Kyle Sterling
Prof. Husowitz
Parallel Programming
17 August, 2023

Mandelbrot Set Parallelization

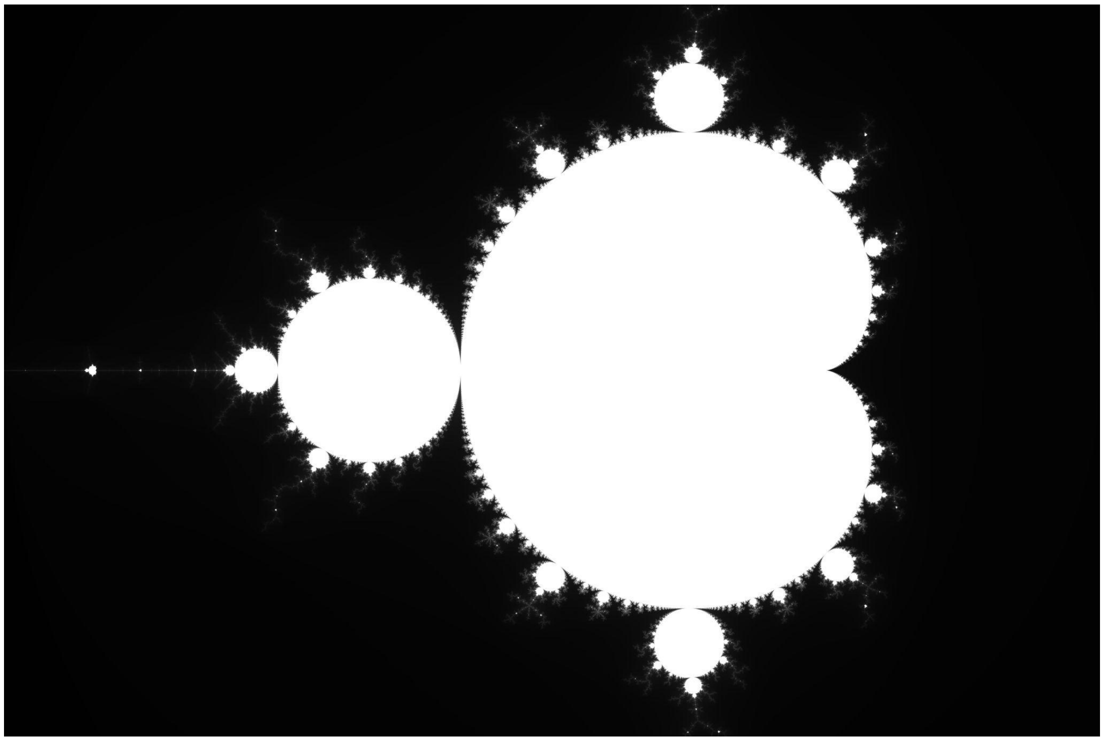
To start off, this assignment simply focused on the parallelization of the mandelbrot set. The mandelbrot set is a large set of complex numbers that together form a fractal structure which is essentially a geometric shape or pattern that repeats itself or appears similar at any magnification. Even as it gets infinitely small or large, the pattern can still be made out. To get these numbers, a set of complex numbers go through multiple iterations to find out if they escape to infinity or not. The numbers that don't escape within a set number of iterations create an image displaying the fractal. At the end of the program, it provides an image from the outside at a specified amount of pixels. With larger resolutions it gives a great depiction of the fractal where the view can be magnified.

The Mandelbrot Set

To dive deeper into the algorithm, it begins with the function $z_1 = z_0^2 + ci$, where z is a real number and c is a complex number. The next step is to iterate through a number of values and plug them into the function. These numbers are in place of z, so they are all real numbers. The mandelbrot set starts with a seed of 0, meaning z_0 will always be 0. While this is happening, ci remains a constant complex value. Notice that z_1 is what the above function is calculating. This z will keep incrementing with the values calculated before it, so if z_0 is 0 and ci is 1 to start, you get 1. Now you do $z_2 = 1^2 + 1= 2$, where that first 1 is the z_1 previously calculated. The next value would come out to 5 if you did another, then after that it would keep getting larger and larger, going on into infinity. This is an example of a complex value that is not in the mandelbrot set, as it eventually escapes into infinity. Using a new example, $z_1 = 0 + -7/4$ will not escape into infinity and is in the mandelbrot set. Upon doing the calculations, it becomes clear that the value always stays within a certain range somewhere around zero. Neither of these examples actually utilize complex numbers, but all values used for ci are within the complex plane. Using a larger number of iterations and increasingly precise values, the mandelbrot set will go deeper and deeper creating the infinitely zooming fractal. Computers have their own limitations, so simple programs like the one used here creates the static image in resolutions somewhere around 48000x32000 maximum, but this starts to take up more time, and images that size also take time to load.

The Set Algorithm

Now for how the program actually does this. The program generates the image in specified dimensions, represented by the integers ny and nx respectively. This image lets you see a mostly unmagnified view of the mandelbrot set, and you can begin to see the patterns just from this outside view. The mandelbrot algorithm begins with two for loops, one nested in the other. These navigate the defined plane, with the first one going from 0 to ny, and the nested loop going from 0 to nx for each value in the first loop. After the loops is a section that calculates the complex values x0 and y0. It does this based on the position in the complex plane using the variables defined xStart, xEnd, yStart, yEnd. With these it can use the current position from the loops to calculate a complex number to use for that position in the final calculation. After this, the program moves on to a while loop where it goes through the iterations to determine whether the complex number is a part of the set or not. First, it sets x, y, and the integer ‘iter’ to 0, then the iteration calculation begins. The loop goes until the iter value hits the iteration limit which is set to 254 in this code. The first line increases the iteration value, to count how many it needs to go through. After that comes the important functions. The first one basically does the $z_1 = z_0^2 + ci$, and then x and y get updated for the next iteration. In the two lines where x and y are updated, y represents the complex value, and x represents the real value. Now that x and y are defined, the program goes through an if statement using x and y that basically checks if the resulting z value exceeds the limit, and will eventually grow to infinity. If the x and y values pass this if statement for every loop until the increment maximum, then the value is a part of the mandelbrot set, and the value gets added into a matrix containing each value of the set. At the end of the program, the matrix is completed for the number of values within the plane, and the matrix gets converted into the final image of the mandelbrot set.



Mandelbrot set image returned by the program.

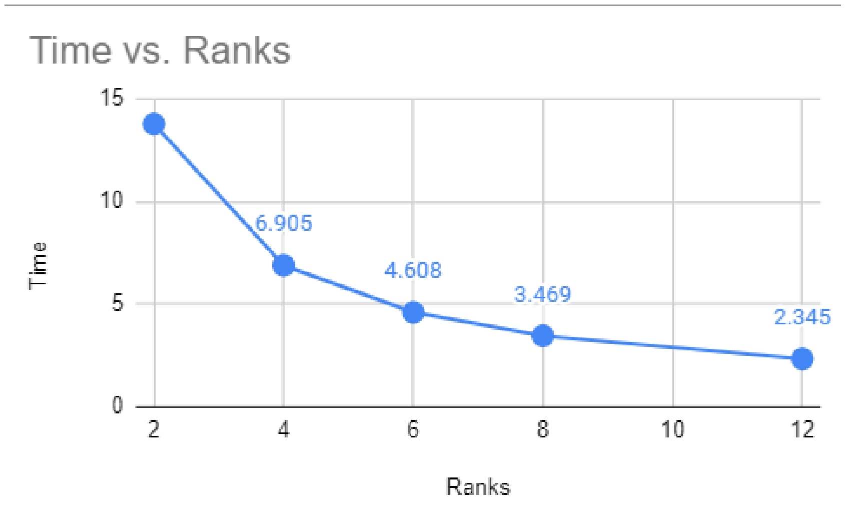
Parallelization and Threads

This program does a lot of calculations, and if it was meant for a larger area it would take a lot of time to complete. This is why the goal was to parallelize the code, in order to speed up the process. This was done using MPI and OpenMP as a way to split up the work and speed up the calculations. The problem with this is that the work done in the mandelbrot set cannot be evenly distributed between the nodes, so another algorithm must be incorporated to try to even out the work.

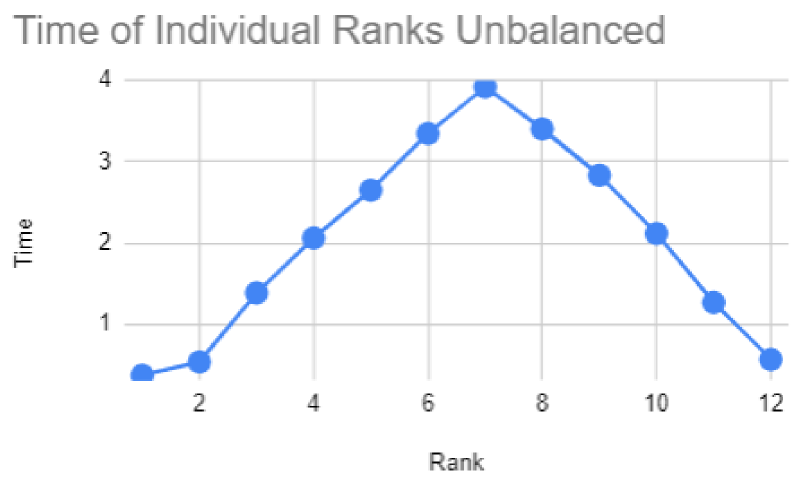
The algorithm uses a master-worker algorithm to try to distribute work more evenly between each node. The main part that gets broken up to achieve this is the ny variable. This represents the rows of the image, so it is broken up into groups that can be specified beforehand. It begins with the master node, who determines the initial number of rows that gets sent to each node. Once this is finished, each node will receive their first set of work, and the worker section begins. The workers execute the code only for the received rows, then completing their work once that section is finished. This section is also where OpenMP does its work. The mandelbrot algorithm section is split up between usually 12 threads. In this program, they work under a schedule system using a dynamic algorithm. This means that as soon as a thread finishes its work, it's given a new set to work through. Once the threads are all finished, the node is done. The worker node sends a value back to the master indicating that the node is complete and needs more work. The master receives the message in a while loop that will continue looping until the flag indicating that all nodes are done is true. After getting the message from the worker node, the master checks that the next set of work is within the range of rows where it will set that node as finished if it isn't. Otherwise, the master node continues on increasing the start and end value for the next set of work then sending this new start and end value to whichever node sent the last message to the master. This then repeats the cycle where the node now has more work, and if the end value sent to the node is less than 0, then the node is considered done and the loop is broken. Once all loops are broken, then the process is finished and the mandelbrot set is complete.

Data and Graphs

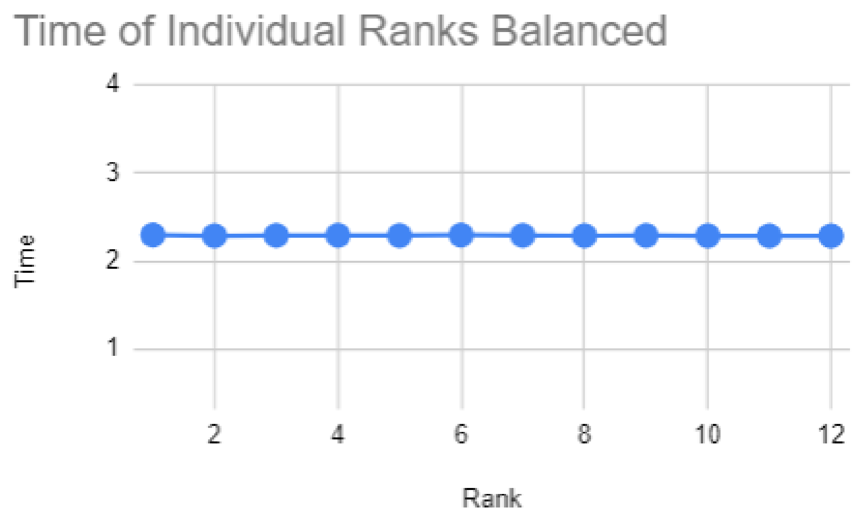
Next is some data collected from the program, starting with total completion time versus number of nodes used.



This first graph represents the amount of time the overall calculations took in seconds versus the number of nodes used. The program completed a 24000x16000 pixel image using chunk sizes of 30 and 12 threads. The time taken gets noticeably smaller as ranks used increases.

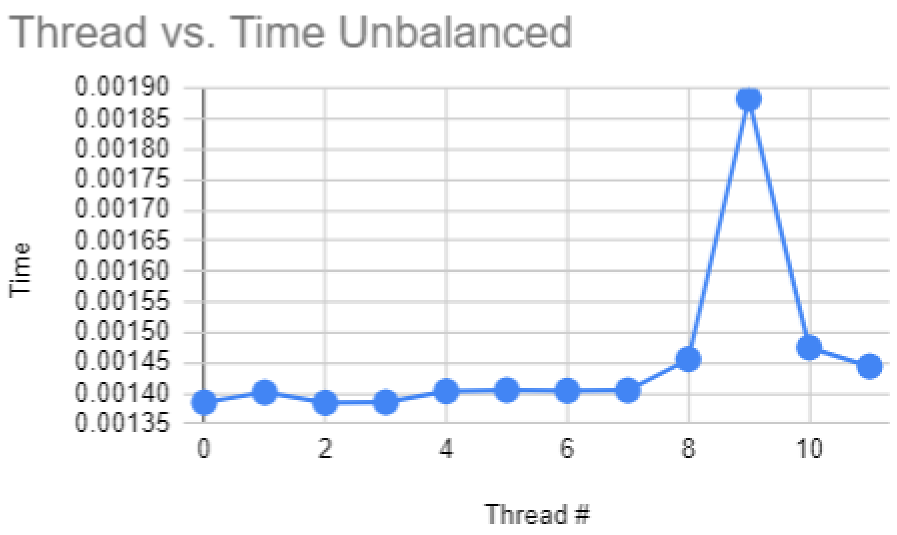


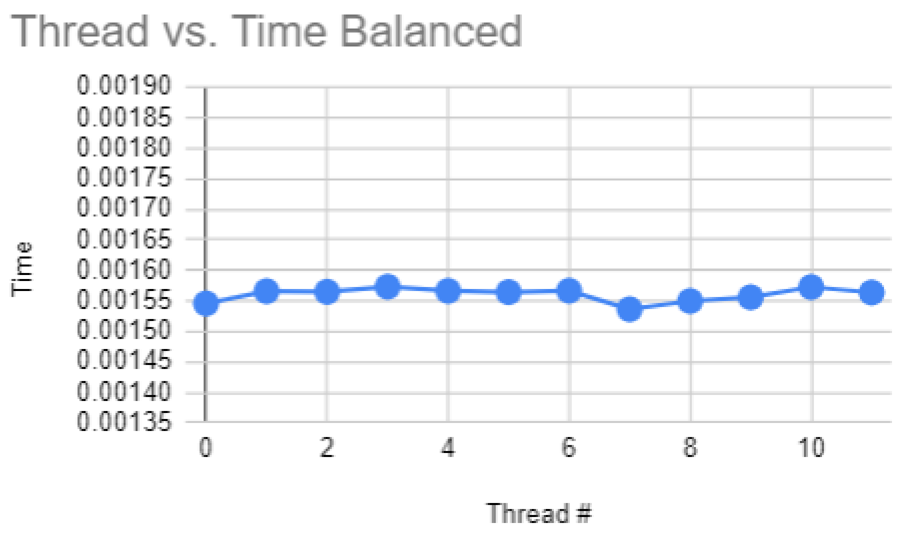
This graph shows the amount of time each individual node took to complete their calculations throughout runtime. This graph represents the unbalanced node times in seconds. These times were also done with 24000x16000 dimensions with a chunk size of ny/numranks where ny is the number of rows and 12 threads. This graph is significant because it shows how much time middle nodes would take compared to end nodes. This is because the mandelbrot set image has a lot more values than a part of the set in the middle, so nodes that receive rows from the center of the image will take much longer to complete.



This graph also represents the amount of time each node took to complete their calculations. This one however is clearly balanced, looking at how even the times are in the end. This also used the 24000x16000 dimensions,chunk size of 30, and 12 threads. By using a master worker algorithm to split up the work, it’s possible to make each node do a very similar amount of work. Comparing this to the last graph shows how large of an improvement can be made, also decreasing total calculation time by about 2 seconds.

These last two graphs instead show thread times on a single node run. Using the same dimensions, 12 nodes, chunk size 120, and 12 threads.





The first graph shows the unbalanced threads, this uses static scheduling and gives each thread the same number of rows to go through. The method of timing only uses a single node runthrough of the calculations, so this only represents a small portion of thread calculations in the whole program, and results can vary. The outlier on thread 9 still gives a good idea of the unbalanced nature. The second graph uses the same slice of the image, but the times are much more balanced. This part uses dynamic scheduling so each thread will get new work as soon as they finish their current set of calculations, meaning their overall workload will be much more balanced.

Conclusion

Thinking through the algorithm and understanding how to best parallelize this code helps give a much better understanding of MPI, OpenMP, and how general parallelization could be done on other projects in order to speed up work time. The mandelbrot set is a good example of a more complex program and thinking through how it works can be beneficial to somebody's algorithm thought processes. Mixing the two methods of OpenMP and MPI together is something common in computing, so the experience of combining their power is valuable. Using them together some good data could be collected to show what each component is doing in the overall program. MPI’s nodes can be used in a master worker fashion that allows the nodes to constantly have work to do while the program is still running. This also ensures that it is being completed as efficiently as possible in terms of time and computer resources. Referencing the graphs shows that there is a big difference in how it works with and without the master worker. Threads in OpenMP maximize that efficiency by allowing each core to have some work at a smaller level. In the scenario of this program, MPI breaks down the rows into pieces, and each piece is broken down into smaller bits for threads to work on. This all comes back together for the final product, which is a potentially very large image done in a relatively short amount of time.