iterations)
{
    // These are the 8 z values
    __m512d z_re = _mm512_set1_pd(0.0);
    __m512d z_im = _mm512_set1_pd(0.0);

    // 8 bit mask. 1 means that the c value at that index has diverged.
    __mmask8 diverged_indices = 0;

    // Assess the 8 c values concurrently
    for (size_t iteration = 0; iteration < max_iterations; iteration++)
    {
        // componentwise: z_re = z_re * z_re + z_im * z_im + c_re
        __m512d xsn = _mm512_add_pd(_mm512_sub_pd(
                                    _mm512_mul_pd(z_re, z_re),
                                    _mm512_mul_pd(z_im, z_im)),
                                x_values);

        // componentwise: z_im = 2 * z_re * z_im + c_im
        __m512d ysn = _mm512_add_pd(_mm512_mul_pd(
```

```
                                    _mm512_add_pd(
                                        z_re,
                                        z_re),
                                    z_im),
                                y_values);

            // Update only those positions where diverged_indices is zero
            __m512d new_z_re = _mm512_mask_mov_pd(z_re, ~diverged_indices, xsn);
            __m512d new_z_im = _mm512_mask_mov_pd(z_im, ~diverged_indices, ysn);
            z_re = new_z_re;
            z_im = new_z_im;

            // compute the magnitude squared componentwise
            __m512d magnitude_squared = _mm512_add_pd(
                _mm512_mul_pd(z_re, z_re),
                _mm512_mul_pd(z_im, z_im));

            // Generate a mask for numbers that have diverged (magnitude squared > 4)
            __mmask8 maskDiverged = _mm512_cmp_pd_mask(magnitude_squared,
                                                _mm512_set1_pd(4.0),
                                                _CMP_GT_OS);

            // Update diverged_indices using bitwise OR operation
            diverged_indices |= maskDiverged;

            // Break if all indices have diverged
            if (diverged_indices == 0xFF)
            {
                return diverged_indices;
            }
        }

        return diverged_indices;
    }
```

A shortcoming of this algorithm is that it requires all indices to diverge in order to break. So there is a worst case scenario where 7 cells near immediately diverge but one cell completes all the iterations. A more advanced implementation might find a way around this issue.

4.2.2 Results

Running the AVX 512 implementation 3 times, the following results were obtained:

Table 2: Serial Wall Clock Time and Area - AVX Algorithm

num_threads	seed	program	wc_time	area
1	12345	serial-avx	17.75959	1.506656
1	12345	serial-avx	17.75925	1.506656
1	12345	serial-avx	17.75885	1.506656

The wall clock time for the algorithm leveraging AVX-512 was measured at approximately 17.8 seconds. This represents a speedup of approximately 2.76 times compared to the standard implementation.[3] Each

[3]This performance gain is suboptimal compared to the theoretical 8 times speedup. Several factors contribute to this disparity. Memory bandwidth limitations can cause delays in loading data into SIMD registers, reducing the effective speedup. The inherent latencies and throughputs of AVX-512 instructions, as well as the overheads associated with loading and storing data from and to SIMD registers, can also affect performance. Moreover, non-vectorizable portions of the code can further dilute the anticipated gains.

run of the algorithm produced an area estimate of 1.506656. This value differs from both the standard serial implementation's estimate and the value suggested by the assignment guide. While the same random numbers as the standard implementation were used in each iteration—ensured by synchronizing the state of the random number generator—the hypothesis for the discrepancy in the area estimate is that AVX-512 intrinsics operate with a distinct floating-point precision profile. This different precision profile could alter the computational outcome, leading to the observed variation in the area estimate.

5 OpenMP Loop Directives

Next, the serial implementation was modified to use loop directives to create parallel, multithreaded versions of the serial implementation. To handle this, the double for loop is wrapped in the `omp parallel` directive; explicitly instructing the compiler to parallelize the chosen block of code. The first for loop is then tagged with the `omp for` directive; instructing the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct. The environment variables `OMP_NUM_THREADS` and `OMP_SCHEDULE` are used to control the core count (one thread per core) and scheduling, respectively.

5.1 Parallel threads using `omp for` - No Scheduling

Each thread has private variables `total_iterations_th` and `number_of_cells_in_mandelbrot_set_th` which tally the total iterations that thread performed and the number of points in the mandelbrot set that thread determined, respectively. Additionally, each thread instantiates its own seed. At the end, each threads private variables are added atomically to the shared variables:

- `number_of_cells_inside_mandelbrot_set`,
- `total_iterations`.

For the AVX version, each thread also needs two 8 length double arrays for the random numbers. Explicitly, the AVX algorithm is something like:[4]

```
#pragma omp parallel shared(number_of_cells_inside_mandelbrot_set, total_iterations)
    private(number_of_cells_inside_mandelbrot_set_th, total_iterations_th, random_x, random_y)
    default(none)
{
    dsrand(seed); // each thread initializes a seed
    random_x = (double *)malloc(8 * sizeof(double));
    random_y = (double *)malloc(8 * sizeof(double));
    number_of_cells_inside_mandelbrot_set_th = 0;
    total_iterations_th = 0;

    #pragma omp for
    for (size_t m = 0; m < 2500; ++m)
    {
        // AVX code from prior
        for (size_t n = 0; n < 1250 / 8 * 8; n += 8) { ... }
        // standard code from prior
        for (size_t n = 1250 / 8 * 8; n < 1250 ; ++n) { ... }
    }
    // add each threads results to the desired totals
    #pragma omp atomic
    number_of_cells_inside_mandelbrot_set += number_of_cells_inside_mandelbrot_set_th;

    #pragma omp atomic
    total_iterations += total_iterations_th;
```

[4]The non AVX version is near identical. The only difference is each thread does not require an array to store the random numbers in since they can be computed on the fly and are not loaded into a register.

```
    free(random_x);
    free(random_y);
}
```

The `omp atomic` directive ensures the addition of each threads results to the shared variables is done atomically.[5]

5.1.1 Lack of Thread Safety

The above algorithm is run using the base `drand` implementation with the environment variables:

- `OMP_NUM_THREADS` set to 2.
- `OMP_SCHEDULE` not set.

This runs the algorithm with two cores (one thread per core) and no scheduling. The results are:

Table 3: OMP Wall Clock Time and Area - drand

num_threads	seed	program	wc_time	area
2	12345	standard	42.09501	1.506796
2	12345	standard	42.09643	1.506670
2	12345	standard	42.08855	1.506686
2	12345	avx	15.16014	1.506838
2	12345	avx	15.17255	1.506796
2	12345	avx	15.16531	1.506812

Table 3 shows both a performance improvement over the serial implementation and the algorithm computing a different area estimate each run. The performance improvement for both the standard and AVX serial algorithm using two threads is approximately 1.2× faster.

5.1.2 Thread Safe Random Number Generation

The differing area estimates are due to the lack of thread safe random number generation. In particular, multiple threads may be modifying the same seed value causing unstable random number generation. This is fixed by modifying the static seed variable in `drand.c` to be `threadprivate`. In particular, the `omp threadprivate` directive makes the static block-scope variable private to each thread. This prevents multiple threads from updating the same seed value and ensures each thread computes its own random numbers. The implementation is a one line change to `drand.c` and is done in `drand-ts.c` to make running the experiments easier. Explicitly:

```
static uint64_t seed_ts;

// ensures each thread is modifying it's own seed
#pragma omp threadprivate(seed_ts)

void dsrand_ts(unsigned s)
{
    seed_ts = s - 1;
    printf("Seed_ts = %lu. RAND_MAX = %d.\n", seed_ts, RAND_MAX);
}

double drand_ts(void)
```

[5] In particular, it ensures that race conditions are avoided through direct control of concurrent threads that might read or write to or from the particular memory location. It allows for more efficient concurrent algorithms due to the use of fewer locks than `omp critical`. https://www.ibm.com/docs/sk/xl-c-aix/13.1.0?topic=processing-pragma-omp-atomic

```
{
    seed_ts = 6364136223846793005ULL * seed_ts + 1;
    return ((double)(seed_ts >> 33) / (double)RAND_MAX);
}
```

A problem with this fix in particular is each thread will use the same sequence of random numbers. This is remedied in section 7. Using `dsrand_ts` and `drand_ts` instead of `dsrand` and `drand` in the 2 thread no schedule experiment yields:

Table 4: OMP Wall Clock Time and Area - drand-ts

num_threads	seed	program	wc_time	area
2	12345	omp-ts	41.87915	1.506622
2	12345	omp-ts	41.86007	1.506622
2	12345	omp-ts	41.85505	1.506622
2	12345	omp-avx-ts	15.01974	1.506716
2	12345	omp-avx-ts	15.02008	1.506716
2	12345	omp-avx-ts	15.01922	1.506716

We observe the same wall clock performance, but this fix now allows for reproducible area values for a fixed number of threads. Moving forward, we run all experiments with the thread safe random number generator.

5.1.3 Loop Directives Wall Clock Time and Area Across Thread Counts - Using `drand-ts`

The algorithm is now run with `OMP_NUM_THREADS` set to 1, 2, 4, 12, and 24 cores (one thread per core) with no scheduling. The results are:

Table 5: OMP Wall Clock Time and Area Across Threads - No Scheduling

program	num_threads	wc_time	area
omp-avx-ts	1	17.631131	1.506656
omp-avx-ts	2	15.021128	1.506716
omp-avx-ts	4	8.000733	1.506778
omp-avx-ts	12	3.447970	1.506652
omp-avx-ts	24	1.937948	1.506664
serial	1	49.25670	1.506648
omp-ts	1	49.006569	1.506648
omp-ts	2	41.859844	1.506622
omp-ts	4	22.352743	1.506532
omp-ts	12	9.674459	1.506682
omp-ts	24	5.409102	1.506800

We observe that increasing the thread count results in faster execution of both the standard and AVX implementation. The performance improvement factor is the same across both algorithms. The serial version has the same area estimate as the parallel version with one thread, which makes sense as it is precisely the same algorithm. As the number of thread counts increases, our random number generator is instantiated on more threads. The algorithm is also processing a different batch of cells for each thread and thus the area estimate is bound to change as a different point is chosen in each cell for varying thread counts.

5.2 Parallel threads using `omp for` - With Scheduling

We add the `schedule(runtime)` clause to the `for` directive to allow for setting the scheduling of iterations to threads. Now, for each thread count `OMP_NUM_THREADS` set to 2, 4, 12, and 24 cores, we assess `OMP_SCHEDULE` set to: `schedule(static, 1)`, `schedule(static, 100)`, `schedule(dynamic)`, `schedule(dynamic, 250)`, and `schedule(guided)`. Respectively, these options perform:

- **schedule(static, 1)**:
  - Method: Each thread receives one iteration at a time until all iterations are exhausted. Iterations are allocated in a round-robin fashion.
  - Pros & Cons: Minimal overhead but poor load balance if iterations have varying computational costs.
- **schedule(static, 100)**:
  - Method: Iterations are divided into chunks of 100 and statically assigned to threads.
  - Pros & Cons: Low overhead and since the number of iterations is divisible by 100 there are no threads left idle.
- **schedule(dynamic)**:
  - Method: Iterations are dynamically assigned to threads as they become available.
  - Pros & Cons: High overhead but excellent load balancing.
- **schedule(dynamic, 250)**:
  - Method: Each thread dynamically grabs chunks of 250 iterations from the loop.
  - Pros & Cons: Once the 2500 iterations to schedule have been assigned we expect no performance gain above about 10 threads.
- **schedule(guided)**:
  - Method: Dynamic assignment of decreasingly sized blocks of iterations to each thread.
  - Pros & Cons: Aims to balance overhead and load by initially assigning large chunks and reducing the size dynamically.

The average wall clock time and area for the AVX algorithm for each thread and schedule were:

Table 6: OMP Wall Clock Time and Area Across Threads - All Schedules AVX

program	schedule	num_threads	wc_time	area
omp-avx-ts	dynamic	2	8.857233	1.506857
omp-avx-ts	dynamic	4	4.426543	1.506663
omp-avx-ts	dynamic	12	1.478229	1.506761
omp-avx-ts	dynamic	24	0.741666	1.506795
omp-avx-ts	dynamic,250	2	9.398375	1.506794
omp-avx-ts	dynamic,250	4	5.694130	1.506746
omp-avx-ts	dynamic,250	12	4.477652	1.506758
omp-avx-ts	dynamic,250	24	4.479328	1.506758
omp-avx-ts	guided	2	8.857341	1.506644
omp-avx-ts	guided	4	4.500216	1.506787
omp-avx-ts	guided	12	1.484397	1.506694
omp-avx-ts	guided	24	0.745663	1.506713
omp-avx-ts	static,1	2	8.851426	1.506732
omp-avx-ts	static,1	4	4.429402	1.506760
omp-avx-ts	static,1	12	1.482386	1.506800
omp-avx-ts	static,1	24	0.743956	1.506802
omp-avx-ts	static,100	2	8.854995	1.506648
omp-avx-ts	static,100	4	5.102193	1.506622
omp-avx-ts	static,100	12	2.041134	1.506880
omp-avx-ts	static,100	24	1.939225	1.506736

The area estimate differs amongst different schedules for the same thread count. This indicates that the cells are being traversed in a different order for the different schedules and thus a different point is likely sampled within each cell (which on the boundary matters a lot for determining if we are inside or outside the set). For the wall clock time, we notice:

- `dynamic`: There is a consistent decrease in wall clock time as the number of threads increases. We are lead to believe that the load is being well-distributed for a chunk size of 1 and that all threads are working efficiently.
- `dynamic, 250` The wall clock time decreases until 12 threads and then plateaus. A too-large chunk size is leading to less efficient use of threads as thread count increases. Indeed, we expect 10 threads to grab the 2500 iterations to schedule over and the remaining threads above 10 are left completely idle. This is precisely what is observed.
- `guided`: Very similar in performance to `dynamic`. This makes sense since the chunk size decreases over time and the threads seem to work more efficiently in this manner.
- `static,1`: This has the same time decrease pattern as `dynamic`. The chunk size being set to 1 appears to allow for a good work distribution despite the potential overhead.
- `static, 100`: The time decrease here is less smooth than that of `static, 1`. This indicates some threads are either being left idle or inefficiently used. There is little performance gain between 12 and 24 threads.

It is worth noting that wall clock time is significantly impacted by even one slow thread. Since the density of cells inside the Mandelbrot set is non constant as we iterate through the columns, threads that have to handle the center are bound to take more time. Thus, the `dynamic, guided,` and `static,1` options handle work distribution better than when we assign threads a large chunk of work as the threads in charge of the

center will undoubtedly take more time. The next table shows the results for the standard algorithm, where the exact same conclusion is drawn.

Table 7: OMP Wall Clock Time and Area Across Threads - All Schedules Standard

program	schedule	num_threads	wc_time	area
omp-ts	dynamic	2	24.546266	1.506733
omp-ts	dynamic	4	12.279111	1.506719
omp-ts	dynamic	12	4.093797	1.506722
omp-ts	dynamic	24	2.049885	1.506824
omp-ts	dynamic,250	2	25.915077	1.506752
omp-ts	dynamic,250	4	15.846655	1.506792
omp-ts	dynamic,250	12	12.539101	1.506682
omp-ts	dynamic,250	24	12.539795	1.506682
omp-ts	guided	2	24.598179	1.506790
omp-ts	guided	4	12.471653	1.506759
omp-ts	guided	12	4.132643	1.506768
omp-ts	guided	24	2.055091	1.506711
omp-ts	static,1	2	24.550046	1.506662
omp-ts	static,1	4	12.277485	1.506794
omp-ts	static,1	12	4.103413	1.506824
omp-ts	static,1	24	2.055128	1.506882
omp-ts	static,100	2	24.671857	1.506674
omp-ts	static,100	4	14.196153	1.506664
omp-ts	static,100	12	5.696408	1.506728
omp-ts	static,100	24	5.445335	1.506714

We conclude that certain types of scheduling improve the load balance amongst threads for loop directives for both the standard and AVX algorithm.

5.3 Parallel threads using `omp for` - With Collapsing & Scheduling

Next, the loop uses the collapse clause; increasing the number of loop chunks that the runtime system can schedule across the threads.. The source code was slightly modified to allow for collapsing. In particular, no code can be inbetween the two loops iterating through X and Y and thus the code for computing the X values is moved to the inner most loop. In the AVX algorithm, since there is a cleanup loop, we have to collapse twice - once for the AVX code and once for the standard cleanup. The AVX algorithm adjustments are shown to emphasize the changes:

```
#pragma omp parallel // insert same shared, private
{ // insert same setup
    #pragma omp for collapse(2) schedule(runtime)
    for (size_t n = 0; n < NUM_X_ITERATIONS; n++)
    {
        for (size_t m = 0; m < NUM_Y_PS; m += 8)
        {
            // to support collapse have to compute the current bottom left x for all y values now
            // insert same AVX sequential code
        }
    }

    // to support collapsing the cleanup iterations must be their own collapse
    #pragma omp for collapse(2) schedule(runtime)
    for (size_t i = 0; i < NUM_X_ITERATIONS; ++i)
    {
        for (size_t j = NUM_Y_PS; j < NUM_Y_ITERATIONS; ++j)
        {
            // have to compute x for all y values to support collapsing
            // insert same standard sequential code
        }
    } // insert same cleanup
}
```

The results were measured for `OMP_NUM_THREADS` set to 2, 4, 12, and 24 cores and `OMP_SCHEDULE` set to the same options as prior. The next table shows the results from collapsing across scheduling options for the AVX algorithm

Table 8: OMP Wall Clock Time and Area Across Threads - All Schedules With Collapse AVX

program	schedule	num_threads	wc_time	area
omp-collapse-avx	dynamic	2	8.924948	1.506725
omp-collapse-avx	dynamic	4	4.463100	1.506759
omp-collapse-avx	dynamic	12	1.489609	1.506655
omp-collapse-avx	dynamic	24	0.748752	1.506757
omp-collapse-avx	dynamic,250	2	8.917002	1.506840
omp-collapse-avx	dynamic,250	4	4.459263	1.506713
omp-collapse-avx	dynamic,250	12	1.487800	1.506748
omp-collapse-avx	dynamic,250	24	0.747705	1.506653
omp-collapse-avx	guided	2	8.948330	1.506829
omp-collapse-avx	guided	4	4.544421	1.506701
omp-collapse-avx	guided	12	1.504375	1.506810
omp-collapse-avx	guided	24	0.751079	1.506727
omp-collapse-avx	static,1	2	8.982337	1.506750
omp-collapse-avx	static,1	4	4.554585	1.506808
omp-collapse-avx	static,1	12	1.599819	1.506694
omp-collapse-avx	static,1	24	0.803146	1.506684
omp-collapse-avx	static,100	2	8.925599	1.506888
omp-collapse-avx	static,100	4	4.460062	1.506850
omp-collapse-avx	static,100	12	1.494543	1.506714
omp-collapse-avx	static,100	24	0.750147	1.506734

The collapse clause offers no noticeably enhanced improvements to `static, 1`, `dynamic`, and `guided`. However, it significantly improves the prior ill performing `dynamic, 250` and `static, 100`. This makes sense for the source code. In the non-collapsed code, there are 2500 iterations to schedule over and the `dynamic, 250` cannot make use of more than 10 threads. However, in the collapsed code, there are 2500*1250 iterations to schedule over in the standard algorithm (2500 * 158 in the AVX version) and thus each thread can be kept busy. In a similar fashion, the load is better distributed for chunk sizes of 100 across the 2500*1250 schedulable iterations for `static, 100`, which no longer has any one thread handle an entire chunk of the center.

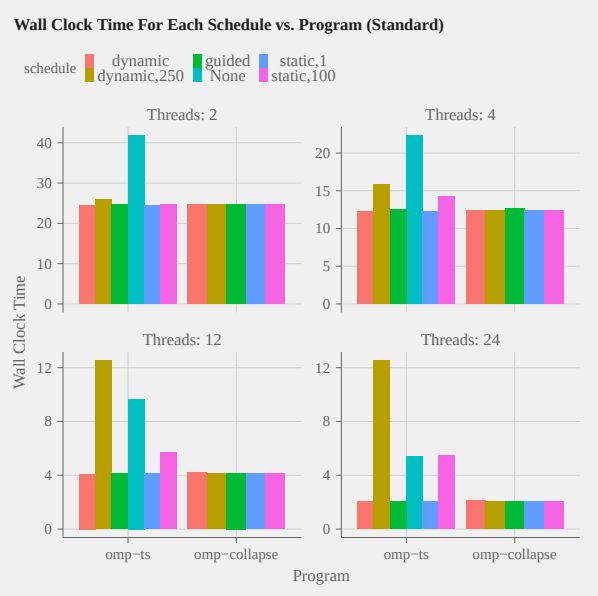
The performance being the same for the other scheduling options is likely because the non-collapse code, which has 2500 iterations to schedule over, gives a sufficient number of schedulable chunks to achieve good load balance. In the collapsed version, there are more chunks to schedule and it appears the overhead of scheduling these chunks outweighs any performance gain. Additionally, we no longer make use of caching the x values for each stride of y. Below is the table of values for the standard algorithm, where the exact same conclusions are drawn.

Table 9: OMP Wall Clock Time and Area Across Threads - All Schedules With Collapse Standard

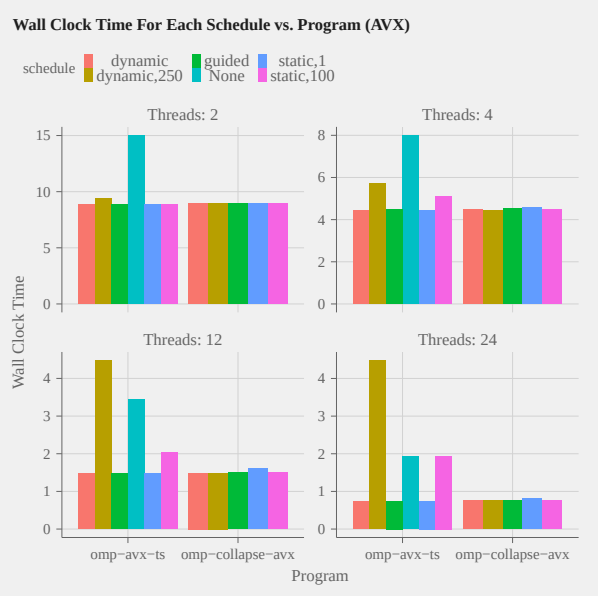
program	schedule	num_threads	wc_time	area
omp-collapse	dynamic	2	24.793561	1.506799
omp-collapse	dynamic	4	12.412081	1.506779
omp-collapse	dynamic	12	4.180311	1.506715
omp-collapse	dynamic	24	2.123831	1.506677
omp-collapse	dynamic,250	2	24.672736	1.506713
omp-collapse	dynamic,250	4	12.336788	1.506789
omp-collapse	dynamic,250	12	4.111878	1.506702
omp-collapse	dynamic,250	24	2.058562	1.506671
omp-collapse	guided	2	24.747149	1.506708
omp-collapse	guided	4	12.602658	1.506728
omp-collapse	guided	12	4.168920	1.506780
omp-collapse	guided	24	2.067399	1.506884
omp-collapse	static,1	2	24.743476	1.506694
omp-collapse	static,1	4	12.381558	1.506850
omp-collapse	static,1	12	4.130896	1.506624
omp-collapse	static,1	24	2.068478	1.506822
omp-collapse	static,100	2	24.667321	1.506718
omp-collapse	static,100	4	12.334259	1.506728
omp-collapse	static,100	12	4.119450	1.506808
omp-collapse	static,100	24	2.063424	1.506754

5.4 Comparing It All

Looking at tables of value can make it difficult to compare results. We assess the performance of collapsing vs. not collapsing across thread counts across schedules graphically to observe the lack of performance gains that scheduling and collapsing provide the algorithm in the following figure:



(a) Standard Algorithm Performance For Each Schedule Across Thread Counts



(b) AVX Algorithm Performance For Each Schedule Across Thread Counts

It is visibly observable that the wall clock change between collapsing and non collapsing is similar for the schedules `static, 1`, `dyamic`, and `guided` while the wall clock time for `static, 100` and `dynamic, 250` is signicantly improved as thread count increases for the collapsed program.

6 OpenMP Tasks

The next task takes the serial implementation and uses OpenMP tasks to parallelize it. Only the AVX version is operated on in this step to compare to the loop directives performance. First, a program is created in which each cell or batch of cells constitutes a task and one thread is dedicated to creating all the tasks. Then, a program is created that treats each column of cells as a task and one thread is dedicated to creating all the tasks. Finally, the program is modified to so that each column of cells is treated as a task but all threads are able to create tasks.

6.1 One Task Per Cell Batch

Since the AVX algorithm operates on 8 cells at once, each batch of 8 cells is its own task. The cleanup loop, which operates on one cell at once, has each individual cell created as a task. To implement this, we:

- Take the `mandseq-avx.c` and wrap the double for loop in the parallel directive - with almost same sharing and private rules as `mandomp-avx.c`.
- Wrap the double for loop in the omp single directive - specifying that the for loop must run a single thread.
- Wrap the `mandelbrot_iteration_avx` and `mandelbrot_iteration` functions in the omp task directive.
- Each task updates the global counters `total_iterations` and `number_of_cells_inside_mandelbrot_set` atomically instead of each thread having its own counter.

In total, $2500 * (1248/8 + (1250 - 1248)) = 395000$ tasks are created. The average area and average time across core counts 1, 2, 4, 12, and 24 was:

Table 10: OMP Tasks Clock Time and Area - Task Per Cell Batch

program	num_threads	wc_time	area
per-cell-task	1	17.771752	1.506656
per-cell-task	2	8.950503	1.506656
per-cell-task	4	4.534696	1.506656
per-cell-task	12	1.657044	1.506656
per-cell-task	24	0.963572	1.506656

It is remarkable that for core count larger than 1 the task algorithm is twice as fast as any scheduled loop directive (same performance for 1 core). There is a possibility that the runtime system has more flexibility in scheduling tasks than in scheduling loop chunks, allowing it to do a better job of balancing the load. Another possibility for the speedup is now we are parallelizing just the the mandelbrot iteration function, which may have a faster longest thread runtime than the entire loop directive version. The average area computation is consistent across all thread counts and equal to that of the sequential algorithm - which was not the case with any of the loop directive AVX algorithms. This is because tasks are being processed with the exact same random number sequence as in the sequential version.

6.2 One Task Per Column - Single Thread Task Generation

The program is modified so that for each x value, one task is created (2500 tasks total). This is a simple change and the inner for loops on y are wrapped in the task directive. The average area and average time across core counts 1, 2, 4, 12, and 24 was:

Table 11: OMP Tasks Clock Time and Area - Task Per Column

program	num_threads	wc_time	area
per-column-task	1	17.584878	1.506656
per-column-task	2	8.812490	1.506645
per-column-task	4	4.432580	1.506692
per-column-task	12	1.510451	1.506747
per-column-task	24	0.777615	1.506649

For 2, 4, and 12 cores there is a minor performance gain. For 24 cores, the per column task is about 1.2x faster than the per cell task. It is still near double the speed of the omp for loop directives version. It is interesting to note that the area estimate now differs again as the core count varies. The reasoning is the random number generator is not on the same state for all $C_{x,y}$ and thus different points are assessed for inclusion.

6.3 One Task Per Column - Multiple Threads Generating Tasks

The program is modified so that all threads generate tasks. This is done by keeping the omp task directive as is, removing the omp single directive wrapping the for loops, and placing an omp parallel directive on the outermost for loop. Placing the omp for directive is necessary so that task creation is split evenly amongst the threads. Without it, every thread tries to create the 2500 tasks leading to a lot of errors. The average area and average time across core counts 1, 2, 4, 12, and 24 was:

Table 12: OMP Tasks Clock Time and Area - Task Per Column Multithreaded Task Creation

program	num_threads	wc_time	area
per-column-task-shared	1	17.655240	1.506656
per-column-task-shared	2	8.841601	1.506771
per-column-task-shared	4	4.422934	1.506579
per-column-task-shared	12	1.486727	1.506735
per-column-task-shared	24	0.751839	1.506671

For 2, 4, 12, and 24 cores there is a minor performance gain over the prior program. We are still at near 2x speedup compared to the loop directive version. It appears the program is capable of load balancing the task creation and execution well even when threads are also busy creating tasks. The area estimates differ from when a single thread was creating all tasks. This is again due to the random number generator not being in the same state for all $C_{x,y}$ and thus different points are assessed for inclusion.

7 Parallel Random Number Generation

A shortcoming of the random number generator used throughout this project is that multiple threads may wind up using the same sequence of random numbers. This is due to the fact that each thread initiates itself at the same seed starting point and thus each thread could wind up using the exact same sequence of random numbers. One simple way to remedy this is via to give each thread its own starting point as a function of the seed. In this way, results are reproducible for the same seed but each thread will wind up using a different sequence of random numbers. Using Ian Foster's documented Leapfrog Method for creating a distributed number generator, [6] we implement the Leapfrog Method:

1. Each thread needs to be initialized with a seed that is a function of the master seed and the thread's ID.
2. The thread updates its seed by advancing the seed by the number of threads. E.g.,
   - Thread 0 uses numbers at position 0, `NUM_THREADS`, 2*`NUM_THREADS`,...
   - Thread 1 uses numbers at position 1, `NUM_THREADS` + 1, 2*`NUM_THREADS` + 1,...
   - Thread i uses numbers at position i, `NUM_THREADS` + i, 2*`NUM_THREADS` + i,...

The following changes are made to `drand-ts.c`:

[6] Source: https://www.mcs.anl.gov/ itf/dbpp/text/node116.html

```
static uint64_t seed_ts;
// ensures each thread is modifying it's own seed
#pragma omp threadprivate(seed_ts)

// Advance the RNG state by `n` steps
void leapfrog(uint64_t *seed, uint64_t n)
{
    for (uint64_t i = 0; i < n; ++i)
    {
        *seed = 6364136223846793005ULL * (*seed) + 1;
    }
}

void dsrand_parallel_ts(unsigned s)
{
    int thread_id = omp_get_thread_num();

    uint64_t master_seed = s - 1;
    seed_ts = master_seed;

    // Each thread leapfrogs `thread_id` steps from master_seed
    leapfrog(&seed_ts, thread_id);
}

double drand_parallel_ts()
{
    // leapfrog the seed ahead num_threads
    int num_threads = omp_get_num_threads();
    leapfrog(&seed_ts, num_threads);

    return ((double)(seed_ts >> 33) / (double)RAND_MAX);
}
```

The program `mandomp-ts-avx.c` is copied into `mandomp-ts-avx-parallel.c` and each thread invokes the function `dsrand_parallel_ts` to set its seed to the appropriate starting value. The function `drand_parallel_ts` gets the next random number for the thread by leaprogging the seed for that thread by the number of threads. The program is rerun with 24 cores with the best performing result from part 2. Since the results in part 2 were all so close, the absolute minimum wall clock time was chosen. This was when there was 24 cores and the schedule was `dynamic`. The average area and wall clock time are:

Table 13: Investigating OMP Loop Directives with Parallel RNG

program	schedule	num_threads	wc_time	area
omp-avx-ts	dynamic	24	0.741666	1.506795
omp-avx-parallel-rng	dynamic	24	0.753090	1.506693

The parallel rng does make a difference on the area estimate. I would have more confidence using the leapfrog method throughout to ensure the randomness in each thread is independent than that of other threads. Since the leapfrog method allows for reproducible results, there is no reason not to use a more parallel safe random number generator.