Tim Magee

8/15/2024

COMP-3450-05C

The Mandelbrot set is a fractal that is created from a set of mathematical equations. To compute the value for a point in the set, start by taking the x and y coordinates of the point and represent them as a complex number C, where C=x+iy. This is the initial value for the specified point. We also set $Z_0 = 0$ to set ourselves up. From this point on we calculate $Z_{n+1} = Z_n^2 + C$, where n is equal to the nth iteration of the equation. We continue this process until we reach one of two escape conditions. The first way to escape this loop is if $|Z|^2 > 4$. The other way to escape the loop is by reaching a specified maximum number of iterations (for us it was 255). If a point reaches the maximum iterations, we assume that that set of x and y is stable and is considered a part of the Mandelbrot set. If the value of $|Z|^2 > 4$ (or $|Z| > 2$) before reaching the maximum number of iterations, the set of x and y is not part of the Mandelbrot set. After completing our calculations for all points, we can turn our results into an image. Any point that is a part of the Mandelbrot set can be represented by black, and points outside of the Mandelbrot set have their color determined by how many iterations it took for them to become unstable (usually warmer colors mean more iterations to become unstable, and colder colors means less iterations needed).

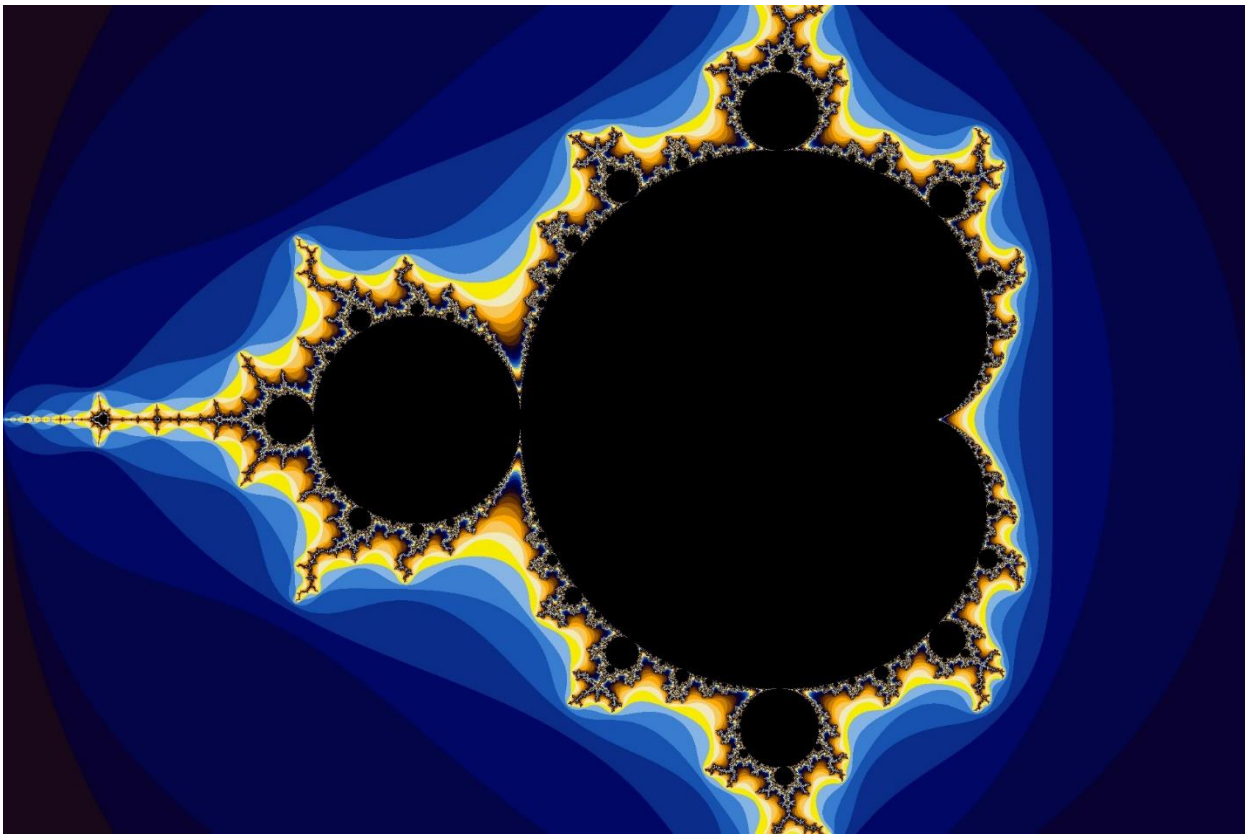

Figure 1 - Result image of the Mandelbrot set

One of the main difficulties of solving the Mandelbrot set through parallel computing is load imbalance. If you were to create a program that solved the Mandelbrot set without any regards to the load imbalance, for example, an implementation where each node calculates a group of rows of pixels, and then the program combines the results from each node into a larger array to convert into an image, the nodes that are responsible for calculating the center areas would be the bottleneck, causing the program to wait for them to finish before combining into the larger array. This can be explained by looking at the final result of the Mandelbrot set (Figure 1). In the image, we can see that the majority of the area that would be calculated by the middle nodes are black. This means that for each of these pixels, the node needs to go through all 255 iterations, whereas nodes that are in charge of the outside areas have much less black and a lot more blue and orange, so each of these pixels will only have to go through a couple of iterations before they reach the $|Z|^2 > 4$ escape condition. To address this issue, a master-worker model using MPI can be implemented. In this approach, the master node manages task distribution by assigning pixel ranges to worker nodes. The master sends these pixel ranges to the worker nodes, which then compute the assigned pixels. As workers complete their calculations, they use MPI to send their results back to the master, which integrates them into the final image array. This process continues until all pixel ranges are processed. Within each worker node, OpenMP is used with dynamic scheduling to further divide the workload among the node's threads. Instead of each thread being assigned a fixed set of pixels, threads dynamically pull tasks as they become available, allowing for more efficient use of computational resources. This dynamic scheduling within each worker ensures that even the more complex pixel ranges, which may require more iterations, are handled efficiently, preventing any single thread from becoming a bottleneck. By using MPI for implementing the master-worker model and OpenMP for dynamic scheduling within each worker, we are able to minimize idle time and solve the load imbalance issue, allowing for an efficient computation of the Mandelbrot set.

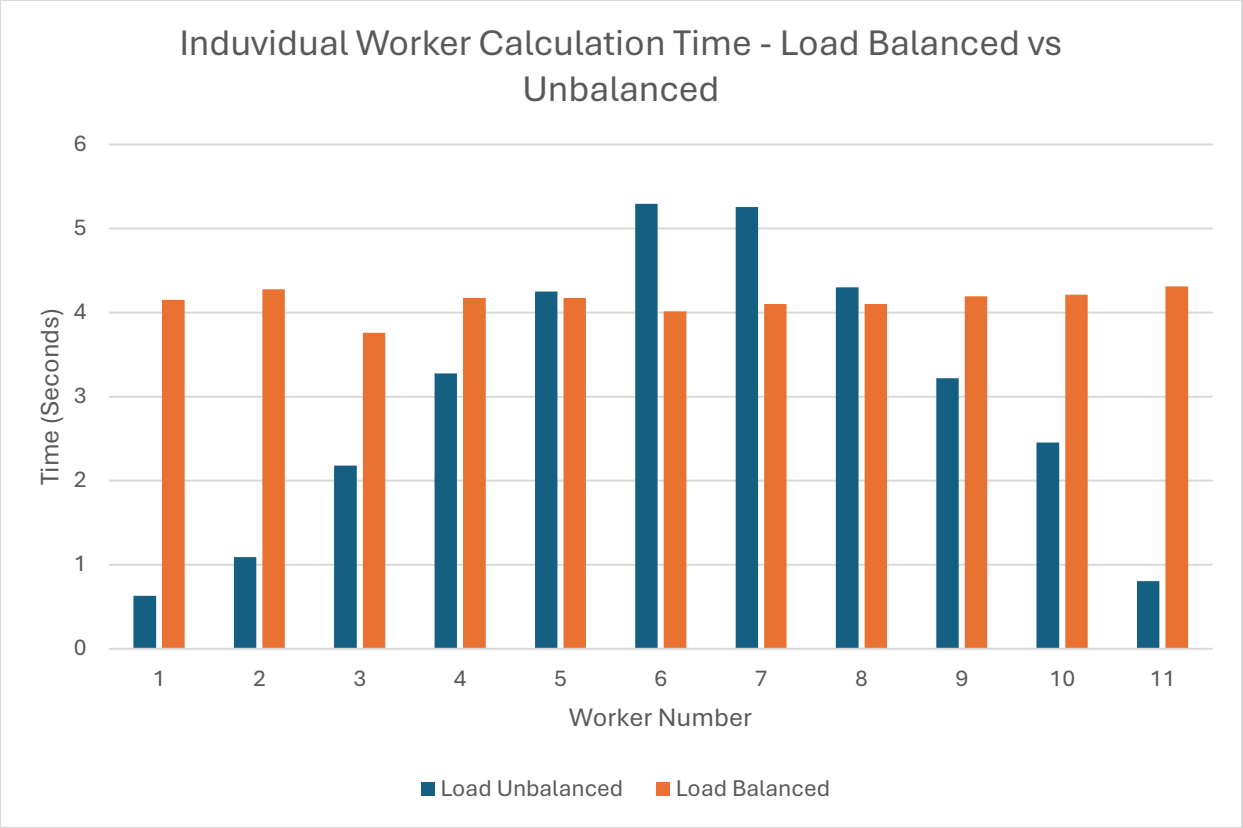

Figure 2 - Worker node computation time comparison

If we take a look at Figure 2, we can see the calculation time difference for each worker when using the load imbalanced implementation vs the load balanced implementation. For the load imbalanced implementation, we gave each node an equal number of pixels to compute all at once, resulting in nodes closer to the center needing more time to finish their work compared to the outside workers. For the load balanced master-worker solution, we repeatedly gave workers pixel ranges to compute until there was no more work to do. With this solution, we can see that each worker needed relatively similar time to compute their workload compared to the load imbalanced implementation. Looking at figure 3, we can see a similar trend with worker threads. If we don't implement dynamic scheduling for our load balanced solution, we can see that some threads take much longer to compute their workload than others, whereas dynamic scheduling really evens out the workload that each thread takes on.

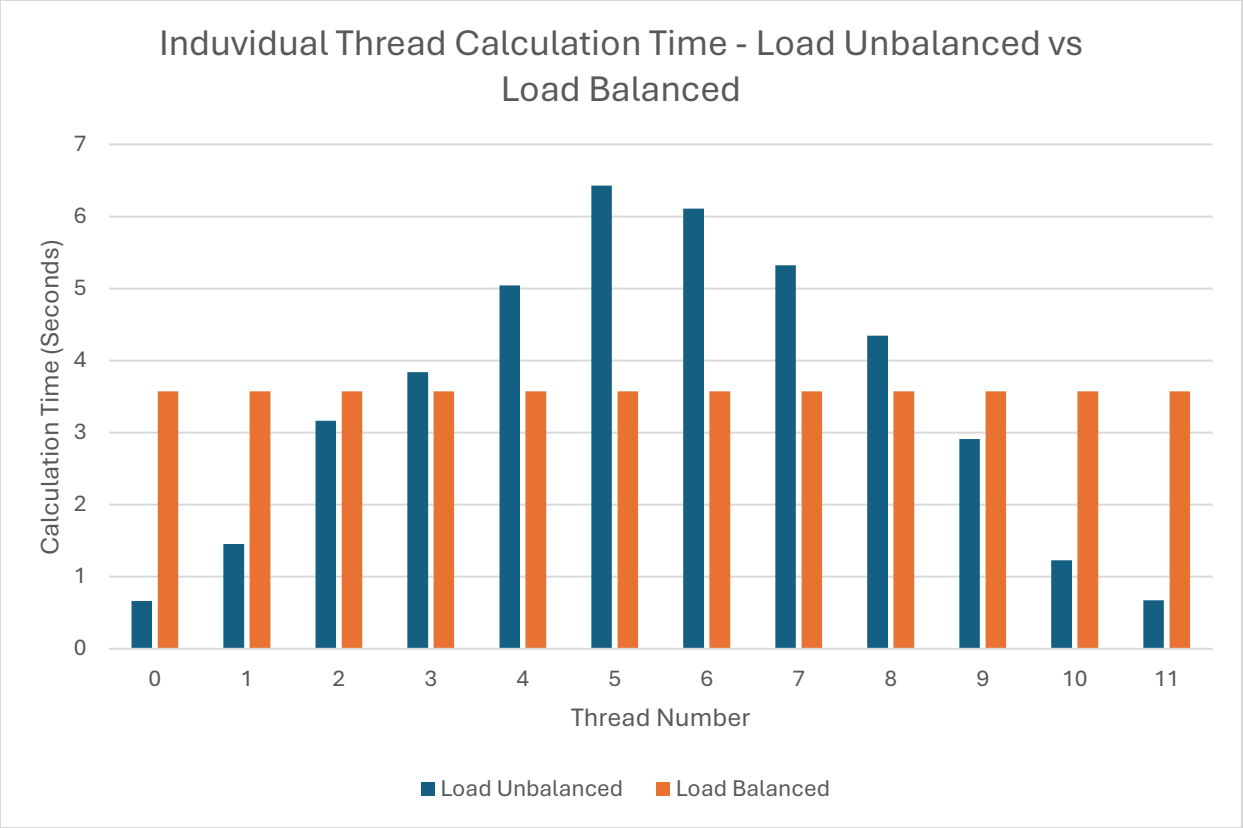

Figure 3 - Individual thread calculation time comparison

In conclusion, solving the Mandelbrot set problem with parallel computing can be tricky due to uneven workloads. The master-worker model, using MPI to manage tasks between different computers and OpenMP with dynamic scheduling to handle tasks within each computer, helps address this issue. MPI allows the master computer to send tasks to other computers, while OpenMP's dynamic scheduling ensures that each computer's threads work efficiently by grabbing new tasks as they become available. This approach balances the workload more evenly, reduces waiting times, and improves overall performance. As a result, it makes computing the Mandelbrot set faster and more efficient, leading to better quality images in the same amount of time.