

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 Mabilia - 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

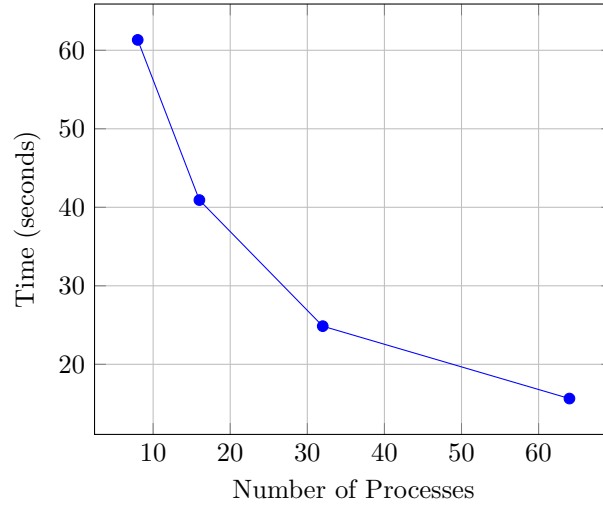
4 Benchmarking

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

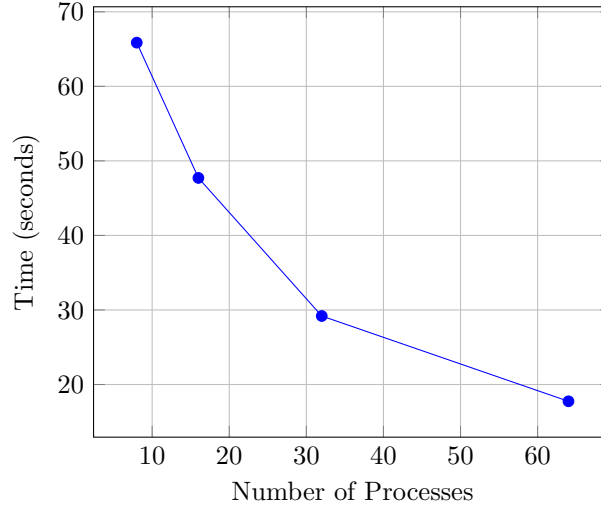
4.1.1 Mandelbrot Generation Graph (Image Size: 10000x10000)

Parallel Execution Time Scaling for Mandelbrot Set Generation



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

Parallel Execution Time Scaling for Julia Set Generation



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

```

1
2 #include <mpi.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h> // Needed for usleep function
6 #include <time.h> // Needed for time functions
7 #include <math.h>
8 #include <png.h>
9
10 #ifndef M_PI
11 #define M_PI 3.14159265358979323846
12 #endif
13
14 #define WIDTH 100
15 #define HEIGHT 100
16 #define MAX_ITERATION 1000
17
18 #define COLOR_CHOICE 1
19
20 void calculate_mandelbrot_array_range(int width, int start_row, int end_row, int *
    result);
21 void map_to_color(int iteration, int *red, int *green, int *blue, int color_choice);
22 double hue_to_rgb(double hue, double saturation, double lightness);
23
24
25 void calculate_mandelbrot_array_range(int width, int start_row, int end_row, int *
    result) {
26
27     // Define the boundaries of the Mandelbrot set in the complex plane
28     double xmin = -2.0, xmax = 1.0, ymin = -1.5, ymax = 1.5;
29
30     // Calculate the step size in the x and y directions
31     double xstep = (xmax - xmin) / width;
32     double ystep = (ymax - ymin) / HEIGHT;
33
34     // Iterate through rows within the specified range
35     for (int y = start_row; y < end_row; y++) {
36         // Calculate the imaginary part of the complex number corresponding to the
    current row
37         double y0 = ymin + y * ystep;
38
39         // Iterate through columns
40         for (int x = 0; x < width; x++) {
41             // Calculate the real part of the complex number corresponding to the
    current column
42             double x0 = xmin + x * xstep;
43

```

```

44         // Initialize variables for the real and imaginary parts of the complex
number
45         double xx = 0.0, yy = 0.0;
46
47         // Initialize the iteration count
48         int iteration = 0;
49
50         // Iterate until the magnitude of the complex number exceeds 2 or maximum
iterations are reached
51         while (xx * xx + yy * yy <= 4.0 && iteration < MAX_ITERATION) {
52             // Update the real part of the complex number
53             double xtemp = xx * xx - yy * yy + x0;
54
55             // Update the imaginary part of the complex number
56             yy = 2 * xx * yy + y0;
57
58             // Update the real part for the next iteration
59             xx = xtemp;
60
61             // Increment the iteration count
62             iteration++;
63         }
64
65         // Store the result in the result array based on the iteration count
66         if (iteration == MAX_ITERATION) {
67             result[(y - start_row) * width + x] = 0; // Inside Mandelbrot set
68         } else {
69             result[(y - start_row) * width + x] = iteration; // Outside
Mandelbrot set
70         }
71     }
72 }
73 }
74
75 void map_to_color(int iteration, int *red, int *green, int *blue, int color_choice) {
76     double t;
77     double hue;
78
79     if (iteration == 0 || iteration == MAX_ITERATION) {
80         // Inside remains black
81         *red = *green = *blue = 0;
82         return;
83     } else {
84         // Normalize iteration count to range [0, 1]
85         t = (double)iteration / MAX_ITERATION;
86     }
87
88     switch (color_choice) {
89         case 1:
90             // Smooth gradient scheme (blue to white)
91             *red = (int)(9 * (1 - t) * t * t * t * 255);
92             *green = (int)(15 * (1 - t) * (1 - t) * t * t * 255);
93             *blue = (int)(8.5 * (1 - t) * (1 - t) * (1 - t) * t * 255);
94             break;
95
96         case 2:
97             // New color scheme with better distribution for large canvases:
98             // Wider range of colors, starting with blue, cycling through green,
yellow, red, and back to blue
99             hue = 0.66 * t + 0.16; // Adjust hue range for desired colors
100             *red = (int)(96 * (1 - fabs(4 * hue - 2)) * 255);
101             *green = (int)(144 * (fabs(4 * hue - 3) - fabs(4 * hue - 1)) * 255);
102             *blue = (int)(85 * (1 - fabs(2 * hue - 1)) * 255);
103             break;
104
105         case 3:
106             // Fire-like color scheme:
107             // Starts with dark red, transitions to orange and yellow, then fades to
white
108             *red = (int)(255 * sqrt(t));
109             *green = (int)(185 * sqrt(t));

```



```

110         *blue = (int)(85 * sqrt(t));
111         break;
112
113     case 4:
114         // Autumn foliage color scheme
115         *red = (int)(255 * (0.5 + 0.5 * cos(2 * M_PI * t)));
116         *green = (int)(255 * (0.2 + 0.3 * cos(2 * M_PI * t + 2 * M_PI / 3)));
117         *blue = (int)(255 * (0.1 + 0.1 * cos(2 * M_PI * t + 4 * M_PI / 3)));
118         break;
119
120     case 5:
121         // Ocean-like color scheme:
122         // Starts with deep blue, transitions to lighter blues and greens
123         *red = (int)(50 + 205 * t);
124         *green = (int)(100 + 155 * t);
125         *blue = (int)(150 + 105 * t);
126         break;
127
128     case 6:
129         // Rainbow color scheme:
130         // Cycles through the rainbow spectrum
131         *red = (int)(255 * (1 - t));
132         *green = (int)(255 * fabs(0.5 - t));
133         *blue = (int)(255 * t);
134         break;
135
136     case 7:
137         // Desert color scheme:
138         // Starts with sandy brown, transitions to reddish-brown
139         *red = (int)(220 * (1 - t));
140         *green = (int)(180 * (1 - t));
141         *blue = (int)(130 * (1 - t));
142         break;
143
144     case 8:
145         // Pastel color scheme:
146         // Delicate, soft colors inspired by pastel art
147         *red = (int)(220 * (0.5 + 0.5 * sin(2 * M_PI * t)));
148         *green = (int)(205 * (0.5 + 0.5 * sin(2 * M_PI * t + 2 * M_PI / 3)));
149         *blue = (int)(255 * (0.5 + 0.5 * sin(2 * M_PI * t + 4 * M_PI / 3)));
150         break;
151
152     case 9:
153         // Night sky color scheme:
154         // Deep blue hues with hints of purple, reminiscent of a starry night sky
155         *red = (int)(20 + 100 * sin(2 * M_PI * t));
156         *green = (int)(10 + 50 * sin(2 * M_PI * t + M_PI / 2));
157         *blue = (int)(50 + 100 * sin(2 * M_PI * t + M_PI));
158         break;
159
160     case 10:
161         // Smooth transition through the entire spectrum, with black for the "
inside"
162         // Inside remains black
163         // Transition through the entire spectrum outside
164         hue = 0.5 + t * 0.5; // Smoothly increase hue from 0.5 (green) to 1.0 (red
)
165         *red = (int)(255 * hue_to_rgb(hue, 0.8, 0.5));
166         *green = (int)(255 * hue_to_rgb(hue - 1.0/3, 0.8, 0.5));
167         *blue = (int)(255 * hue_to_rgb(hue - 2.0/3, 0.8, 0.5));
168         break;
169
170     case 11:
171         // Twilight Sky Color Scheme
172         *red = (int)(0 * (1 - t) + 30 * t);
173         *green = (int)(0 * (1 - t) + 0 * t);
174         *blue = (int)(128 * (1 - t) + 128 * t);
175         break;
176
177     case 12:
178         // Summer Sunset Color Scheme:

```

```

179     *red = (int)(255 * (1 - t));
180     *green = (int)(69 * (1 - t) + 128 * t);
181     *blue = (int)(0 * (1 - t) + 128 * t);
182     break;
183
184     case 13:
185         hue = 6.0 * t;
186         int sector = (int)floor(hue); // Integer part determines color sector
187         double offset = hue - sector;
188
189         switch (sector % 6) {
190             case 0:
191                 *red = 255;
192                 *green = (int)(255 * offset);
193                 *blue = 0;
194                 break;
195             case 1:
196                 *red = (int)(255 * (1 - offset));
197                 *green = 255;
198                 *blue = 0;
199                 break;
200             case 2:
201                 *red = 0;
202                 *green = 255;
203                 *blue = (int)(255 * offset);
204                 break;
205             case 3:
206                 *red = 0;
207                 *green = (int)(255 * (1 - offset));
208                 *blue = 255;
209                 break;
210             case 4:
211                 *red = (int)(255 * offset);
212                 *green = 0;
213                 *blue = 255;
214                 break;
215             case 5:
216                 *red = 255;
217                 *green = 0;
218                 *blue = (int)(255 * (1 - offset));
219                 break;
220         }
221         break;
222
223     case 14:
224         hue = 6.0 * t;
225         *red = (int)(255 * (1 - fabs(4 * hue - 2)) * pow(fabs(4 * hue - 2), 2));
226         // Emphasize red
227         *green = (int)(255 * fabs(4 * hue - 3) * pow(fabs(4 * hue - 3), 1.5)); //
228         // Emphasize green less
229         *blue = (int)(255 * fabs(4 * hue - 4)); // Blue not emphasized
230         break;
231
232     case 15:
233         hue = fmod(t, 1.0); // Wrap hue value between 0 and 1
234         float angle = M_PI * 2.0 * hue;
235         float radius = 1.0;
236
237         *red = (int)(255 * (radius * cos(angle) + 0.5));
238         *green = (int)(255 * (radius * sin(angle) + 0.5));
239         *blue = (int)(255 * (1.0 - radius));
240         break;
241
242     case 16:
243         hue = 6.0 * t;
244         int sector2 = (int)floor(hue); // Integer part determines color sector
245         double offset2 = hue - sector2;
246
247         switch (sector2 % 6) {
248             case 0:
249                 *red = 255;

```

```

248         *green = (int)(255 * offset2);
249         *blue = 0;
250         break;
251     case 1:
252         *red = (int)(255 * (1 - offset2));
253         *green = 255;
254         *blue = 0;
255         break;
256     case 2:
257         *red = 0;
258         *green = 255;
259         *blue = (int)(255 * offset2);
260         break;
261     case 3:
262         *red = 0;
263         *green = (int)(255 * (1 - offset2));
264         *blue = 255;
265         break;
266     case 4:
267         *red = (int)(255 * offset2);
268         *green = 0;
269         *blue = 255;
270         break;
271     case 5:
272         *red = 255;
273         *green = 0;
274         *blue = (int)(255 * (1 - offset2));
275         break;
276     }
277
278     // Add secondary inverted rainbow with transparency
279     switch ((sector2 + 3) % 6) {
280     case 0:
281         *red += (int)(128 * (1 - offset2));
282         break;
283     case 1:
284         *green += (int)(128 * (1 - offset2));
285         break;
286     case 2:
287         *blue += (int)(128 * (1 - offset2));
288         break;
289     case 3:
290         *red += (int)(128 * offset2);
291         break;
292     case 4:
293         *green += (int)(128 * offset2);
294         break;
295     case 5:
296         *blue += (int)(128 * offset2);
297         break;
298     }
299     break;
300
301 case 17:
302     {
303         double y = t * 2.0 - 1.0; // Normalize and scale y-axis for effect
304         double h1 = fmod(atan2(y, 1.0) / (2.0 * M_PI) + 0.5, 1.0); // Hue for fire
305         double h2 = fmod(atan2(-y, 1.0) / (2.0 * M_PI) + 0.5, 1.0); // Hue for ice
306
307         // Fire colors with smooth transition
308         *red = (int)(255 * (1.0 - h1) * pow(h1, 2));
309         *green = (int)(255 * h1 * pow(h1, 1.5));
310         *blue = 0;
311
312         // Ice colors with smooth transition and transparency
313         *red += (int)(128 * (1.0 - h2) * pow(h2, 3));
314         *green += (int)(128 * h2 * pow(h2, 2));
315         *blue += (int)(255 * h2);
316     }
317     break;
318

```

```

319     case 18:
320         // Spring color scheme:
321         // Starts with light green, transitions to vibrant greens and yellows
322         *red = (int)(150 * (1 - t));
323         *green = (int)(255 * t);
324         *blue = (int)(100 * (1 - t) + 155 * t);
325         break;
326
327     case 19:
328         // Sakura color scheme:
329         // Shades of pink and white, resembling cherry blossom petals
330         *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
331         *green = (int)(200 * (0.5 + 0.5 * sin(2 * M_PI * t)));
332         *blue = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
333         break;
334
335     case 20:
336         // Autumn Leaves color scheme:
337         // Starts with deep orange, transitions to red and brown hues
338         *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
339         *green = (int)(100 * (0.5 + 0.5 * sin(2 * M_PI * t)));
340         *blue = (int)(0 * (0.9 + 0.1 * cos(2 * M_PI * t)));
341         break;
342
343     case 21:
344         // Mystic Forest color scheme:
345         // Mixture of dark greens and purples, evoking a mysterious atmosphere
346         *red = (int)(30 + 50 * sin(2 * M_PI * t));
347         *green = (int)(80 + 50 * sin(2 * M_PI * t + M_PI / 2));
348         *blue = (int)(100 + 50 * sin(2 * M_PI * t + M_PI));
349         break;
350
351     case 22:
352         // Golden Sunset color scheme:
353         // Starts with warm yellow, transitions to orange and deep red
354         *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
355         *green = (int)(200 * (0.6 + 0.4 * sin(2 * M_PI * t)));
356         *blue = (int)(50 * (0.5 + 0.5 * sin(2 * M_PI * t)));
357         break;
358
359     case 23:
360         hue = 0.66 * t + 0.16; // Adjust hue range for desired colors
361         *red = (int)(96 * (1 - fabs(4 * hue - 2)) * pow(fabs(4 * hue - 2), 0.5));
362         // Emphasize red with smooth falloff
363         *green = (int)(144 * (fabs(4 * hue - 3) - fabs(4 * hue - 1)) * pow(fabs(4
364 * hue - 2.5), 0.75)); // Emphasize green with smoother falloff
365         *blue = (int)(85 * (1 - fabs(2 * hue - 1)) * pow(1.0 - fabs(2 * hue - 1),
366 1.25)); // Blue fades smoothly to black
367
368         // Adjust falloff power terms and multipliers for finer control
369
370         break;
371
372     default:
373         // Default to black for unknown color choice
374         *red = *green = *blue = 0;
375         break;
376 }
377
378 double hue_to_rgb(double hue, double saturation, double lightness) {
379     // Calculate chroma (color intensity)
380     double chroma = (1 - fabs(2 * lightness - 1)) * saturation;
381
382     // Convert hue to hue_mod, which is a value between 0 and 6
383     double hue_mod = hue * 6;
384
385     // Calculate intermediate value x
386     double x = chroma * (1 - fabs(fmod(hue_mod, 2) - 1));
387
388     double r, g, b;

```

```

387
388 // Determine RGB components based on hue_mod
389 if (hue_mod < 1) {
390     r = chroma;
391     g = x;
392     b = 0;
393 } else if (hue_mod < 2) {
394     r = x;
395     g = chroma;
396     b = 0;
397 } else if (hue_mod < 3) {
398     r = 0;
399     g = chroma;
400     b = x;
401 } else if (hue_mod < 4) {
402     r = 0;
403     g = x;
404     b = chroma;
405 } else if (hue_mod < 5) {
406     r = x;
407     g = 0;
408     b = chroma;
409 } else {
410     r = chroma;
411     g = 0;
412     b = x;
413 }
414
415 // Calculate lightness modifier (m)
416 double m = lightness - 0.5 * chroma;
417
418 // Return final RGB value by adding lightness modifier to each RGB component
419 return r + m;
420 }
421
422 int main(int argc, char *argv[]) {
423
424     int rank, size;
425     double start_time, end_time, elapsed_time, tick;
426
427     MPI_Init(&argc, &argv);
428     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
429     MPI_Comm_size(MPI_COMM_WORLD, &size);
430
431     // Returns the precision of the results returned by MPI_Wtime
432     tick = MPI_Wtick();
433
434     // Ensures all processes will enter the measured section of the code at the same
435     // time
436     MPI_Barrier(MPI_COMM_WORLD);
437
438     start_time = MPI_Wtime();
439
440     // Determine rows to compute for each process
441     int rows_per_process = HEIGHT / size;
442     int remaining_rows = HEIGHT % size; // Rows left after distributing evenly
443
444     int start_row, end_row;
445
446     if (rank < remaining_rows) {
447         // Distribute remaining rows evenly among the first 'remaining_rows' processes
448         start_row = rank * (rows_per_process + 1);
449         end_row = start_row + (rows_per_process + 1);
450     } else {
451         // Distribute remaining rows among the remaining processes
452         start_row = rank * rows_per_process + remaining_rows;
453         end_row = start_row + rows_per_process;
454     }
455
456     int local_total_elements = WIDTH * (end_row - start_row);

```

```

457 // Allocate memory for local Mandelbrot sets on each process
458 int *local_mandelbrot_set;
459 local_mandelbrot_set = malloc(sizeof(int) * local_total_elements);
460 if (local_mandelbrot_set == NULL) {
461     fprintf(stderr, "Error: Memory allocation failed\n");
462     MPI_Finalize();
463     return 1;
464 }
465
466 // Generate the Mandelbrot set
467 calculate_mandelbrot_array_range(WIDTH, start_row, end_row, local_mandelbrot_set);
468
469 // Send and Receive local results (instead of Gather)
470 if (rank != 0) {
471
472     // Send local_mandelbrot_set size (consider uneven distribution)
473     MPI_Send(&local_total_elements, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
474     MPI_Send(local_mandelbrot_set, local_total_elements, MPI_INT, 0, 1,
475 MPI_COMM_WORLD);
476
477 } else { // Root process receives from all processes
478
479     char filename[100]; // Buffer to hold the filename
480
481     // Format the filename with height and width
482     snprintf(filename, sizeof(filename), "mandelbrot_%dx%d_color-%d_iterations-%d.
483 png", WIDTH, HEIGHT, COLOR_CHOICE, MAX_ITERATION);
484
485     // Open file for writing (binary mode)
486     FILE *fp = fopen(filename, "wb");
487     if (!fp) {
488         fprintf(stderr, "Error opening file for writing\n");
489         return 1;
490     }
491
492     // Create PNG structures
493     png_structp png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL,
494 NULL, NULL);
495     if (!png_ptr) {
496         fclose(fp);
497         fprintf(stderr, "Error creating PNG write structure\n");
498         return 1;
499     }
500
501     png_info_ptr info_ptr = png_create_info_struct(png_ptr);
502     if (!info_ptr) {
503         png_destroy_write_struct(&png_ptr, NULL);
504         fclose(fp);
505         fprintf(stderr, "Error creating PNG info structure\n");
506         return 1;
507     }
508
509     // Error handling setup
510     if (setjmp(png_jmpbuf(png_ptr))) {
511         png_destroy_write_struct(&png_ptr, &info_ptr);
512         fclose(fp);
513         fprintf(stderr, "Error during PNG creation\n");
514         return 1;
515     }
516
517     // Set image properties
518     png_set_IHDR(png_ptr, info_ptr, WIDTH, HEIGHT, 8, PNG_COLOR_TYPE_RGBA,
519 PNG_INTERLACE_NONE,
520 PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_BASE);
521
522     // Initialize I/O for writing to file
523     png_init_io(png_ptr, fp);
524
525     // Write PNG header (including all required information)
526     png_write_info(png_ptr, info_ptr);

```

```

524 // Initialize current pixel count
525 unsigned long long current_pixel = 0;
526 int* array;
527 int received_size;
528
529 for (int i = 0; i < size; i++) {
530
531     if (i == 0){
532
533         array = local_mandelbrot_set;
534         received_size = local_total_elements;
535
536     } else {
537
538         // Allocate memory for received data
539         received_size;
540         MPI_Recv(&received_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
541
542         array = malloc(sizeof(int) * received_size);
543         MPI_Recv(array, received_size, MPI_INT, i, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
544
545     }
546
547     // Allocate memory for section of image data
548     png_bytep image_data = (png_bytep)malloc(received_size * 4 * sizeof(
png_byte)); // 4 bytes per pixel for RGBA
549
550     if (!image_data) {
551         fprintf(stderr, "Error allocating memory for image data\n");
552         png_destroy_write_struct(&png_ptr, &info_ptr);
553         fclose(fp);
554         return 1;
555     }
556
557     // Fill image data with solid blue color
558     for (int y = 0; y < (received_size/WIDTH); y++) {
559         for (int x = 0; x < WIDTH; x++) {
560
561             // Get pixel colour
562             int red, green, blue;
563             map_to_color(array[y * WIDTH + x], &red, &green, &blue,
COLOR_CHOICE);
564
565             // Calculate offset for pixel
566             int offset = x * 4; // 4 bytes per pixel
567
568             // Assign RGBA values to image data
569             image_data[offset] = red; // Red
570             image_data[offset + 1] = green; // Green
571             image_data[offset + 2] = blue; // Blue
572             image_data[offset + 3] = 255; // Alpha (fully opaque)
573
574             // Increment current pixel count
575             current_pixel++;
576
577             // Print progress percentage
578             if (current_pixel % (WIDTH / 10) == 0){
579                 printf("\rPNG Pixel Progress: %.2f%% Pixel Count: %llu", (
double)current_pixel / ((double)WIDTH * HEIGHT) * 100, current_pixel);
580             }
581         }
582
583         // Write current row to PNG
584         png_write_row(png_ptr, &image_data[0]);
585     }
586
587     free(image_data);
588     free(array);
589 }

```

```

590
591 // Print newline after progress percentage
592 printf("\n");
593
594 // Write the end of the PNG information
595 png_write_end(png_ptr, info_ptr);
596
597 // Clean up
598 png_destroy_write_struct(&png_ptr, &info_ptr);
599 fclose(fp);
600
601 // Print success message
602 printf("\nPNG image created successfully: %s \n", filename);
603
604 }
605
606 // Ensures all processes will enter the measured section of the code at the same
607 // time
608 MPI_Barrier(MPI_COMM_WORLD);
609
610 end_time = MPI_Wtime();
611
612 // Calculate the elapsed time
613 elapsed_time = end_time - start_time;
614
615 MPI_Finalize();
616
617 // if rank is 0, print out the time analysis for merging arrays
618 if (rank == 0) {
619     printf("\n***** PNG Creation Time *****\n");
620     printf("Total processes: %d\n", size);
621     printf("Total computation time: %e seconds\n", elapsed_time);
622     printf("Computation time per process: %e seconds\n", elapsed_time / size);
623     printf("Resolution of MPI_Wtime: %e seconds\n", tick);
624     printf("%d,%d,%d,%e,%e,%e", WIDTH, HEIGHT, size, elapsed_time, (elapsed_time /
625 size), tick);
626 }
627
628 return 0;
629 }

```

7.1.2 Julia Sets

```

1 #include <mpi.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h> // Needed for usleep function
5 #include <time.h> // Needed for time functions
6 #include <math.h>
7 #include <png.h>
8
9 #ifndef M_PI
10 #define M_PI 3.14159265358979323846
11 #endif
12
13 #define WIDTH 100
14 #define HEIGHT 100
15 #define MAX_ITERATION 1000
16
17 #define REAL_NUMBER -0.8
18 #define IMAGINARY_NUMBER -0.089
19
20 // so far 1, 3, 16 are actually kind of nice lolol
21 // 14 are a bit odd
22 #define COLOR_CHOICE 1
23
24 typedef struct {
25     double real;
26     double imag;
27 } Complex;

```



```

28
29 void calculate_julia_array_range(int width, int start_row, int end_row, int *result,
    double real, double imaginary);
30 void map_to_color(int iteration, int *red, int *green, int *blue, int color_choice);
31 double hue_to_rgb(double hue, double saturation, double lightness);
32
33
34 void calculate_julia_array_range(int width, int start_row, int end_row, int *result,
    double real, double imaginary) {
35
36     // Define constant for Julia set
37     Complex constant = {.real = real, .imag = imaginary}; // Example constant
38
39     // Iterate through rows within the specified range
40     for (int y = start_row; y < end_row; y++) {
41         for (int x = 0; x < width; x++) {
42
43             // Map pixel coordinates (x, y) directly to the rectangular region in the
44             complex plane
45             // The complex plane is mapped to a rectangular region defined by:
46             // - Real part (x-axis): Range from -1.75 (leftmost) to 1.75 (rightmost)
47             // - Imaginary part (y-axis): Range from -1.75 (bottom) to 1.75 (top)
48             // The width and height of the rectangular region are adjusted to match
49             the aspect ratio of the image.
50             Complex z = {.real = x / (double)width * 3.5 - 1.75, .imag = y / (double)
HEIGHT * 3.5 - 1.75};
51             int iteration = 0;
52             while (z.real * z.real + z.imag * z.imag <= 4.0 && iteration <
MAX_ITERATION) {
53                 double temp = z.real * z.real - z.imag * z.imag + constant.real;
54                 z.imag = 2.0 * z.real * z.imag + constant.imag;
55                 z.real = temp;
56                 iteration++;
57             }
58
59             // Store the result in the result array based on the iteration count
60             if (iteration == MAX_ITERATION) {
61                 result[(y - start_row) * width + x] = 0; // Inside Mandelbrot set
62             } else {
63                 result[(y - start_row) * width + x] = iteration; // Outside
Mandelbrot set
64             }
65         }
66     }
67 }
68
69 void map_to_color(int iteration, int *red, int *green, int *blue, int color_choice) {
70     double t;
71     double hue;
72
73     if (iteration == 0 || iteration == MAX_ITERATION) {
74         // Inside remains black
75         *red = *green = *blue = 0;
76         return;
77     } else {
78         // Normalize iteration count to range [0, 1]
79         t = (double)iteration / MAX_ITERATION;
80     }
81
82     switch (color_choice) {
83     case 1:
84         // Smooth gradient scheme (blue to white)
85         *red = (int)(9 * (1 - t) * t * t * t * 255);
86         *green = (int)(15 * (1 - t) * (1 - t) * t * t * 255);
87         *blue = (int)(8.5 * (1 - t) * (1 - t) * (1 - t) * t * 255);
88         break;
89
90     case 2:
91         // New color scheme with better distribution for large canvases:
92         // Wider range of colors, starting with blue, cycling through green,
93         yellow, red, and back to blue

```

```

91     hue = 0.66 * t + 0.16; // Adjust hue range for desired colors
92     *red = (int)(96 * (1 - fabs(4 * hue - 2)) * 255);
93     *green = (int)(144 * (fabs(4 * hue - 3) - fabs(4 * hue - 1)) * 255);
94     *blue = (int)(85 * (1 - fabs(2 * hue - 1)) * 255);
95     break;
96
97     case 3:
98         // Fire-like color scheme:
99         // Starts with dark red, transitions to orange and yellow, then fades to
white
100         *red = (int)(255 * sqrt(t));
101         *green = (int)(185 * sqrt(t));
102         *blue = (int)(85 * sqrt(t));
103         break;
104
105     case 4:
106         // Autumn foliage color scheme
107         *red = (int)(255 * (0.5 + 0.5 * cos(2 * M_PI * t)));
108         *green = (int)(255 * (0.2 + 0.3 * cos(2 * M_PI * t + 2 * M_PI / 3)));
109         *blue = (int)(255 * (0.1 + 0.1 * cos(2 * M_PI * t + 4 * M_PI / 3)));
110         break;
111
112     case 5:
113         // Ocean-like color scheme:
114         // Starts with deep blue, transitions to lighter blues and greens
115         *red = (int)(50 + 205 * t);
116         *green = (int)(100 + 155 * t);
117         *blue = (int)(150 + 105 * t);
118         break;
119
120     case 6:
121         // Rainbow color scheme:
122         // Cycles through the rainbow spectrum
123         *red = (int)(255 * (1 - t));
124         *green = (int)(255 * fabs(0.5 - t));
125         *blue = (int)(255 * t);
126         break;
127
128     case 7:
129         // Desert color scheme:
130         // Starts with sandy brown, transitions to reddish-brown
131         *red = (int)(220 * (1 - t));
132         *green = (int)(180 * (1 - t));
133         *blue = (int)(130 * (1 - t));
134         break;
135
136     case 8:
137         // Pastel color scheme:
138         // Delicate, soft colors inspired by pastel art
139         *red = (int)(220 * (0.5 + 0.5 * sin(2 * M_PI * t)));
140         *green = (int)(205 * (0.5 + 0.5 * sin(2 * M_PI * t + 2 * M_PI / 3)));
141         *blue = (int)(255 * (0.5 + 0.5 * sin(2 * M_PI * t + 4 * M_PI / 3)));
142         break;
143
144     case 9:
145         // Night sky color scheme:
146         // Deep blue hues with hints of purple, reminiscent of a starry night sky
147         *red = (int)(20 + 100 * sin(2 * M_PI * t));
148         *green = (int)(10 + 50 * sin(2 * M_PI * t + M_PI / 2));
149         *blue = (int)(50 + 100 * sin(2 * M_PI * t + M_PI));
150         break;
151
152     case 10:
153         // Smooth transition through the entire spectrum, with black for the "
inside"
154         // Inside remains black
155         // Transition through the entire spectrum outside
156         hue = 0.5 + t * 0.5; // Smoothly increase hue from 0.5 (green) to 1.0 (red
)
157         *red = (int)(255 * hue_to_rgb(hue, 0.8, 0.5));
158         *green = (int)(255 * hue_to_rgb(hue - 1.0/3, 0.8, 0.5));

```

```

159         *blue = (int)(255 * hue_to_rgb(hue - 2.0/3, 0.8, 0.5));
160         break;
161
162     case 11:
163         // Twilight Sky Color Scheme
164         *red = (int)(0 * (1 - t) + 30 * t);
165         *green = (int)(0 * (1 - t) + 0 * t);
166         *blue = (int)(128 * (1 - t) + 128 * t);
167         break;
168
169     case 12:
170         // Summer Sunset Color Scheme:
171         *red = (int)(255 * (1 - t));
172         *green = (int)(69 * (1 - t) + 128 * t);
173         *blue = (int)(0 * (1 - t) + 128 * t);
174         break;
175
176     case 13:
177         hue = 6.0 * t;
178         int sector = (int)floor(hue); // Integer part determines color sector
179         double offset = hue - sector;
180
181         switch (sector % 6) {
182             case 0:
183                 *red = 255;
184                 *green = (int)(255 * offset);
185                 *blue = 0;
186                 break;
187             case 1:
188                 *red = (int)(255 * (1 - offset));
189                 *green = 255;
190                 *blue = 0;
191                 break;
192             case 2:
193                 *red = 0;
194                 *green = 255;
195                 *blue = (int)(255 * offset);
196                 break;
197             case 3:
198                 *red = 0;
199                 *green = (int)(255 * (1 - offset));
200                 *blue = 255;
201                 break;
202             case 4:
203                 *red = (int)(255 * offset);
204                 *green = 0;
205                 *blue = 255;
206                 break;
207             case 5:
208                 *red = 255;
209                 *green = 0;
210                 *blue = (int)(255 * (1 - offset));
211                 break;
212         }
213         break;
214
215     case 14:
216         hue = 6.0 * t;
217         *red = (int)(255 * (1 - fabs(4 * hue - 2)) * pow(fabs(4 * hue - 2), 2));
218         // Emphasize red
219         *green = (int)(255 * fabs(4 * hue - 3) * pow(fabs(4 * hue - 3), 1.5)); //
220         // Emphasize green less
221         *blue = (int)(255 * fabs(4 * hue - 4)); // Blue not emphasized
222         break;
223
224     case 15:
225         hue = fmod(t, 1.0); // Wrap hue value between 0 and 1
226         float angle = M_PI * 2.0 * hue;
227         float radius = 1.0;
228
229         *red = (int)(255 * (radius * cos(angle) + 0.5));

```

```

228     *green = (int)(255 * (radius * sin(angle) + 0.5));
229     *blue = (int)(255 * (1.0 - radius));
230     break;
231
232 case 16:
233     hue = 6.0 * t;
234     int sector2 = (int)floor(hue); // Integer part determines color sector
235     double offset2 = hue - sector2;
236
237     switch (sector2 % 6) {
238     case 0:
239         *red = 255;
240         *green = (int)(255 * offset2);
241         *blue = 0;
242         break;
243     case 1:
244         *red = (int)(255 * (1 - offset2));
245         *green = 255;
246         *blue = 0;
247         break;
248     case 2:
249         *red = 0;
250         *green = 255;
251         *blue = (int)(255 * offset2);
252         break;
253     case 3:
254         *red = 0;
255         *green = (int)(255 * (1 - offset2));
256         *blue = 255;
257         break;
258     case 4:
259         *red = (int)(255 * offset2);
260         *green = 0;
261         *blue = 255;
262         break;
263     case 5:
264         *red = 255;
265         *green = 0;
266         *blue = (int)(255 * (1 - offset2));
267         break;
268     }
269
270     // Add secondary inverted rainbow with transparency
271     switch ((sector2 + 3) % 6) {
272     case 0:
273         *red += (int)(128 * (1 - offset2));
274         break;
275     case 1:
276         *green += (int)(128 * (1 - offset2));
277         break;
278     case 2:
279         *blue += (int)(128 * (1 - offset2));
280         break;
281     case 3:
282         *red += (int)(128 * offset2);
283         break;
284     case 4:
285         *green += (int)(128 * offset2);
286         break;
287     case 5:
288         *blue += (int)(128 * offset2);
289         break;
290     }
291     break;
292
293 case 17:
294     {
295         double y = t * 2.0 - 1.0; // Normalize and scale y-axis for effect
296         double h1 = fmod(atan2(y, 1.0) / (2.0 * M_PI) + 0.5, 1.0); // Hue for fire
297         double h2 = fmod(atan2(-y, 1.0) / (2.0 * M_PI) + 0.5, 1.0); // Hue for ice
298

```

```

299         // Fire colors with smooth transition
300         *red = (int)(255 * (1.0 - h1) * pow(h1, 2));
301         *green = (int)(255 * h1 * pow(h1, 1.5));
302         *blue = 0;
303
304         // Ice colors with smooth transition and transparency
305         *red += (int)(128 * (1.0 - h2) * pow(h2, 3));
306         *green += (int)(128 * h2 * pow(h2, 2));
307         *blue += (int)(255 * h2);
308     }
309     break;
310
311 case 18:
312     // Spring color scheme:
313     // Starts with light green, transitions to vibrant greens and yellows
314     *red = (int)(150 * (1 - t));
315     *green = (int)(255 * t);
316     *blue = (int)(100 * (1 - t) + 155 * t);
317     break;
318
319 case 19:
320     // Sakura color scheme:
321     // Shades of pink and white, resembling cherry blossom petals
322     *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
323     *green = (int)(200 * (0.5 + 0.5 * sin(2 * M_PI * t)));
324     *blue = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
325     break;
326
327 case 20:
328     // Autumn Leaves color scheme:
329     // Starts with deep orange, transitions to red and brown hues
330     *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
331     *green = (int)(100 * (0.5 + 0.5 * sin(2 * M_PI * t)));
332     *blue = (int)(0 * (0.9 + 0.1 * cos(2 * M_PI * t)));
333     break;
334
335 case 21:
336     // Mystic Forest color scheme:
337     // Mixture of dark greens and purples, evoking a mysterious atmosphere
338     *red = (int)(30 + 50 * sin(2 * M_PI * t));
339     *green = (int)(80 + 50 * sin(2 * M_PI * t + M_PI / 2));
340     *blue = (int)(100 + 50 * sin(2 * M_PI * t + M_PI));
341     break;
342
343 case 22:
344     // Golden Sunset color scheme:
345     // Starts with warm yellow, transitions to orange and deep red
346     *red = (int)(255 * (0.9 + 0.1 * cos(2 * M_PI * t)));
347     *green = (int)(200 * (0.6 + 0.4 * sin(2 * M_PI * t)));
348     *blue = (int)(50 * (0.5 + 0.5 * sin(2 * M_PI * t)));
349     break;
350
351 case 23:
352     hue = 0.66 * t + 0.16; // Adjust hue range for desired colors
353     *red = (int)(96 * (1 - fabs(4 * hue - 2)) * pow(fabs(4 * hue - 2), 0.5));
354     // Emphasize red with smooth falloff
355     *green = (int)(144 * (fabs(4 * hue - 3) - fabs(4 * hue - 1)) * pow(fabs(4
356 * hue - 2.5), 0.75)); // Emphasize green with smoother falloff
357     *blue = (int)(85 * (1 - fabs(2 * hue - 1)) * pow(1.0 - fabs(2 * hue - 1),
358 1.25)); // Blue fades smoothly to black
359
360     // Adjust falloff power terms and multipliers for finer control
361
362     break;
363
364 default:
365     // Default to black for unknown color choice
366     *red = *green = *blue = 0;
367     break;
368 }

```

```

367
368 double hue_to_rgb(double hue, double saturation, double lightness) {
369     // Calculate chroma (color intensity)
370     double chroma = (1 - fabs(2 * lightness - 1)) * saturation;
371
372     // Convert hue to hue_mod, which is a value between 0 and 6
373     double hue_mod = hue * 6;
374
375     // Calculate intermediate value x
376     double x = chroma * (1 - fabs(fmod(hue_mod, 2) - 1));
377
378     double r, g, b;
379
380     // Determine RGB components based on hue_mod
381     if (hue_mod < 1) {
382         r = chroma;
383         g = x;
384         b = 0;
385     } else if (hue_mod < 2) {
386         r = x;
387         g = chroma;
388         b = 0;
389     } else if (hue_mod < 3) {
390         r = 0;
391         g = chroma;
392         b = x;
393     } else if (hue_mod < 4) {
394         r = 0;
395         g = x;
396         b = chroma;
397     } else if (hue_mod < 5) {
398         r = x;
399         g = 0;
400         b = chroma;
401     } else {
402         r = chroma;
403         g = 0;
404         b = x;
405     }
406
407     // Calculate lightness modifier (m)
408     double m = lightness - 0.5 * chroma;
409
410     // Return final RGB value by adding lightness modifier to each RGB component
411     return r + m;
412 }
413
414 int main(int argc, char *argv[]) {
415
416     int rank, size;
417     double start_time, end_time, elapsed_time, tick;
418
419     MPI_Init(&argc, &argv);
420     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
421     MPI_Comm_size(MPI_COMM_WORLD, &size);
422
423     // Returns the precision of the results returned by MPI_Wtime
424     tick = MPI_Wtick();
425
426     // Ensures all processes will enter the measured section of the code at the same
    time
427     MPI_Barrier(MPI_COMM_WORLD);
428
429     start_time = MPI_Wtime();
430
431     // Determine rows to compute for each process
432     int rows_per_process = HEIGHT / size;
433     int remaining_rows = HEIGHT % size; // Rows left after distributing evenly
434
435     int start_row, end_row;
436

```

```

437     if (rank < remaining_rows) {
438         // Distribute remaining rows evenly among the first 'remaining_rows' processes
439         start_row = rank * (rows_per_process + 1);
440         end_row = start_row + (rows_per_process + 1);
441     } else {
442         // Distribute remaining rows among the remaining processes
443         start_row = rank * rows_per_process + remaining_rows;
444         end_row = start_row + rows_per_process;
445     }
446
447     int local_total_elements = WIDTH * (end_row - start_row);
448
449     // Allocate memory for local Mandelbrot sets on each process
450     int *local_julia_set;
451     local_julia_set = malloc(sizeof(int) * local_total_elements);
452     if (local_julia_set == NULL) {
453         fprintf(stderr, "Error: Memory allocation failed\n");
454         MPI_Finalize();
455         return 1;
456     }
457
458     // Generate the Mandelbrot set
459     calculate_julia_array_range(WIDTH, start_row, end_row, local_julia_set,
460                                REAL_NUMBER, IMAGINARY_NUMBER);
461
462     // Send and Receive local results (instead of Gather)
463     if (rank != 0) {
464         // Send local_julia_set size (consider uneven distribution)
465         MPI_Send(&local_total_elements, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
466         MPI_Send(local_julia_set, local_total_elements, MPI_INT, 0, 1, MPI_COMM_WORLD);
467     };
468
469     } else { // Root process receives from all processes
470
471         char filename[100]; // Buffer to hold the filename
472
473         // Format the filename with height and width
474         snprintf(filename, sizeof(filename), "julia-set_%dx%d_color-%d_iterations-%d_real-%f_imaginary-%f.png", WIDTH, HEIGHT, COLOR_CHOICE, MAX_ITERATION,
475                  REAL_NUMBER, IMAGINARY_NUMBER);
476
477         // Open file for writing (binary mode)
478         FILE *fp = fopen(filename, "wb");
479         if (!fp) {
480             fprintf(stderr, "Error opening file for writing\n");
481             return 1;
482         }
483
484         // Create PNG structures
485         png_structp png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL,
486              NULL, NULL);
487         if (!png_ptr) {
488             fclose(fp);
489             fprintf(stderr, "Error creating PNG write structure\n");
490             return 1;
491         }
492
493         png_info_ptr info_ptr = png_create_info_struct(png_ptr);
494         if (!info_ptr) {
495             png_destroy_write_struct(&png_ptr, NULL);
496             fclose(fp);
497             fprintf(stderr, "Error creating PNG info structure\n");
498             return 1;
499         }
500
501         // Error handling setup
502         if (setjmp(png_jmpbuf(png_ptr))) {
503             png_destroy_write_struct(&png_ptr, &info_ptr);
504             fclose(fp);
505             fprintf(stderr, "Error during PNG creation\n");

```

```

503         return 1;
504     }
505
506     // Set image properties
507     png_set_IHDR(png_ptr, info_ptr, WIDTH, HEIGHT, 8, PNG_COLOR_TYPE_RGBA,
508     PNG_INTERLACE_NONE,
509     PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_BASE);
510
511     // Initialize I/O for writing to file
512     png_init_io(png_ptr, fp);
513
514     // Write PNG header (including all required information)
515     png_write_info(png_ptr, info_ptr);
516
517     // Initialize current pixel count
518     unsigned long long current_pixel = 0;
519     int* array;
520     int received_size;
521
522     for (int i = 0; i < size; i++) {
523         if (i == 0){
524             array = local_julia_set;
525             received_size = local_total_elements;
526
527         } else {
528             // Allocate memory for received data
529             received_size;
530             MPI_Recv(&received_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
531             MPI_STATUS_IGNORE);
532
533             array = malloc(sizeof(int) * received_size);
534             MPI_Recv(array, received_size, MPI_INT, i, 1, MPI_COMM_WORLD,
535             MPI_STATUS_IGNORE);
536
537         }
538
539         // Allocate memory for section of image data
540         png_bytep image_data = (png_bytep)malloc(received_size * 4 * sizeof(
541         png_byte)); // 4 bytes per pixel for RGBA
542
543         if (!image_data) {
544             fprintf(stderr, "Error allocating memory for image data\n");
545             png_destroy_write_struct(&png_ptr, &info_ptr);
546             fclose(fp);
547             return 1;
548         }
549
550         // Fill image data with solid blue color
551         for (int y = 0; y < (received_size/WIDTH); y++) {
552             for (int x = 0; x < WIDTH; x++) {
553                 // Get pixel colour
554                 int red, green, blue;
555                 map_to_color(array[y * WIDTH + x], &red, &green, &blue,
556                 COLOR_CHOICE);
557
558                 // Calculate offset for pixel
559                 int offset = x * 4; // 4 bytes per pixel
560
561                 // Assign RGBA values to image data
562                 image_data[offset] = red;           // Red
563                 image_data[offset + 1] = green;     // Green
564                 image_data[offset + 2] = blue;      // Blue
565                 image_data[offset + 3] = 255;       // Alpha (fully opaque)
566
567                 // Increment current pixel count
568                 current_pixel++;

```



```

569         // Print progress percentage
570         if (current_pixel % (WIDTH / 10) == 0){
571             printf("\rPNG Pixel Progress: %.2f%% Pixel Count: %llu", (
double)current_pixel / ((double)WIDTH * HEIGHT) * 100, current_pixel);
572         }
573     }
574
575     // Write current row to PNG
576     png_write_row(png_ptr, &image_data[0]);
577 }
578
579     free(image_data);
580     free(array);
581 }
582
583     // Print newline after progress percentage
584     printf("\n");
585
586     // Write the end of the PNG information
587     png_write_end(png_ptr, info_ptr);
588
589     // Clean up
590     png_destroy_write_struct(&png_ptr, &info_ptr);
591     fclose(fp);
592
593     // Print success message
594     printf("\nPNG image created successfully: %s \n", filename);
595
596 }
597
598 // Ensures all processes will enter the measured section of the code at the same
time
599 MPI_Barrier(MPI_COMM_WORLD);
600
601 end_time = MPI_Wtime();
602
603 // Calculate the elapsed time
604 elapsed_time = end_time - start_time;
605
606 MPI_Finalize();
607
608 // if rank is 0, print out the time analysis for merging arrays
609 if (rank == 0) {
610     printf("\n***** PNG Creation Time *****\n");
611     printf("Total processes: %d\n", size);
612     printf("Total computation time: %e seconds\n", elapsed_time);
613     printf("Computation time per process: %e seconds\n", elapsed_time / size);
614     printf("Resolution of MPI_Wtime: %e seconds\n", tick);
615     printf("%d,%d,%d,%e,%e,%e,%f,%f",WIDTH, HEIGHT, size, elapsed_time, (
elapsed_time / size), tick, REAL_NUMBER, IMAGINARY_NUMBER);
616 }
617
618 return 0;
619
620 }

```