Jack Woodleigh

Professor Husowitz

Parallel Programming

August 16, 2023

Parallelization of the Mandelbrot Set

Introduction

The Mandelbrot set is a mesmerizing marvel that, through its infinite depth of intricate patterns, illustrates the wonders of the mathematical universe. Parallelizing the computation of the Mandelbrot set also presents a unique challenge. Due to its fractal nature, the parallelization will have asymmetrical workloads depending on the location of the pixel. Regions deep inside the set often require significantly more iterations and computational time than those farther away. This project tackles the challenge of optimizing the process using several different load-balancing techniques.

Method

To begin, the Mandelbrot set is a two-dimensional set that uses the function $f_c(z) = z^2 + c$ to describe points on the complex plane. The function starts with $z = 0$ and c where c represents a point with a real and an imaginary component. The function is applied iteratively, using the output of the previous iteration as the input for the next (e.g., $z, f_c(z), f_c(f_c(z))$). If after many iterations, the magnitude remains bounded then c is

considered a part of the Mandelbrot set. Alternatively, if *z* runs off to infinity *c* is not a part of the set.

To create the fractals that appear in popular images of the set, it is important to first define the range of values and images desired resolution. For this project, the x and y axis will range from [-2,1] and [-1,1], respectively, and image resolution will be set to 2400x1600 pixels. The x values will be represented through the columns of the image and the y values through the rows. Using these definitions, the rows will have numerical values going from -2 to 1, with a steps size equal to $\frac{range\ max - range\ min}{width} = \frac{2-(-1)}{2400} = \frac{3}{2400}$. Similarly, the columns will range from -1 to 1, with a step size of $\frac{range\ max - range\ min}{height} = \frac{1-(-1)}{1600} = \frac{2}{1600}$. As a result, the numerical value of each pixel can then be determined by multiplying these values by their respective row or column number.

It is also important to note the original formula for the Mandelbrot set, $f_c(z) = z^2 + c$, has a singular numerical input for *z*. Therefore, for this to work, the equation must be decomposed into real and imaginary parts. With this in mind, both *z* and *c* can be expressed using both real and imaginary parts. Through further decomposing, the parts can be separated into real and imaginary calculations:

$$z = x + yi$$

$$c = x_0 + yi_0$$

$$f(x, y) = (x + yi)^2 + (x_0 + yi_0)$$

$$f(x, y) = x^2 + 2xyi - y^2 + (x_0 + yi_0)$$

$$f_{real}(x, y) = x^2 - y^2 + x_0$$

$$f_{imaginary}(x, y) = 2xy + y_0$$

The program begins by setting the *x* and *y* values equal to 0, ensuring that *z* starts as 0. It will then iterate through each pixel by first calculating both real and imaginary parts and setting them equal to x and y, respectively. This is because the x-axis represents real numbers while the y-axis is imaginary. For each pixel, the program performs these calculations iteratively, up to 255 times, checking to see if it diverges towards infinity. The color of that pixel is then determined by the number of iterations taken. If the pixel diverges towards infinity quickly it will have a lower pixel value than one that diverges slowly. If the pixel does not diverge at all, then it will have a max value of 255. These values will correspond to the color in the final image.

Additionally, this code can be easily modified to produce the Julia set, given their similarities. Both sets use the same iterative calculation but with different initial values. For the Mandelbrot set, the value of c is set to the current pixel's complex coordinate and z starts at 0. In contrast, for the Julia set, c is a predetermined constant for the entire set, and z is initialized to the current pixel's complex coordinate. Other than an additional if-statement for variable initiation, there are no other changes required to accommodate this change.

However, depending on the resolution, both the Mandelbrot and Julia set can take quite a long time to compute. So, it's beneficial to speed up the process with parallelization. Using OpenMPI, the image can be divided into regions of pixels, each processed separately by different nodes. However, this introduces a load-balancing challenge: regions farther from the set often require significantly less processing time than those closer to it. As a result, depending on the region a node is assigned, it might sit idle waiting for others to finish. This inefficiency can be addressed using the Master-Worker parallelization technique.

The Master-Worker technique involves designating one processing node as the ‘master’ which exclusively delegates work to the other ‘worker nodes’. The ‘master worker’ assigns chunks, based on a predetermined chunk size, to the workers. Once a worker has completed its task, it returns the results and receives another chunk of work. This process continues until the master worker informs the workers that the work is finished. In practice, the master worker assigns a range of pixels to the worker nodes for them to process using the aforementioned calculation. After finishing, the worker will return the results for each pixel, then receive their next assignment. The Master-Worker implementation will make the code more efficient and flexible by allowing nodes to take longer with their calculations while not causing the other nodes to sit idly by as a result.

While Master-Worker implementation significantly improves the efficiency of the node network, there is still one area to potentially improve bottlenecking. Each worker node still faces the same limitations as before the initial implementation of parallelization. So to improve this, instead of running the calculations in sequential order, the problem can once again be parallelized using OpenMP. By splitting up the workers calculations onto threads, the process is further expedited.

While implementing OpenMP can significantly reduce the overall runtime, it's important to note that, by default, OpenMP employs static scheduling. This means that the work is divided evenly among the threads. As a result, if the workload for each iteration is not uniform, this can lead to load imbalance issues, similar to what might be seen in a naive parallel implementation. A straightforward remedy to this issue is to switch from static to dynamic scheduling. Dynamic scheduling allocates small chunks of work to threads as they become available, thus mitigating

potential bottlenecks and offering a load-balancing approach reminiscent of the Master-Worker paradigm.

Results

By applying the Master-Worker parallelization technique in tandem with OpenMP parallel regions, the following images of the Mandelbrot and Julia set were generated:

Figure 1

Color and Grayscale Image of the Mandelbrot set

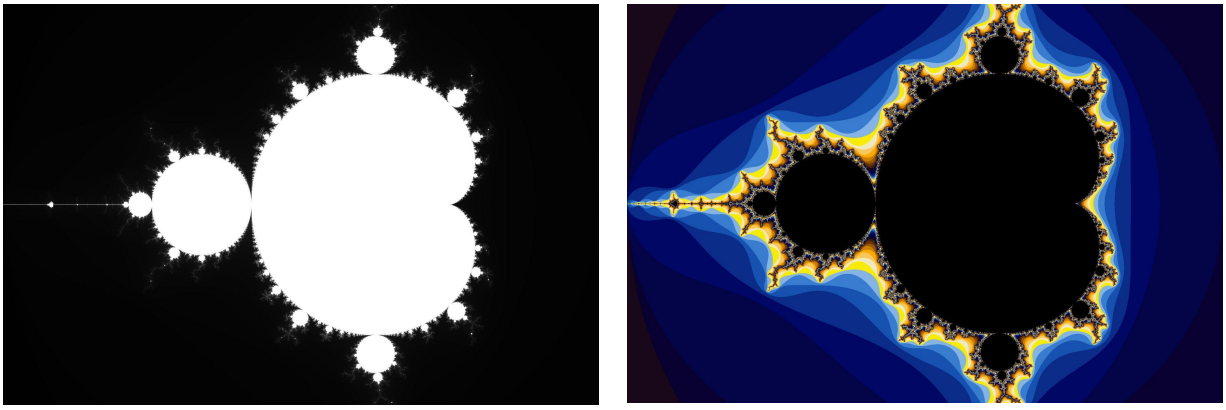
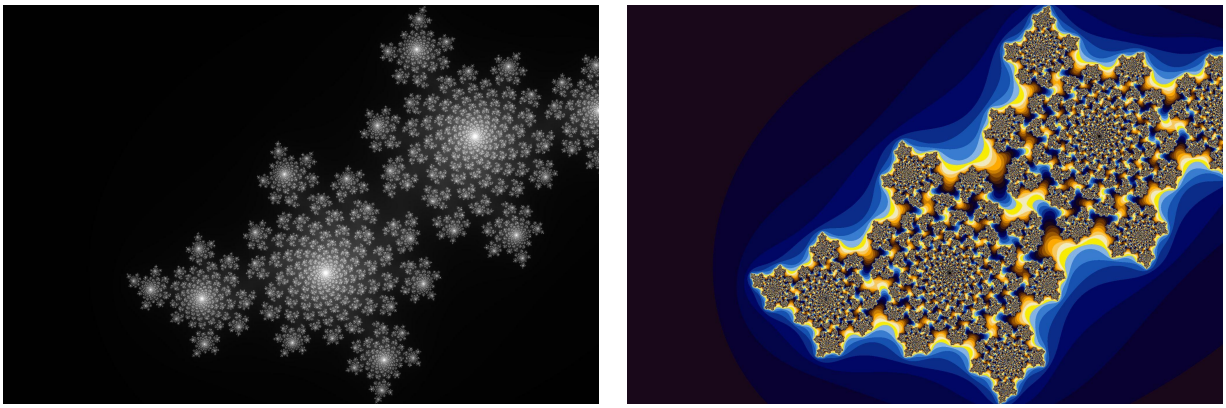


Figure 2

Color and Grayscale Image of the Julia Set at -0.4+0.6i



The following data was collected to measure the runtime performance of the parallelization of the Mandelbrot set.

Figure 3

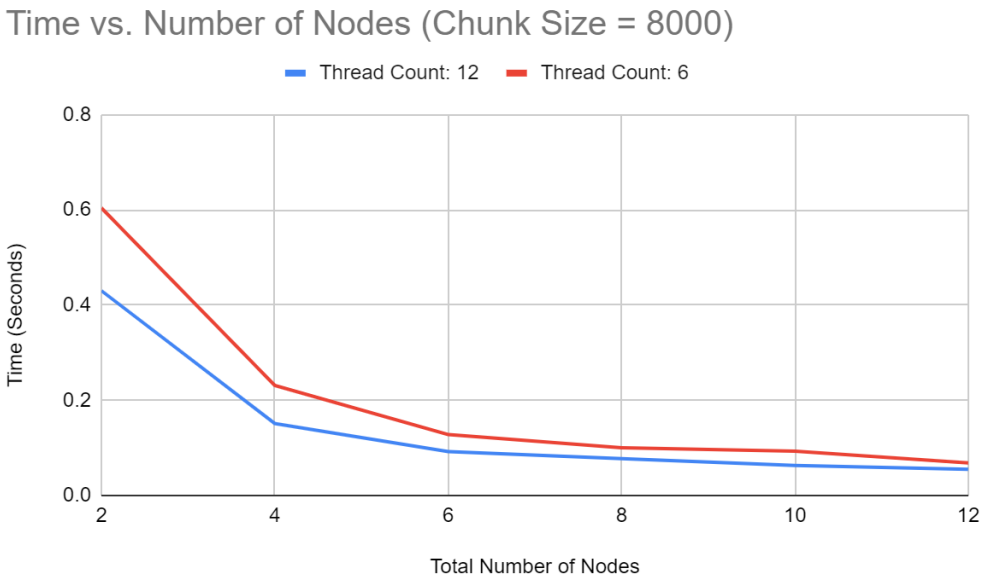


Figure 4

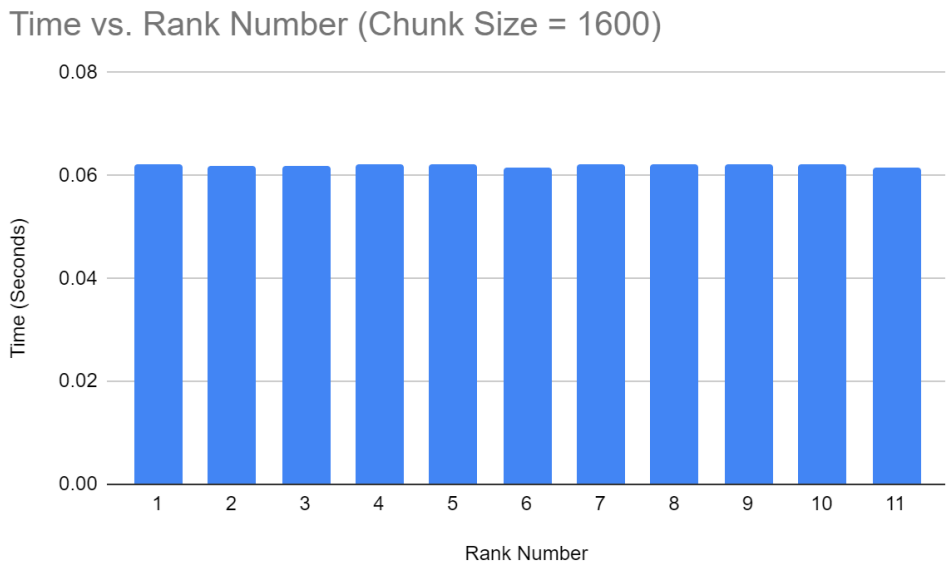
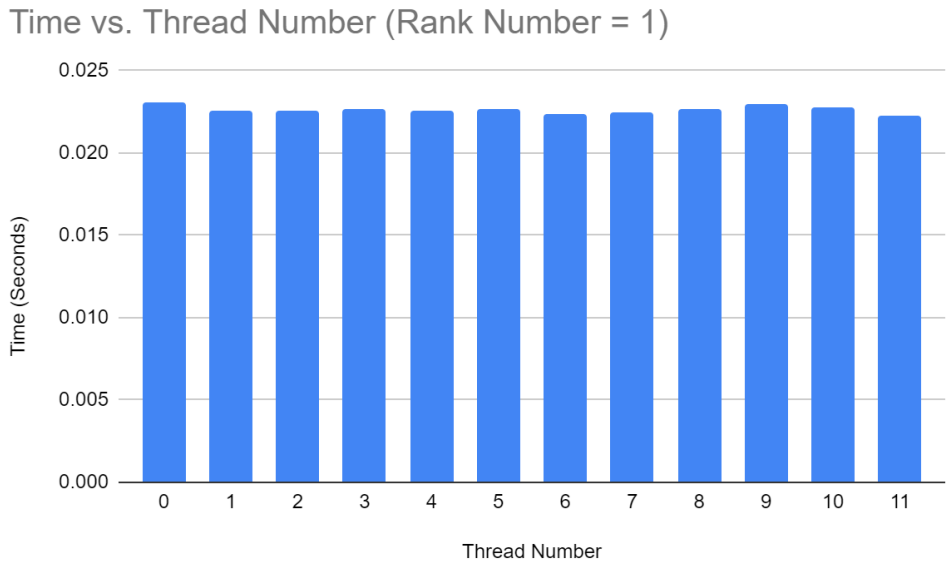


Figure 5



Discussion

The results indicate that not only was the parallelization successful, but it was also successfully load-balanced. Figures 1 and 2 show generated images of both Mandelbrot set and a Julia set in color and grayscale. They highlight their amazing beauty and incredible intricacies, while underscoring the significant computation intensity required to render higher-resolution images.

Figure 3 illustrates the difference between 6 and 12 threads with a varying number of nodes. The disparity is minimal with a larger number of nodes. However, as the node count decreases, a pronounced reduction in runtime becomes evident. This can be primarily attributed to the calculation comprising an increasingly smaller amount of runtime, with communication making up the majority.

Figures 4 and 5 visually illustrate the Master-Worker and OpenMP implementations in action. Across both nodes and threads, the runtimes were notably consistent, highlighting effective load-balancing. Such uniformity in the runtimes across nodes and threads indicates that the available computational resources were optimally utilized.

Conclusion

In conclusion, although the Mandelbrot set poses a challenge to parallelize as a result of its asymmetrical workloads, it is possible to subvert the imbalance by using a combination of the Master-Worker technique and OpenMP with dynamic scheduling. These optimization approaches not only reduce computational time but offer a scalable method for generating higher-resolution images.