Julia Sets

Wilfrid Laurier Univeristy

Due Tuesday February 13, 2024

Brandon Parker - 191593730
Torin Borton-McCallum - 190824620
Grant Westerholm - 190462000
Rhea Sharma - 200576620
Taha Amir - 190728860
William Mabia - 190133240
Riley Adams - 190416070

Contents

1 Introduction **2**

2 Algorithm Overview **2**
2.1 Design Approach 2
2.2 Challenges Faced 2

3 Algorithm Results **3**

4 Benchmarking **3**
4.1 Performance Analysis 4
4.1.1 Mandelbrot Generation Graph (Image Size: 10000x10000) 4
4.1.2 Julia Set Generation Graph (Image Size: 10000x10000) 4
4.1.3 Performance Conclusion 4
4.2 MPI Wtime Resolution 5

5 Interactive Visualization of High-Resolution Fractal Images **5**
5.1 Overview 5
5.2 Functionality 5
5.3 Purpose 5

6 Conclusion **5**

7 Appendices **6**
7.1 Code Implementation 6
7.1.1 Mandelbrot 6
7.1.2 Julia Sets 15

1 Introduction

Fractals, characterized by their self-similar patterns at different scales, have captivated mathematicians, scientists, and artists for centuries. Among the large amounts of fractals discovered, the Julia and Mandelbrot sets stand out for their aesthetic appeal and mathematical significance. The Julia set is defined by a particular complex quadratic polynomial. It exhibits intricate patterns when plotted in the complex plane, revealing an infinite variety of shapes and structures. The Mandelbrot set is generated by iterating complex numbers through a simple mathematical formula.

In this paper, we present an approach to generate Julia and Mandelbrot sets in parallel using the Message Passing Interface (MPI) library. By harnessing the power of parallel processing, we aim to accelerate the computation and visualization of these fractals.

2 Algorithm Overview

To parallelize the generation of Julia and Mandelbrot sets, we utilize the Message Passing Interface (MPI) library. Our approach involves dividing the computation of Julia and Mandelbrot sets into independent tasks, each assigned to a separate MPI process. By distributing the workload among multiple processes, we can exploit the parallelism inherent in modern computing architectures, thereby reducing the overall computation time.

2.1 Design Approach

1. **Distributed Computation:** Our approach involves dividing the complex plane into different rows, with each section assigned to a separate MPI process. This distributed computation strategy ensures efficient parallelization of the fractal generation task, enabling simultaneous processing of multiple regions of the complex plane.

2. **Iteration and Testing:** For each point in the assigned section, we iteratively apply the complex function (for Julia set) or quadratic polynomial (for Mandelbrot set) and test for convergence or divergence. This iterative process determines whether each point belongs to the fractal set or not, based on a predefined threshold of maximum iterations.

3. **Coloring and Visualization:** Upon determining the status of each point (inside or outside the fractal set), we map the iteration count or divergence test results to colors. This mapping allows us to generate a visual representation of the fractal, where distinct colors denote different levels of divergence or iteration counts.

4. **Image Stitching:** Finally, we gather the computed results from all MPI processes and stitch them together to form the final image of the fractal. This consolidation step integrates the individual contributions of each process, resulting in a cohesive visual representation of the entire fractal.

2.2 Challenges Faced

Throughout the development of the Julia and Mandelbrot set generation program, several challenges were encountered and overcome. Below are some of the notable difficulties we faced:

1. **Positioning and Alignment:** Another challenge arose from ensuring that the Mandelbrot and Julia sets were correctly positioned within the image frame. Incorrect positioning could result in portions of the sets being cropped out or displayed at the wrong location, leading to distorted or incomplete images. To address this, we carefully mapped the pixel coordinates of the sets to the corresponding points in the complex plane, ensuring accurate alignment and placement within the image frame. Additionally, we adjusted the scaling and translation parameters to center the sets within the image and maintain their relative proportions.

2. **Memory Management:** One significant issue arose when attempting to store large arrays representing the Mandelbrot and Julia sets in memory. As the size of the dataset increased, memory consumption became a limiting factor, leading to out-of-memory errors and segmentation

faults. To mitigate this issue, we adopted a strategy of calculating the sets in smaller, manageable chunks. Instead of trying to store the entire dataset in memory simultaneously, we processed and wrote sections of the sets to file incrementally, thereby reducing the memory footprint and avoiding memory exhaustion.

3. **Finding Suitable Color Maps:** Selecting an appropriate color map for visualizing the Mandelbrot and Julia sets posed a significant challenge. Aesthetic considerations, such as color harmony, contrast, and perceptual uniformity, were crucial in ensuring that the rendered images were visually appealing and easy to interpret. We experimented with different color schemes, gradients, and palettes to find a balance between aesthetics and functional readability. This iterative process involved evaluating the impact of color choices on the clarity and interoperability of the sets, as well as their overall visual impact.

3 Algorithm Results

We implemented the parallel fractal generation algorithm using MPI and conducted experiments to evaluate its performance. The algorithm was tested on a computational cluster, with varying numbers of MPI processes (8, 16, 32, and 64) and a fixed image size of 10000x10000 pixels.

The table below summarizes the computation time (in seconds) for generating Mandelbrot and Julia sets using different numbers of MPI processes:

Table 1: Computation Time for Fractal Generation

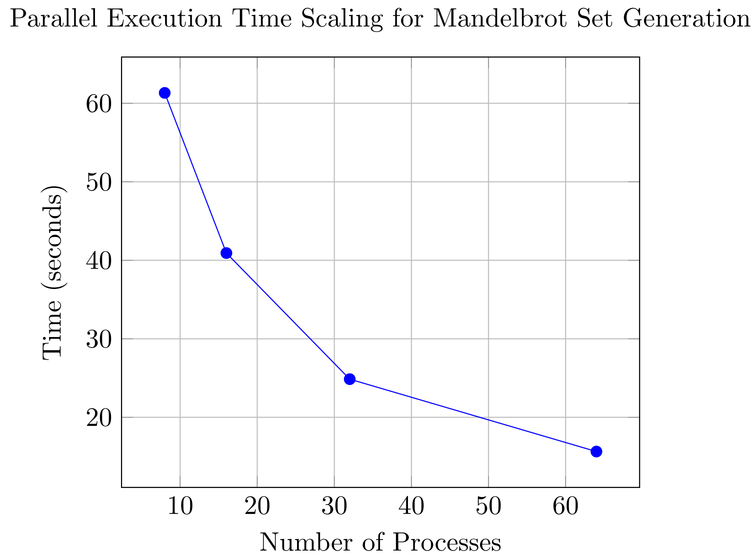
Total Processes	**Mandelbrot (s)**	**Julia Set (s)**
8	61.3226	65.8631
16	40.92115	47.71285
32	24.85912	29.1897
64	15.63836	17.74668

4 Benchmarking

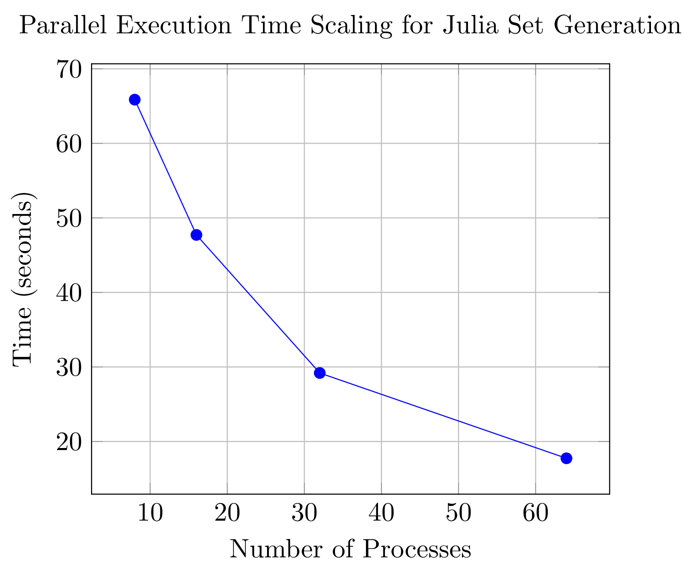
The benchmarking results presented here evaluate the performance of the parallelized algorithms for generating fractal images of size 10000x10000 pixels. The key statistics we are reviewing are the total computation time. The computations were completed on the Teach SHARCnet cluster utilizing MPI, with parallel execution on 8, 16, 32, and 64 processes. By examining these metrics across different numbers of processes, we aim to assess the scalability and efficiency of our parallel implementations for generating fractal images.

4.1 Performance Analysis

4.1.1 Mandelbrot Generation Graph (Image Size: 10000x10000)



4.1.2 Julia Set Generation Graph (Image Size: 10000x10000)



4.1.3 Performance Conclusion

The benchmarking results for both Mandelbrot and Julia set generation demonstrate a clear trend of decreasing total computation time as the number of processes increases. For Mandelbrot set generation, the total computation time decreased from 61.3226 seconds with 8 processes to 15.63836 seconds with 64 processes. Similarly, for Julia set generation, the total computation time decreased from 65.8631 seconds with 8 processes to 17.74668 seconds with 64 processes.

This trend highlights the scalability and efficiency of our parallel implementations. By distributing the workload across multiple processes, we were able to harness power of parallel processing, resulting in significantly reduced computation times for generating fractal images. The results underscore the effectiveness of parallel computing in accelerating complex computational tasks and optimizing resource utilization.

4.2 MPI Wtime Resolution

It is important to note that the resolution of MPI Wtime, a function measuring time in MPI programs, remained consistent at 1.000000e-09 seconds across all experiments. This consistent resolution highlights the accuracy of the timing measurements.

5 Interactive Visualization of High-Resolution Fractal Images

We have developed an interactive visualization platform using a Node.js server and the OpenSeadragon library to explore high-resolution fractal images generated from the Mandelbrot and Julia sets. This provides a dynamic interface to navigate through the intricate patterns of the fractals with deep zoom capabilities.

5.1 Overview

The visualization platform integrates several key components:

1. **Node.js Server**: The Node.js server serves the HTML viewer page, OpenSeadragon library, and Deep Zoom Image (DZI) files. It efficiently handles requests from clients and delivers the necessary content for visualization.

2. **OpenSeadragon Library**: OpenSeadragon is a powerful JavaScript library designed for viewing and exploring high-resolution images with deep zoom capabilities. It enables smooth panning, zooming, and navigation through large image datasets.

3. **Deep Zoom Image (DZI) Files**: DZI files represent large images as a collection of image tiles at various resolutions. These files, generated from fractal images using tools like VIPS, allow for seamless navigation and zooming without loading the entire image at once.

5.2 Functionality

The visualization platform offers the following features:

- **Interactive Exploration**: Users can load and explore high-resolution fractal images generated from Mandelbrot and Julia sets.

- **Deep Zoom Capabilities**: The platform enables smooth panning and zooming, allowing users to navigate through fractal patterns at different scales.

- **Detail Analysis**: Users can analyze intricate details and structures of the fractals by zooming in to individual regions of interest.

5.3 Purpose

The interactive visualization platform serves as a tool for in-depth exploration and analysis of pixel-dense fractal images. By leveraging the deep zoom capabilities of OpenSeadragon and the efficient serving of DZI files through Node.js, we provide a simple way for visualizing and understanding the complexity of fractal geometry.

6 Conclusion

In this paper, we presented parallel implementations for generating the Mandelbrot and Julia sets using the Message Passing Interface (MPI) library. Leveraging parallel processing, our aim was not only to accelerate the computation but also to enhance the visualization of these fractals, enabling exploration and analysis of their mesmerizing patterns and structures.

The parallel algorithms demonstrated significant efficiency gains, as evidenced by the benchmarking results. Both Mandelbrot and Julia set generation exhibited scalability, with total computation time decreasing consistently as the number of MPI processes increased. This trend underscores the effectiveness of parallel computing in optimizing the generation of fractal images, allowing for faster exploration and analysis of their complex geometries.

Moreover, the development of an interactive visualization platform using Node.js server and OpenSeadragon library enhances the accessibility and usability of the generated fractal images. By leveraging deep zoom capabilities and efficient serving of DZI files, this makes it easier to explore the intricate details and structures of the fractals at different scales.

Our work highlights the importance of efficient workload distribution utilization in parallel computing, showcasing how leveraging multiple processes can effectively distribute the computational workload, leading to reductions in total computation time.

In summary, our work contributes to the growing body of knowledge on parallel computing applications in fractal generation, paving the way for future developments and discoveries in computational mathematics.

7 Appendices

7.1 Code Implementation

7.1.1 Mandelbrot

```c

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // Needed for usleep function
#include <time.h> // Needed for time functions
#include <math.h>
#include <png.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define WIDTH 100
#define HEIGHT 100
#define MAX_ITERATION 1000

#define COLOR_CHOICE 1

void calculate_mandelbrot_array_range(int width, int start_row, int end_row, int *
    result);
void map_to_color(int iteration, int *red, int *green, int *blue, int color_choice);
double hue_to_rgb(double hue, double saturation, double lightness);


void calculate_mandelbrot_array_range(int width, int start_row, int end_row, int *
    result) {

    // Define the boundaries of the Mandelbrot set in the complex plane
    double xmin = -2.0, xmax = 1.0, ymin = -1.5, ymax = 1.5;

    // Calculate the step size in the x and y directions
    double xstep = (xmax - xmin) / width;
    double ystep = (ymax - ymin) / HEIGHT;

    // Iterate through rows within the specified range
    for (int y = start_row; y < end_row; y++) {
        // Calculate the imaginary part of the complex number corresponding to the
    current row
        double y0 = ymin + y * ystep;

        // Iterate through columns
        for (int x = 0; x < width; x++) {
            // Calculate the real part of the complex number corresponding to the
    current column
            double x0 = xmin + x * xstep;

```

```c
            // Initialize variables for the real and imaginary parts of the complex
    number
            double xx = 0.0, yy = 0.0;

            // Initialize the iteration count
            int iteration = 0;

            // Iterate until the magnitude of the complex number exceeds 2 or maximum
    iterations are reached
            while (xx * xx + yy * yy <= 4.0 && iteration < MAX_ITERATION) {
                // Update the real part of the complex number
                double xtemp = xx * xx - yy * yy + x0;

                // Update the imaginary part of the complex number
                yy = 2 * xx * yy + y0;

                // Update the real part for the next iteration
                xx = xtemp;

                // Increment the iteration count
                iteration++;
            }

            // Store the result in the result array based on the iteration count
            if (iteration == MAX_ITERATION) {
                result[(y - start_row) * width + x] = 0;  // Inside Mandelbrot set
            } else {
                result[(y - start_row) * width + x] = iteration;  // Outside
    Mandelbrot set
            }
        }
    }
}

void map_to_color(int iteration, int *red, int *green, int *blue, int color_choice) {
    double t;
    double hue;

    if (iteration == 0 || iteration == MAX_ITERATION) {
        // Inside remains black
        *red = *green = *blue = 0;
        return;
    } else {
        // Normalize iteration count to range [0, 1]
        t = (double)iteration / MAX_ITERATION;
    }

    switch (color_choice) {
        case 1:
            // Smooth gradient scheme (blue to white)
            *red = (int)(9 * (1 - t) * t * t * t * 255);
            *green = (int)(15 * (1 - t) * (1 - t) * t * t * 255);
            *blue = (int)(8.5 * (1 - t) * (1 - t) * (1 - t) * t * 255);
            break;

        case 2:
            // New color scheme with better distribution for large canvases:
            // Wider range of colors, starting with blue, cycling through green,
    yellow, red, and back to blue
            hue = 0.66 * t + 0.16;  // Adjust hue range for desired colors
            *red = (int)(96 * (1 - fabs(4 * hue - 2)) * 255);
            *green = (int)(144 * (fabs(4 * hue - 3) - fabs(4 * hue - 1)) * 255);
            *blue = (int)(85 * (1 - fabs(2 * hue - 1)) * 255);
            break;

        case 3:
            // Fire-like color scheme:
            // Starts with dark red, transitions to orange and yellow, then fades to
    white
            *red = (int)(255 * sqrt(t));
            *green = (int)(185 * sqrt(t));
```

```
            *blue = (int)(85 * sqrt(t));
            break;

        case 4:
            // Autumn foliage color scheme
            *red = (int)(255 * (0.5 + 0.5 * cos(2 * M_PI * t)));
            *green = (int)(255 * (0.2 + 0.3 * cos(2 * M_PI * t + 2 * M_PI / 3)));
            *blue = (int)(255 * (0.1 + 0.1 * cos(2 * M_PI * t + 4 * M_PI / 3)));
            break;

        case 5:
            // Ocean-like color scheme:
            // Starts with deep blue, transitions to lighter blues and greens
            *red = (int)(50 + 205 * t);
            *green = (int)(100 + 155 * t);
            *blue = (int)(150 + 105 * t);
            break;

        case 6:
            // Rainbow color scheme:
            // Cycles through the rainbow spectrum
            *red = (int)(255 * (1 - t));
            *green = (int)(255 * fabs(0.5 - t));
            *blue = (int)(255 * t);
            break;

        case 7:
            // Desert color scheme:
            // Starts with sandy brown, transitions to reddish-brown
            *red = (int)(220 * (1 - t));
            *green = (int)(180 * (1 - t));
            *blue = (int)(130 * (1 - t));
            break;

        case 8:
            // Pastel color scheme:
            // Delicate, soft colors inspired by pastel art
            *red = (int)(220 * (0.5 + 0.5 * sin(2 * M_PI * t)));
            *green = (int)(205 * (0.5 + 0.5 * sin(2 * M_PI * t + 2 * M_PI / 3)));
            *blue = (int)(255 * (0.5 + 0.5 * sin(2 * M_PI * t + 4 * M_PI / 3)));
            break;

        case 9:
            // Night sky color scheme:
            // Deep blue hues with hints of purple, reminiscent of a starry night sky
            *red = (int)(20 + 100 * sin(2 * M_PI * t));
            *green = (int)(10 + 50 * sin(2 * M_PI * t + M_PI / 2));
            *blue = (int)(50 + 100 * sin(2 * M_PI * t + M_PI));
            break;

        case 10:
            // Smooth transition through the entire spectrum, with black for the "inside"
            // Inside remains black
             // Transition through the entire spectrum outside
            hue = 0.5 + t * 0.5; // Smoothly increase hue from 0.5 (green) to 1.0 (red)
            *red = (int)(255 * hue_to_rgb(hue, 0.8, 0.5));
            *green = (int)(255 * hue_to_rgb(hue - 1.0/3, 0.8, 0.5));
            *blue = (int)(255 * hue_to_rgb(hue - 2.0/3, 0.8, 0.5));
            break;

        case 11:
            // Twilight Sky Color Scheme
            *red = (int)(0 * (1 - t) + 30 * t);
            *green = (int)(0 * (1 - t) + 0 * t);
            *blue = (int)(128 * (1 - t) + 128 * t);
            break;

        case 12:
            // Summer Sunset Color Scheme:
```

```c
            *red = (int)(255 * (1 - t));
            *green = (int)(69 * (1 - t) + 128 * t);
            *blue = (int)(0 * (1 - t) + 128 * t);
            break;

        case 13:
            hue = 6.0 * t;
            int sector = (int)floor(hue); // Integer part determines color sector
            double offset = hue - sector;

            switch (sector % 6) {
                case 0:
                    *red = 255;
                    *green = (int)(255 * offset);
                    *blue = 0;
                    break;
                case 1:
                    *red = (int)(255 * (1 - offset));
                    *green = 255;
                    *blue = 0;
                    break;
                case 2:
                    *red = 0;
                    *green = 255;
                    *blue = (int)(255 * offset);
                    break;
                case 3:
                    *red = 0;
                    *green = (int)(255 * (1 - offset));
                    *blue = 255;
                    break;
                case 4:
                    *red = (int)(255 * offset);
                    *green = 0;
                    *blue = 255;
                    break;
                case 5:
                    *red = 255;
                    *green = 0;
                    *blue = (int)(255 * (1 - offset));
                    break;
            }
            break;

        case 14:
            hue = 6.0 * t;
            *red = (int)(255 * (1 - fabs(4 * hue - 2)) * pow(fabs(4 * hue - 2), 2)); // Emphasize red
            *green = (int)(255 * fabs(4 * hue - 3) * pow(fabs(4 * hue - 3), 1.5)); // Emphasize green less
            *blue = (int)(255 * fabs(4 * hue - 4)); // Blue not emphasized
            break;

        case 15:
            hue = fmod(t, 1.0); // Wrap hue value between 0 and 1
            float angle = M_PI * 2.0 * hue;
            float radius = 1.0;

            *red = (int)(255 * (radius * cos(angle) + 0.5));
            *green = (int)(255 * (radius * sin(angle) + 0.5));
            *blue = (int)(255 * (1.0 - radius));
            break;

        case 16:
            hue = 6.0 * t;
            int sector2 = (int)floor(hue); // Integer part determines color sector
            double offset2 = hue - sector2;

            switch (sector2 % 6) {
                case 0:
                    *red = 255;
```

```
                    *green = (int)(255 * offset2);
                    *blue = 0;
                    break;
                case 1:
                    *red = (int)(255 * (1 - offset2));
                    *green = 255;
                    *blue = 0;
                    break;
                case 2:
                    *red = 0;
                    *green = 255;
                    *blue = (int)(255 * offset2);
                    break;
                case 3:
                    *red = 0;
                    *green = (int)(255 * (1 - offset2));
                    *blue = 255;
                    break;
                case 4:
                    *red = (int)(255 * offset2);
                    *green = 0;
                    *blue = 255;
                    break;
                case 5:
                    *red = 255;
                    *green = 0;
                    *blue = (int)(255 * (1 - offset2));
                    break;
            }

            // Add secondary inverted rainbow with transparency
            switch ((sector2 + 3) % 6) {
                case 0:
                    *red += (int)(128 * (1 - offset2));
                    break;
                case 1:
                    *green += (int)(128 * (1 - offset2));
                    break;
                case 2:
                    *blue += (int)(128 * (1 - offset2));
                    break;
                case 3:
                    *red += (int)(128 * offset2);
                    break;
                case 4:
                    *green += (int)(128 * offset2);
                    break;
                case 5:
                    *blue += (int)(128 * offset2);
                    break;
            }
            break;

        case 17:
            {
            double y = t * 2.0 - 1.0; // Normalize and scale y-axis for effect
            double h1 = fmod(atan2(y, 1.0) / (2.0 * M_PI) + 0.5, 1.0); // Hue for fire
            double h2 = fmod(atan2(-y, 1.0) / (2.0 * M_PI) + 0.5, 1.0); // Hue for ice

            // Fire colors with smooth transition
            *red = (int)(255 * (1.0 - h1) * pow(h1, 2));
            *green = (int)(255 * h1 * pow(h1, 1.5));
            *blue = 0;

            // Ice colors with smooth transition and transparency
            *red += (int)(128 * (1.0 - h2) * pow(h2, 3));
            *green += (int)(128 * h2 * pow(h2, 2));
            *blue += (int)(255 * h2);
            }
            break;

```

```
        case 18:
            // Spring color scheme:
            // Starts with light green, transitions to vibrant greens and yellows
            *red = (int)(150 * (1 - t));
            *green = (int)(255 * t);
            *blue = (int)(100 * (1 - t) + 155 * t);
            break;

        case 19:
            // Sakura color scheme:
            // Shades of pink and white, resembling cherry blossom petals
            *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
            *green = (int)(200 * (0.5 + 0.5 * sin(2 * M_PI * t)));
            *blue = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
            break;

        case 20:
            // Autumn Leaves color scheme:
            // Starts with deep orange, transitions to red and brown hues
            *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
            *green = (int)(100 * (0.5 + 0.5 * sin(2 * M_PI * t)));
            *blue = (int)(0 * (0.9 + 0.1 * cos(2 * M_PI * t)));
            break;

        case 21:
            // Mystic Forest color scheme:
            // Mixture of dark greens and purples, evoking a mysterious atmosphere
            *red = (int)(30 + 50 * sin(2 * M_PI * t));
            *green = (int)(80 + 50 * sin(2 * M_PI * t + M_PI / 2));
            *blue = (int)(100 + 50 * sin(2 * M_PI * t + M_PI));
            break;

        case 22:
            // Golden Sunset color scheme:
            // Starts with warm yellow, transitions to orange and deep red
            *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
            *green = (int)(200 * (0.6 + 0.4 * sin(2 * M_PI * t)));
            *blue = (int)(50 * (0.5 + 0.5 * sin(2 * M_PI * t)));
            break;

        case 23:
            hue = 0.66 * t + 0.16; // Adjust hue range for desired colors
            *red = (int)(96 * (1 - fabs(4 * hue - 2)) * pow(fabs(4 * hue - 2), 0.5)); // Emphasize red with smooth falloff
            *green = (int)(144 * (fabs(4 * hue - 3) - fabs(4 * hue - 1)) * pow(fabs(4 * hue - 2.5), 0.75)); // Emphasize green with smoother falloff
            *blue = (int)(85 * (1 - fabs(2 * hue - 1)) * pow(1.0 - fabs(2 * hue - 1), 1.25)); // Blue fades smoothly to black

            // Adjust falloff power terms and multipliers for finer control

            break;

        default:
            // Default to black for unknown color choice
            *red = *green = *blue = 0;
            break;
    }
}

double hue_to_rgb(double hue, double saturation, double lightness) {
    // Calculate chroma (color intensity)
    double chroma = (1 - fabs(2 * lightness - 1)) * saturation;

    // Convert hue to hue_mod, which is a value between 0 and 6
    double hue_mod = hue * 6;

    // Calculate intermediate value x
    double x = chroma * (1 - fabs(fmod(hue_mod, 2) - 1));

    double r, g, b;
```

```c

    // Determine RGB components based on hue_mod
    if (hue_mod < 1) {
        r = chroma;
        g = x;
        b = 0;
    } else if (hue_mod < 2) {
        r = x;
        g = chroma;
        b = 0;
    } else if (hue_mod < 3) {
        r = 0;
        g = chroma;
        b = x;
    } else if (hue_mod < 4) {
        r = 0;
        g = x;
        b = chroma;
    } else if (hue_mod < 5) {
        r = x;
        g = 0;
        b = chroma;
    } else {
        r = chroma;
        g = 0;
        b = x;
    }

    // Calculate lightness modifier (m)
    double m = lightness - 0.5 * chroma;

    // Return final RGB value by adding lightness modifier to each RGB component
    return r + m;
}

int main(int argc, char *argv[]) {

    int rank, size;
    double start_time, end_time, elapsed_time, tick;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Returns the precision of the results returned by MPI_Wtime
    tick = MPI_Wtick();

    // Ensures all processes will enter the measured section of the code at the same
    time
    MPI_Barrier(MPI_COMM_WORLD);

    start_time = MPI_Wtime();

   // Determine rows to compute for each process
    int rows_per_process = HEIGHT / size;
    int remaining_rows = HEIGHT % size; // Rows left after distributing evenly

    int start_row, end_row;

    if (rank < remaining_rows) {
        // Distribute remaining rows evenly among the first 'remaining_rows' processes
        start_row = rank * (rows_per_process + 1);
        end_row = start_row + (rows_per_process + 1);
    } else {
        // Distribute remaining rows among the remaining processes
        start_row = rank * rows_per_process + remaining_rows;
        end_row = start_row + rows_per_process;
    }

    int local_total_elements = WIDTH * (end_row - start_row);

```

```c
     // Allocate memory for local Mandelbrot sets on each process
    int *local_mandelbrot_set;
    local_mandelbrot_set = malloc(sizeof(int) * local_total_elements);
    if (local_mandelbrot_set == NULL) {
        fprintf(stderr, "Error: Memory allocation failed\n");
        MPI_Finalize();
        return 1;
    }

    // Generate the Mandelbrot set
    calculate_mandelbrot_array_range(WIDTH, start_row, end_row, local_mandelbrot_set);

    // Send and Receive local results (instead of Gather)
    if (rank != 0) {

        // Send local_mandelbrot_set size (consider uneven distribution)
        MPI_Send(&local_total_elements, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(local_mandelbrot_set, local_total_elements, MPI_INT, 0, 1, MPI_COMM_WORLD);

    } else { // Root process receives from all processes

        char filename[100]; // Buffer to hold the filename

        // Format the filename with height and width
        snprintf(filename, sizeof(filename), "mandelbrot_%dx%d_color-%d_iterations-%d.png", WIDTH, HEIGHT, COLOR_CHOICE, MAX_ITERATION);

        // Open file for writing (binary mode)
        FILE *fp = fopen(filename, "wb");
        if (!fp) {
            fprintf(stderr, "Error opening file for writing\n");
            return 1;
        }

        // Create PNG structures
        png_structp png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
        if (!png_ptr) {
            fclose(fp);
            fprintf(stderr, "Error creating PNG write structure\n");
            return 1;
        }

        png_infop info_ptr = png_create_info_struct(png_ptr);
        if (!info_ptr) {
            png_destroy_write_struct(&png_ptr, NULL);
            fclose(fp);
            fprintf(stderr, "Error creating PNG info structure\n");
            return 1;
        }

        // Error handling setup
        if (setjmp(png_jmpbuf(png_ptr))) {
            png_destroy_write_struct(&png_ptr, &info_ptr);
            fclose(fp);
            fprintf(stderr, "Error during PNG creation\n");
            return 1;
        }

        // Set image properties
        png_set_IHDR(png_ptr, info_ptr, WIDTH, HEIGHT, 8, PNG_COLOR_TYPE_RGBA, PNG_INTERLACE_NONE,
                     PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_BASE);

        // Initialize I/O for writing to file
        png_init_io(png_ptr, fp);

        // Write PNG header (including all required information)
        png_write_info(png_ptr, info_ptr);

```

```
        // Initialize current pixel count
        unsigned long long current_pixel = 0;
        int* array;
        int received_size;

        for (int i = 0; i < size; i++) {

            if (i == 0){

                array = local_mandelbrot_set;
                received_size = local_total_elements;

            } else {

                // Allocate memory for received data
                received_size;
                MPI_Recv(&received_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                array = malloc(sizeof(int) * received_size);
                MPI_Recv(array, received_size, MPI_INT, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            }

            // Allocate memory for section of image data
            png_bytep image_data = (png_bytep)malloc(received_size * 4 * sizeof(png_byte)); // 4 bytes per pixel for RGBA

            if (!image_data) {
                fprintf(stderr, "Error allocating memory for image data\n");
                png_destroy_write_struct(&png_ptr, &info_ptr);
                fclose(fp);
                return 1;
            }

            // Fill image data with solid blue color
            for (int y = 0; y < (received_size/WIDTH); y++) {
                for (int x = 0; x < WIDTH; x++) {

                    // Get pixel colour
                    int red, green, blue;
                    map_to_color(array[y * WIDTH + x], &red, &green, &blue, COLOR_CHOICE);

                    // Calculate offset for pixel
                    int offset = x * 4; // 4 bytes per pixel

                    // Assign RGBA values to image data
                    image_data[offset] = red;          // Red
                    image_data[offset + 1] = green;    // Green
                    image_data[offset + 2] = blue;     // Blue
                    image_data[offset + 3] = 255;      // Alpha (fully opaque)

                    // Increment current pixel count
                    current_pixel++;

                    // Print progress percentage
                    if (current_pixel % (WIDTH / 10) == 0){
                        printf("\rPNG Pixel Progress: %.2f%% Pixel Count: %llu", (double)current_pixel / ((double)WIDTH * HEIGHT) * 100, current_pixel);
                    }
                }

                // Write current row to PNG
                png_write_row(png_ptr, &image_data[0]);
            }

            free(image_data);
            free(array);
        }
```

```

        // Print newline after progress percentage
        printf("\n");

        // Write the end of the PNG information
        png_write_end(png_ptr, info_ptr);

        // Clean up
        png_destroy_write_struct(&png_ptr, &info_ptr);
        fclose(fp);

        // Print success message
        printf("\nPNG image created successfully: %s \n", filename);

    }

    // Ensures all processes will enter the measured section of the code at the same time
    MPI_Barrier(MPI_COMM_WORLD);

    end_time = MPI_Wtime();

    // Calculate the elapsed time
    elapsed_time = end_time - start_time;

    MPI_Finalize();

    // if rank is 0, print out the time analysis for merging arrays
    if (rank == 0) {
        printf("\n********** PNG Creation Time **********\n");
        printf("Total processes: %d\n", size);
        printf("Total computation time: %e seconds\n", elapsed_time);
        printf("Computation time per process: %e seconds\n", elapsed_time / size);
        printf("Resolution of MPI_Wtime: %e seconds\n", tick);
        printf("%d,%d,%d,%e,%e,%e",WIDTH, HEIGHT, size, elapsed_time, (elapsed_time / size), tick);
    }

    return 0;

}
```

7.1.2 Julia Sets

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // Needed for usleep function
#include <time.h> // Needed for time functions
#include <math.h>
#include <png.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define WIDTH 100
#define HEIGHT 100
#define MAX_ITERATION 1000

#define REAL_NUMBER -0.8
#define IMAGINARY_NUMBER -0.089

// so far 1, 3, 16 are actually kind of nice lolol
// 14 are a bit odd
#define COLOR_CHOICE 1

typedef struct {
    double real;
    double imag;
} Complex;
```

```c

void calculate_julia_array_range(int width, int start_row, int end_row, int *result,
    double real, double imaginary);
void map_to_color(int iteration, int *red, int *green, int *blue, int color_choice);
double hue_to_rgb(double hue, double saturation, double lightness);


void calculate_julia_array_range(int width, int start_row, int end_row, int *result,
    double real, double imaginary) {

    // Define constant for Julia set
    Complex constant = {.real = real, .imag = imaginary}; // Example constant

    // Iterate through rows within the specified range
    for (int y = start_row; y < end_row; y++) {
        for (int x = 0; x < width; x++) {

            // Map pixel coordinates (x, y) directly to the rectangular region in the
    complex plane
            // The complex plane is mapped to a rectangular region defined by:
            // - Real part (x-axis): Range from -1.75 (leftmost) to 1.75 (rightmost)
            // - Imaginary part (y-axis): Range from -1.75 (bottom) to 1.75 (top)
            // The width and height of the rectangular region are adjusted to match
    the aspect ratio of the image.
            Complex z = {.real = x / (double)width * 3.5 - 1.75, .imag = y / (double)
    HEIGHT * 3.5 - 1.75};
            int iteration = 0;
            while (z.real * z.real + z.imag * z.imag <= 4.0 && iteration <
    MAX_ITERATION) {
                double temp = z.real * z.real - z.imag * z.imag + constant.real;
                z.imag = 2.0 * z.real * z.imag + constant.imag;
                z.real = temp;
                iteration++;
            }

            // Store the result in the result array based on the iteration count
            if (iteration == MAX_ITERATION) {
                result[(y - start_row) * width + x] = 0;  // Inside Mandelbrot set
            } else {
                result[(y - start_row) * width + x] = iteration;  // Outside
    Mandelbrot set
            }
        }
    }
}

void map_to_color(int iteration, int *red, int *green, int *blue, int color_choice) {
    double t;
    double hue;

    if (iteration == 0 || iteration == MAX_ITERATION) {
        // Inside remains black
        *red = *green = *blue = 0;
        return;
    } else {
        // Normalize iteration count to range [0, 1]
        t = (double)iteration / MAX_ITERATION;
    }

    switch (color_choice) {
        case 1:
            // Smooth gradient scheme (blue to white)
            *red = (int)(9 * (1 - t) * t * t * t * 255);
            *green = (int)(15 * (1 - t) * (1 - t) * t * t * 255);
            *blue = (int)(8.5 * (1 - t) * (1 - t) * (1 - t) * t * 255);
            break;

        case 2:
            // New color scheme with better distribution for large canvases:
            // Wider range of colors, starting with blue, cycling through green,
    yellow, red, and back to blue
```

```c
            hue = 0.66 * t + 0.16;  // Adjust hue range for desired colors
            *red = (int)(96 * (1 - fabs(4 * hue - 2)) * 255);
            *green = (int)(144 * (fabs(4 * hue - 3) - fabs(4 * hue - 1)) * 255);
            *blue = (int)(85 * (1 - fabs(2 * hue - 1)) * 255);
            break;

        case 3:
            // Fire-like color scheme:
            // Starts with dark red, transitions to orange and yellow, then fades to white
            *red = (int)(255 * sqrt(t));
            *green = (int)(185 * sqrt(t));
            *blue = (int)(85 * sqrt(t));
            break;

        case 4:
            // Autumn foliage color scheme
            *red = (int)(255 * (0.5 + 0.5 * cos(2 * M_PI * t)));
            *green = (int)(255 * (0.2 + 0.3 * cos(2 * M_PI * t + 2 * M_PI / 3)));
            *blue = (int)(255 * (0.1 + 0.1 * cos(2 * M_PI * t + 4 * M_PI / 3)));
            break;

        case 5:
            // Ocean-like color scheme:
            // Starts with deep blue, transitions to lighter blues and greens
            *red = (int)(50 + 205 * t);
            *green = (int)(100 + 155 * t);
            *blue = (int)(150 + 105 * t);
            break;

        case 6:
            // Rainbow color scheme:
            // Cycles through the rainbow spectrum
            *red = (int)(255 * (1 - t));
            *green = (int)(255 * fabs(0.5 - t));
            *blue = (int)(255 * t);
            break;

        case 7:
            // Desert color scheme:
            // Starts with sandy brown, transitions to reddish-brown
            *red = (int)(220 * (1 - t));
            *green = (int)(180 * (1 - t));
            *blue = (int)(130 * (1 - t));
            break;

        case 8:
            // Pastel color scheme:
            // Delicate, soft colors inspired by pastel art
            *red = (int)(220 * (0.5 + 0.5 * sin(2 * M_PI * t)));
            *green = (int)(205 * (0.5 + 0.5 * sin(2 * M_PI * t + 2 * M_PI / 3)));
            *blue = (int)(255 * (0.5 + 0.5 * sin(2 * M_PI * t + 4 * M_PI / 3)));
            break;

        case 9:
            // Night sky color scheme:
            // Deep blue hues with hints of purple, reminiscent of a starry night sky
            *red = (int)(20 + 100 * sin(2 * M_PI * t));
            *green = (int)(10 + 50 * sin(2 * M_PI * t + M_PI / 2));
            *blue = (int)(50 + 100 * sin(2 * M_PI * t + M_PI));
            break;

        case 10:
            // Smooth transition through the entire spectrum, with black for the "inside"
            // Inside remains black
             // Transition through the entire spectrum outside
            hue = 0.5 + t * 0.5; // Smoothly increase hue from 0.5 (green) to 1.0 (red)
            *red = (int)(255 * hue_to_rgb(hue, 0.8, 0.5));
            *green = (int)(255 * hue_to_rgb(hue - 1.0/3, 0.8, 0.5));
```

```
            *blue = (int)(255 * hue_to_rgb(hue - 2.0/3, 0.8, 0.5));
            break;

        case 11:
            // Twilight Sky Color Scheme
            *red = (int)(0 * (1 - t) + 30 * t);
            *green = (int)(0 * (1 - t) + 0 * t);
            *blue = (int)(128 * (1 - t) + 128 * t);
            break;

        case 12:
            // Summer Sunset Color Scheme:
            *red = (int)(255 * (1 - t));
            *green = (int)(69 * (1 - t) + 128 * t);
            *blue = (int)(0 * (1 - t) + 128 * t);
            break;

        case 13:
            hue = 6.0 * t;
            int sector = (int)floor(hue); // Integer part determines color sector
            double offset = hue - sector;

            switch (sector % 6) {
                case 0:
                    *red = 255;
                    *green = (int)(255 * offset);
                    *blue = 0;
                    break;
                case 1:
                    *red = (int)(255 * (1 - offset));
                    *green = 255;
                    *blue = 0;
                    break;
                case 2:
                    *red = 0;
                    *green = 255;
                    *blue = (int)(255 * offset);
                    break;
                case 3:
                    *red = 0;
                    *green = (int)(255 * (1 - offset));
                    *blue = 255;
                    break;
                case 4:
                    *red = (int)(255 * offset);
                    *green = 0;
                    *blue = 255;
                    break;
                case 5:
                    *red = 255;
                    *green = 0;
                    *blue = (int)(255 * (1 - offset));
                    break;
            }
            break;

        case 14:
            hue = 6.0 * t;
            *red = (int)(255 * (1 - fabs(4 * hue - 2)) * pow(fabs(4 * hue - 2), 2)); // Emphasize red
            *green = (int)(255 * fabs(4 * hue - 3) * pow(fabs(4 * hue - 3), 1.5)); // Emphasize green less
            *blue = (int)(255 * fabs(4 * hue - 4)); // Blue not emphasized
            break;

        case 15:
            hue = fmod(t, 1.0); // Wrap hue value between 0 and 1
            float angle = M_PI * 2.0 * hue;
            float radius = 1.0;

            *red = (int)(255 * (radius * cos(angle) + 0.5));
```

```
            *green = (int)(255 * (radius * sin(angle) + 0.5));
            *blue = (int)(255 * (1.0 - radius));
            break;

        case 16:
            hue = 6.0 * t;
            int sector2 = (int)floor(hue); // Integer part determines color sector
            double offset2 = hue - sector2;

            switch (sector2 % 6) {
                case 0:
                    *red = 255;
                    *green = (int)(255 * offset2);
                    *blue = 0;
                    break;
                case 1:
                    *red = (int)(255 * (1 - offset2));
                    *green = 255;
                    *blue = 0;
                    break;
                case 2:
                    *red = 0;
                    *green = 255;
                    *blue = (int)(255 * offset2);
                    break;
                case 3:
                    *red = 0;
                    *green = (int)(255 * (1 - offset2));
                    *blue = 255;
                    break;
                case 4:
                    *red = (int)(255 * offset2);
                    *green = 0;
                    *blue = 255;
                    break;
                case 5:
                    *red = 255;
                    *green = 0;
                    *blue = (int)(255 * (1 - offset2));
                    break;
            }

            // Add secondary inverted rainbow with transparency
            switch ((sector2 + 3) % 6) {
                case 0:
                    *red += (int)(128 * (1 - offset2));
                    break;
                case 1:
                    *green += (int)(128 * (1 - offset2));
                    break;
                case 2:
                    *blue += (int)(128 * (1 - offset2));
                    break;
                case 3:
                    *red += (int)(128 * offset2);
                    break;
                case 4:
                    *green += (int)(128 * offset2);
                    break;
                case 5:
                    *blue += (int)(128 * offset2);
                    break;
            }
            break;

        case 17:
            {
            double y = t * 2.0 - 1.0; // Normalize and scale y-axis for effect
            double h1 = fmod(atan2(y, 1.0) / (2.0 * M_PI) + 0.5, 1.0); // Hue for fire
            double h2 = fmod(atan2(-y, 1.0) / (2.0 * M_PI) + 0.5, 1.0); // Hue for ice

```

```
            // Fire colors with smooth transition
            *red = (int)(255 * (1.0 - h1) * pow(h1, 2));
            *green = (int)(255 * h1 * pow(h1, 1.5));
            *blue = 0;

            // Ice colors with smooth transition and transparency
            *red += (int)(128 * (1.0 - h2) * pow(h2, 3));
            *green += (int)(128 * h2 * pow(h2, 2));
            *blue += (int)(255 * h2);
            }
            break;

        case 18:
            // Spring color scheme:
            // Starts with light green, transitions to vibrant greens and yellows
            *red = (int)(150 * (1 - t));
            *green = (int)(255 * t);
            *blue = (int)(100 * (1 - t) + 155 * t);
            break;

        case 19:
            // Sakura color scheme:
            // Shades of pink and white, resembling cherry blossom petals
            *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
            *green = (int)(200 * (0.5 + 0.5 * sin(2 * M_PI * t)));
            *blue = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
            break;

        case 20:
            // Autumn Leaves color scheme:
            // Starts with deep orange, transitions to red and brown hues
            *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
            *green = (int)(100 * (0.5 + 0.5 * sin(2 * M_PI * t)));
            *blue = (int)(0 * (0.9 + 0.1 * cos(2 * M_PI * t)));
            break;

        case 21:
            // Mystic Forest color scheme:
            // Mixture of dark greens and purples, evoking a mysterious atmosphere
            *red = (int)(30 + 50 * sin(2 * M_PI * t));
            *green = (int)(80 + 50 * sin(2 * M_PI * t + M_PI / 2));
            *blue = (int)(100 + 50 * sin(2 * M_PI * t + M_PI));
            break;

        case 22:
            // Golden Sunset color scheme:
            // Starts with warm yellow, transitions to orange and deep red
            *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
            *green = (int)(200 * (0.6 + 0.4 * sin(2 * M_PI * t)));
            *blue = (int)(50 * (0.5 + 0.5 * sin(2 * M_PI * t)));
            break;

        case 23:
            hue = 0.66 * t + 0.16; // Adjust hue range for desired colors
            *red = (int)(96 * (1 - fabs(4 * hue - 2)) * pow(fabs(4 * hue - 2), 0.5));
    // Emphasize red with smooth falloff
            *green = (int)(144 * (fabs(4 * hue - 3) - fabs(4 * hue - 1)) * pow(fabs(4
    * hue - 2.5), 0.75)); // Emphasize green with smoother falloff
            *blue = (int)(85 * (1 - fabs(2 * hue - 1)) * pow(1.0 - fabs(2 * hue - 1),
    1.25)); // Blue fades smoothly to black

            // Adjust falloff power terms and multipliers for finer control

            break;

        default:
            // Default to black for unknown color choice
            *red = *green = *blue = 0;
            break;
    }
}
```

```c

double hue_to_rgb(double hue, double saturation, double lightness) {
    // Calculate chroma (color intensity)
    double chroma = (1 - fabs(2 * lightness - 1)) * saturation;

    // Convert hue to hue_mod, which is a value between 0 and 6
    double hue_mod = hue * 6;

    // Calculate intermediate value x
    double x = chroma * (1 - fabs(fmod(hue_mod, 2) - 1));

    double r, g, b;

    // Determine RGB components based on hue_mod
    if (hue_mod < 1) {
        r = chroma;
        g = x;
        b = 0;
    } else if (hue_mod < 2) {
        r = x;
        g = chroma;
        b = 0;
    } else if (hue_mod < 3) {
        r = 0;
        g = chroma;
        b = x;
    } else if (hue_mod < 4) {
        r = 0;
        g = x;
        b = chroma;
    } else if (hue_mod < 5) {
        r = x;
        g = 0;
        b = chroma;
    } else {
        r = chroma;
        g = 0;
        b = x;
    }

    // Calculate lightness modifier (m)
    double m = lightness - 0.5 * chroma;

    // Return final RGB value by adding lightness modifier to each RGB component
    return r + m;
}

int main(int argc, char *argv[]) {

    int rank, size;
    double start_time, end_time, elapsed_time, tick;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Returns the precision of the results returned by MPI_Wtime
    tick = MPI_Wtick();

    // Ensures all processes will enter the measured section of the code at the same time
    MPI_Barrier(MPI_COMM_WORLD);

    start_time = MPI_Wtime();

   // Determine rows to compute for each process
    int rows_per_process = HEIGHT / size;
    int remaining_rows = HEIGHT % size; // Rows left after distributing evenly

    int start_row, end_row;

```

```c
    if (rank < remaining_rows) {
        // Distribute remaining rows evenly among the first 'remaining_rows' processes
        start_row = rank * (rows_per_process + 1);
        end_row = start_row + (rows_per_process + 1);
    } else {
        // Distribute remaining rows among the remaining processes
        start_row = rank * rows_per_process + remaining_rows;
        end_row = start_row + rows_per_process;
    }

    int local_total_elements = WIDTH * (end_row - start_row);

     // Allocate memory for local Mandelbrot sets on each process
    int *local_julia_set;
    local_julia_set = malloc(sizeof(int) * local_total_elements);
    if (local_julia_set == NULL) {
        fprintf(stderr, "Error: Memory allocation failed\n");
        MPI_Finalize();
        return 1;
    }

    // Generate the Mandelbrot set
    calculate_julia_array_range(WIDTH, start_row, end_row, local_julia_set,
    REAL_NUMBER, IMAGINARY_NUMBER);

    // Send and Receive local results (instead of Gather)
    if (rank != 0) {

        // Send local_julia_set size (consider uneven distribution)
        MPI_Send(&local_total_elements, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(local_julia_set, local_total_elements, MPI_INT, 0, 1, MPI_COMM_WORLD)
    ;

    } else { // Root process receives from all processes

        char filename[100]; // Buffer to hold the filename

        // Format the filename with height and width
        snprintf(filename, sizeof(filename), "julia-set_%dx%d_color-%d_iterations-%
    d_real-%f_imaginary-%f.png", WIDTH, HEIGHT, COLOR_CHOICE, MAX_ITERATION,
    REAL_NUMBER, IMAGINARY_NUMBER);

        // Open file for writing (binary mode)
        FILE *fp = fopen(filename, "wb");
        if (!fp) {
            fprintf(stderr, "Error opening file for writing\n");
            return 1;
        }

        // Create PNG structures
        png_structp png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL,
    NULL, NULL);
        if (!png_ptr) {
            fclose(fp);
            fprintf(stderr, "Error creating PNG write structure\n");
            return 1;
        }

        png_infop info_ptr = png_create_info_struct(png_ptr);
        if (!info_ptr) {
            png_destroy_write_struct(&png_ptr, NULL);
            fclose(fp);
            fprintf(stderr, "Error creating PNG info structure\n");
            return 1;
        }

        // Error handling setup
        if (setjmp(png_jmpbuf(png_ptr))) {
            png_destroy_write_struct(&png_ptr, &info_ptr);
            fclose(fp);
            fprintf(stderr, "Error during PNG creation\n");
```

```
        return 1;
    }

    // Set image properties
    png_set_IHDR(png_ptr, info_ptr, WIDTH, HEIGHT, 8, PNG_COLOR_TYPE_RGBA, PNG_INTERLACE_NONE,
                PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_BASE);

    // Initialize I/O for writing to file
    png_init_io(png_ptr, fp);

    // Write PNG header (including all required information)
    png_write_info(png_ptr, info_ptr);

    // Initialize current pixel count
    unsigned long long current_pixel = 0;
    int* array;
    int received_size;

    for (int i = 0; i < size; i++) {

        if (i == 0){

            array = local_julia_set;
            received_size = local_total_elements;

        } else {

            // Allocate memory for received data
            received_size;
            MPI_Recv(&received_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            array = malloc(sizeof(int) * received_size);
            MPI_Recv(array, received_size, MPI_INT, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        }

        // Allocate memory for section of image data
        png_bytep image_data = (png_bytep)malloc(received_size * 4 * sizeof(png_byte)); // 4 bytes per pixel for RGBA

        if (!image_data) {
            fprintf(stderr, "Error allocating memory for image data\n");
            png_destroy_write_struct(&png_ptr, &info_ptr);
            fclose(fp);
            return 1;
        }

        // Fill image data with solid blue color
        for (int y = 0; y < (received_size/WIDTH); y++) {
            for (int x = 0; x < WIDTH; x++) {

                // Get pixel colour
                int red, green, blue;
                map_to_color(array[y * WIDTH + x], &red, &green, &blue, COLOR_CHOICE);

                // Calculate offset for pixel
                int offset = x * 4; // 4 bytes per pixel

                // Assign RGBA values to image data
                image_data[offset] = red;         // Red
                image_data[offset + 1] = green;   // Green
                image_data[offset + 2] = blue;    // Blue
                image_data[offset + 3] = 255;     // Alpha (fully opaque)

                // Increment current pixel count
                current_pixel++;

```

```c
                    // Print progress percentage
                    if (current_pixel % (WIDTH / 10) == 0){
                        printf("\rPNG Pixel Progress: %.2f%% Pixel Count: %llu", (double)current_pixel / ((double)WIDTH * HEIGHT) * 100, current_pixel);
                    }
                }

                // Write current row to PNG
                png_write_row(png_ptr, &image_data[0]);
            }

            free(image_data);
            free(array);
        }

        // Print newline after progress percentage
        printf("\n");

        // Write the end of the PNG information
        png_write_end(png_ptr, info_ptr);

        // Clean up
        png_destroy_write_struct(&png_ptr, &info_ptr);
        fclose(fp);

        // Print success message
        printf("\nPNG image created successfully: %s \n", filename);

    }

    // Ensures all processes will enter the measured section of the code at the same time
    MPI_Barrier(MPI_COMM_WORLD);

    end_time = MPI_Wtime();

    // Calculate the elapsed time
    elapsed_time = end_time - start_time;

    MPI_Finalize();

    // if rank is 0, print out the time analysis for merging arrays
    if (rank == 0) {
        printf("\n********** PNG Creation Time **********\n");
        printf("Total processes: %d\n", size);
        printf("Total computation time: %e seconds\n", elapsed_time);
        printf("Computation time per process: %e seconds\n", elapsed_time / size);
        printf("Resolution of MPI_Wtime: %e seconds\n", tick);
        printf("%d,%d,%d,%e,%e,%e,%f,%f",WIDTH, HEIGHT, size, elapsed_time, (elapsed_time / size), tick, REAL_NUMBER, IMAGINARY_NUMBER);
    }

    return 0;

}
```