CPSC 524 Assignment 2: Area of the Mandelbrot Set

Rami Pellumbi[*]

October 1, 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 **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 **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 $|z| > 2$ 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. Finally, 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 adopts a well-structured folder hierarchy to ensure clean separation of various components such as source files, header files, compiled objects, and documentation.

```
a2
├─ bin/
├─ docs/
├─ include/
│  ├─ complex.h
│  ├─ drand.h
│  ├─ mandelbrot.h
│  ├─ timing.h
│  └─ utilities.h
├─ out/
├─ src/
│  ├─ drand.c
│  ├─ mandomp-ts.c
│  ├─ mandomp.c
│  ├─ mondseq.c
│  ├─ timing.c
│  └─ utilities.c
├─ build-run-mandseq.sh
├─ build-run-mandomp.sh
├─ build-run-mandomp-ts.sh
└─ Makefile
```

- **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 acts as a repository for all output files generated during program execution. It employs an organized sub-folder structure based on the executed scripts, making debugging and performance analysis more efficient.
- **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`.
- **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 Building and Running the Code

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**

  1. Run the following command to submit the Part 1 build and execution script:
     `sbatch build-run-mandseq.sh`

  This script used the modified `Makefile` to handle the compilation and linking of Part 1 specific code.

- **Part 2 - 1a: Naive `omp for`**

- **Part 2 - 1b: `omp for` with thread safe number generation**

- **Part 2 - 2: Assessing scheduling options**
- **Part 2 - 3: Collapse**

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 heavily optimized to achieve a runtime of approximately 4.8 seconds. First, we iterate over the cells in **R** by a double for loop through the sets X and Y. Concretely, the below loops iterate over all $(x, y) \in T$:

```
for (size_t n = 0; n < 2500; n++)
{
    double current_bottom_left_x = -2.0 + CELL_SIDE_LENGTH * n;
    for (size_t m = 0; m < 1250; m++)
    {
        double current_bottom_left_y = 0.0 + CELL_SIDE_LENGTH * m;
    }
}
```

Next, for each cell, 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. Adjusting our for-loop above:

```
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 = get_random_double_in_bounds(current_bottom_left_x, max_x);
        double random_y = get_random_double_in_bounds(current_bottom_left_y, max_y);
        // 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 function `get_random_double_in_bounds` is the standard way to generate a random number inbetween a minimum and maximum:

```
static inline double get_random_double_in_bounds(double min, double max)
{
    return min + drand() * (max - min);
}
```

The real optimization magic happens in the function `mandelbrot_iteration`. Instead of doing one mandelbrot iteration and magnitude check at a time, we compute 10 mandelbrot updates per magnitude check. This reduces the amount of time we perform an if statement check at the small cost a few extra floating point operations if we have already diverged. The complex arithmetic is custom implemented in `complex.h` and is the standard complex addition and multiplication.

```
static inline int mandelbrot_iteration(double c_re, double c_im)
{
    // Initialize z to 0. z is a complex number represented by z_re and z_im.
    // The Mandelbrot sequence is generated based on the iterative formula: z = z^2 + c
    double z_re = 0.0, z_im = 0.0;

    // Temporary variables to hold intermediate results of complex operations.
    double temp_re, temp_im;

    // Maximum number of iterations to decide whether the number is in the Mandelbrot set or
       not.
    // If the magnitude of z goes above 2 and stays there, it's not in the set.
    size_t MAX_ITERATIONS = 25000;

    // The count of iterations to unroll in the inner loop for performance optimization.
    size_t UNROLL_COUNT = 10;

    // Outer loop: Process batches of UNROLL_COUNT iterations at a time
    for (size_t i = 0; i < MAX_ITERATIONS / UNROLL_COUNT; i += 1)
    {
        // This is a loop unrolling optimization to minimize the loop overhead.
        for (size_t i = 0; i < UNROLL_COUNT; ++i)
        {
            // Calculate z^2. Store result in temporary variables temp_re and temp_im
            complex_multiply(z_re, z_im, z_re, z_im, &temp_re, &temp_im);

            // Update z by adding c to z^2. Store result in z_re and z_im
            complex_add(temp_re, temp_im, c_re, c_im, &z_re, &z_im);
        }

        // Check if the magnitude squared of the current z value squared exceeds 4.
        if (complex_magnitude_squared(z_re, z_im) > 4.0)
        {
            // If it does, the number does not belong to the Mandelbrot set and we return 0.
            return 0;
        }
    }

    // If we've reached this point, the number is assumed to be in the Mandelbrot set.
    return 1;
}
```

5 OpenMP Loop Directives