Project #6 Report

Kyle Pontius

Introduction

The value of "big data" in today's world stands out not only as a business buzz world, but as a topic with real application. MPI's ability to scale is one of its greatest assets, making it a robust and effective tool for tackling these big data problems. This project focuses on taking the extremely parallelizable example of the Mandelbrot set and farming it out to any number of nodes to more efficiently complete the set calculation. The program also defines several parameters taken into consideration in the calculation, some of which include the zoom level, the width and height of the image itself, and the work "chunk" size (equivalent to the number of columns). The goal of the program is to leverage the master-slave paradigm to send out work on demand as particular nodes finish.

Approach

My approach to calculating the Mandelbrot set was to: First, send out work requests to as many nodes as are available until the either number of available chunks is reached or the number of available nodes is reached. I perform this operation using a non-blocking send (***MPI_Isend()***) from the master node request, so all the nodes can start working as soon as possible. All work requests are sent using this same non-blocking function call. In the case where the number of chunks is reached first, and there are unused nodes leftover, then the program immediately terminates those unused nodes to prevent the program from hanging while they wait for work.

Next, after this setup phase, there is a while loop that contains most of the execution time. For slave nodes, each iteration consists of blocking to receive work information, performing the appropriate calculations, then sending in that information back to the master node along with getting the current status of the system. If there are more work chunks available to do, that node loops back to receive more work and start over. Otherwise, it exits immediately.

For the master node, it blocks waiting for nodes to finish their calculations, until all have been received. When a given calculation is received, the master node, which contains one large image, takes the portion of the total calculation performed by the slave node and copies the value at a given position into the larger image. It was a bit of a challenge to figure out how to best send these subsets of the whole problem back and forth, but I determined that the most effective approach was to determine the size of data based on the proportion of the chunk size. For example, when the chunk size was 20% of the total image size, then the program calculated a container for the subset calculations that was 20% of the original image size. This prevented

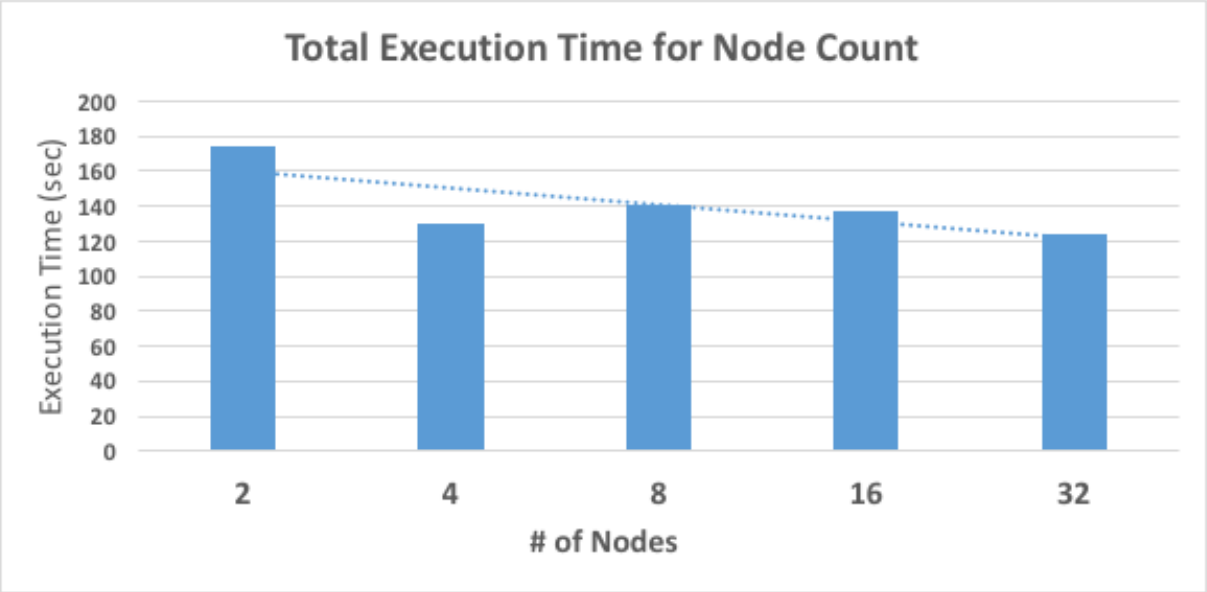
us from sending back the whole image back and forth between nodes and only made it necessary to send a small portion instead. This is analogous to passing a short note to someone using a strip of paper, instead of the whole 8.5” x 11” page.

Once all of the subsets had returned from the slave nodes, the master node would end up with a complete image of the entire Mandelbrot set. The copying portion of the algorithm was by far the most expensive part of the operation, primarily because this is a strictly serial portion. As a result, no matter if we have 5 nodes or 50 nodes, all nodes must wait on the master node to copy the values from the slave nodes into the master image. This results in a large queue of slave nodes waiting to send their data to the master node, slowing down the execution process. In another section labeled “Alternative Approaches” I touch on an idea I had to potential decrease time spent waiting on the master node to complete this expensive copy operation.

Finally, as per the spec, I’d like to touch on the potential difference between just MPI, and OMP with MPI. MPI excels as distributing large problems across many nodes to complete portions in tandem. The challenge with MPI is that the problem themselves have to be able to broken up into large chunks before being distributed. OMP, on the other hand, excels at taking a repeating calculation, such as the Mandelbrot calculation, and parallelizing just that portion of the execution time. From what I’ve seen in completing this project, the real hold up is the serial portion on the master node there the data is copied into the master array. Loops such as these drain the performance out of parallel algorithms, so it seems like a natural choice to implement OMP here to break up the oft-repeated copying portion of the code.

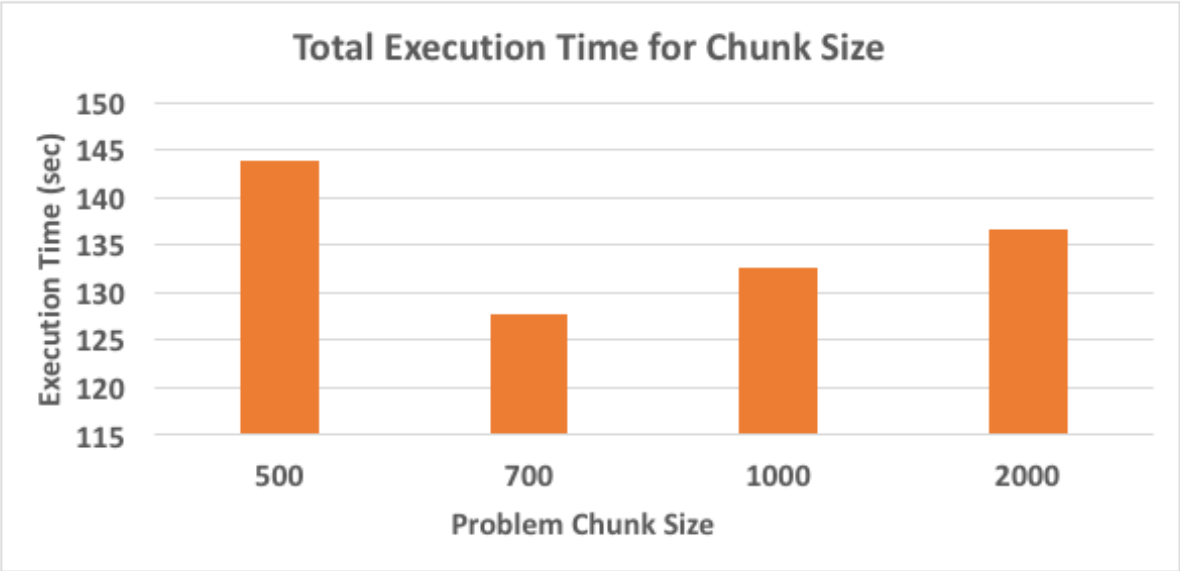
Experimental Setup

In order to focus on the comparison in execution time between different work chunk sizes, I kept a consist number of nodes and total width and height for the program. The number of nodes for this comparison was always 32, and the width/height for the program was always 28,000 pixels. I then ran my program in chunks of 500, 700, 1000, and 2000, comparing the results in the graph below. The nodes themselves were the CS department’s open lab computers networked using MPI. To reduce noise and outliers from this data set, I ran this calculation twice for each chunk size. For the second graph, I kept the chunk size at 700 (as it was the most optimum from the first round of results) and set the number of nodes to 2, 4, 8, 16, and 32 to show the difference in execution time between the different # of nodes.



Results

The results were really quite interesting. I expected there to be significantly more speedup from the parallelization of the algorithm. As can be seen from blue graph above, the execution time for the algorithm did have about 30% reduction in total execution time. I attribute any lack of speedup largely to the significant dependence of the algorithm on the serial portion of the code, where the results from the subsets of the Mandelbrot set were copied byte-by-byte into the master image on the root node. With this in mind, the results seem less surprising. With MPI, as I've mentioned previously, we can farm out any number of calculations to our child nodes, but as long as we're copying information like we did, the child nodes will simply queue up and be forced to wait. In reviewing ways to improve the algorithm, I touch on a few approaches that may be more effective in my "Alternative Approaches" section below.



For this graph above, it was very interesting that there's a clear leader in the execution times. The chunk size of 700 columns performed by far the best in this algorithm. On a high level, this means that this particular chunk size most efficiently balanced the proportion of the time spent in the slave node with the time spent in the master node. In an attempt to get the big picture, I scaled up to as many columns as it would allow me to transport while still being divisible by the problem size. This limited was caused, as I've mentioned before, to the message buffer size limit in our configuration of MPI. The maximum size it could send turned out to be 2,000 columns. It appears that as we grow to this higher number of columns the execution time steadily climbed as well. I would be very interested to know if this trend continued as the message size got larger (especially since my execution time in the root node dominated the overall completion time), potentially forcing the slave nodes to spend more time in their calculations. This, in turn, could reduce the queue size and ultimately distribute the workload more evenly.

Alternative Approaches

I don't normally include a section on alternative approaches, but there were a couple changes to the algorithm I would've liked to do, given the time. First, my implementation focused on the master-slave paradigm, which sends out work based on the availability of any given node. The result of this operation was that each and every node dumped its results on the root node when finished, then the root node added the newly calculated values to the master image. An interesting alternative would be to have the non-root nodes communicate with each other to create a larger piece of the overall problem, which is then sent to the root node. While this would create more communication ***between*** non-root nodes, it seems like it would reduce the amount of traffic required with the root node, as we can send larger blocks

with each communication. Additionally, you could build a hypercube centered algorithm with this larger buffer size. This may increase the power of our algorithm as well.

Second, it would've been interesting to chart the differences in speed when variables other than chunk size were changed. For example, does changing the zoom level, or hue per iteration, or center x/y affect the total execution time? Part of the purpose of the master-slave paradigm is that the work for each node is not the same, thus it seems reasonable that the results could potentially be affected in any one of these scenarios.

Third, with a different version of my implementation that used larger message sizes to pass intermediate information, I ran into the buffer size limit in our configuration for MPI on the lab machines. It would've been very interesting to modify the message size and see what the optimum buffer size (and the related chunk size) is, as well. If we could could send larger messages back and forth, I believe we could more effectively distribute workload across machines.

Creative Contribution

As I touched on briefly before, I used non-blocking ***MPI_Isend()*** function calls. Anytime work was sent to the slave nodes, this was my approach. The rationale behind this was to free up the root node as soon as possible to allow it to return to the task of copying results into the master image. It would be very interesting to see if/how we could somehow use non-blocking ***MPI_Irecv()*** calls in our implementation as well.

Conclusions

A few key conclusions from this report were: First, while parallelism is a powerful tool, careful planning is clearly necessary. My results turned out to produce only mediocre increase in performance for the given problem sizes. Second, according to many of theories on parallelism we covered in class this semester, the serial portion of the code is by far the limiting factor. This is evident in this case, since the serial portion of the code (the root node's copying loop) decreased the performance of this system considerably. Finally, there are many powerful and exciting alternatives to the algorithm we built here. It would be exciting to the observe this difference in result, given the time.