Sam Tsoi; 913032178
Franklin Shih; 912667159

ECS 158 Project 3 Report

Program #2: Matrix-multiplication with message passing

The implementation of first program is similar to first project. After error checking of parameters, and declaring the necessary variables, we called *MPI_Init()* to start using the MPI APIs. In our program, *comm_rank* will indicate if the thread is master or workers; it will operate the assigned job accordingly. In master thread, we initialized *A B and C* and assignment equal number of *numRows* for each workers to calculate. Additionally, we assigned additional rows to the workers using *extraRows* to keep track of them. If there are any extra rows that are leftover, extraRow is used to keep track of these rows and we find this by using modulo. Then, we assign these extra rows to each of the workers using a for loop, by adding one more row each time there is an extra row for each of the threads. With all the set up complete, master thread will assign necessary variables to each workers to calculate; more specifically, master thread sent *offset, numRows, A[offset*N] and* everything in *B* to the workers. We did so in the for loop for both assigning variables and receiving variables. However, we noticed that master and worker threads can finish at different time, so we synchronize the threads to make sure master thread will finish last to get consistent result. We used *MPI_Barrier()* to accomplish the synchronization task and it gave us a consistent result. From the perspective of worker threads, it will receive the necessary variables from master and complete calculation. *MPI_Barrier()* is used when calculation is complete and wait for other threads to complete as well.

MPI version for various matrix orders and in different configurations:

Type \ N	50	200	500	1000	2000
2	0.000270 secs	0.004338 secs	0.078546 secs	0.611026 secs	4.717689 secs
4	0.000231 secs	0.003309 secs	0.036715 secs	0.363520 secs	2.869768 secs
8	0.000405 secs	0.002548 secs	0.051601 secs	0.379298 secs	2.963445 secs
Average	**0.000302 secs**	**0.003398 secs**	**0.055620 secs**	**0.451281 secs**	**3.516967 secs**

OpenMP version from Project 2 for various matrix orders and in different configurations:

Type \ N	50	200	500	1000	2000
2	0.517342 secs	0.828791 secs	0.493380 secs	0.363763 secs	2.957880 secs
4	0.939499 secs	0.838022 secs	0.313429 secs	0.246216 secs	1.429562 secs
8	0.345440 secs	0.891644 secs	0.251323 secs	0.174395 secs	1.448225 secs
Average	**0.600760 secs**	**0.852819 secs**	**0.352710 secs**	**0.2614581secs**	**1.945222 secs**

As evident above, the MPI version for various matrix orders, on average, performs a lot better for lower matrix orders, such as N=50,200, and 500. However, it seems like the OpenMP

version from Project 2 seems to do a lot better, on average, when N=1000 and N=2000. We see that 4 workers is optimal for our program in this case and for our number of cores. This is due to the time it would take to send all the information to the workers, receive, and send it back. We see that this is the case for when we spawn 8 workers for the MPI version. When we compare 4 workers for N=50 in the MPI version to the corresponding result in the OpenMP version from Project 2, we can see that the MPI version is much, much faster. This result is similar for N=200 and N=500 as well. With more cores, the performance does not necessarily get better, since many cores requires the balance of the usage of the resources and a lot of overhead. However, as the matrix order increases, the OpenMP version from Project 2 seem to outperform the corresponding results in the MPI version. An advantage of OpenMP is the simplicity of the implementation. Our program to matrix multiplication is not too complex, and OpenMP will be able to do the work efficiently. So, it seems like OpenMP probably does better with greater matrix order. However, it seems like MPI works really well when the matrix order is lower.

We found that running on the local computer is faster for smaller calculations, but slower for larger calculations. We suspect that this is because once the parameter surpass a certain threshold, local computer does not have as many core/computation power as the remote computer. By distributing smaller tasks to remote computer and have them send back the result, the performance will increase compared to computing everything in a local computer.

Program #2: Mandelbrot set

First, we read in each argument and we implemented error checking. (i.e. If the program does not receive 4 arguments, if the program's x center and y center are not between -10.0 and 10 or the zoom level is not between 0 and 100) After error checking, we do some initial calculations to prepare to calculate each element in the matrix. We calculated the x_min and y_min in order to find where the complex number c starts for matrix at the starting position (i.e. M[0]). We found that in order for our matrix to match his checksum, x_min and y_min started from the bottom left of the 1024x1024 box from the prompt.

In order to find x_min and y_min, we received x_center from the argument after error checking and subtracted from the distance we calculated multiplied by half the size of the box (which is 1024 in the case). These values are where x_curr and y_curr start, which are used to calculate each c at each point. We go into two double for loops that iterates through the matrix M. We implemented another for loop for each element that goes up to the number of iterations at cutoff. If the magnitude at each iteration is less than 2, we are going to keep going until it is not less than 2. In this case, we know that at this iteration, it diverges and that it is part of the mandelbrot set. We assign this to the element in M. Each element will reflect the iterations and indicate whether each point converges or diverges, and when.

We implemented MPI in this program similar to the first program. First, we initialize MPI with MPI_Init(), get the number of processes in the communicator with MPI_Comm_size(), and the rank of the calling process with MPI_Comm_rank(). In the master thread, which is when the rank of the calling process is 0, we did some error checking. Similar to the first program, we initialized numRows for each workers to calculate. If there are any extra rows that are leftover, extraRow is used to keep track of these rows and we find this by using modulo. Then, we assign these extra rows to each of the workers using a for loop, by adding one more row each time there is an extra row for each of the threads. Now, the master thread will send all the necessary variables for each workers to calculate. This is done in a for loop for each worker for both

sending and receiving. Similar to the first program, since the different workers work at different times, we use MPI_Barrier to make sure the workers are synchronized and the results are consistent.

Changing the zoom level, center (-1, -0.2) and cutoff at 256

	zoom of 9	zoom of 100	zoom of 1
2	0.770250 secs	2.484164 secs	0.016742 secs
4	0.594309 secs	0.861788 secs	0.011438 secs
8	0.560439 secs	0.751099 secs	0.005740 secs
Average	**0.641666 secs**	**1.36568 secs**	**0.011306 secs**

Changing the cutoff, center (-1, -0.2) and zoom of 9

	cutoff at 256	cutoff at 10	cutoff at 800
2	0.770250 secs	0.066608 secs	2.239766 secs
4	0.594309 secs	0.034793 secs	1.750437 secs
8	0.560439 secs	0.028216 secs	1.761891 secs
Average	**0.641666 secs**	**0.043205 secs**	**1.917364 secs**

As seen above, changing the parameters makes a difference in the performance. When the cutoff is smaller, the program as a whole has less loops to go through to check whether a value converges or diverges. So, when the cutoff is large, it takes more time, on average, and if the cutoff is small, it takes less time, on average. As we increase the zoom level, the more computationally expensive the calculation would be. When the zoom level is higher, the less distance there is between each point and the more precise between each point in the complex plane would be. So, when the zoom level is small, the performance is better. We notice that when the zoom level is large, using less number of threads significantly worsens the performance. Using 2 threads when zoom level is 100 takes 2.484164 secs, whereas it takes 4 threads 0.861788 secs. This may indicate that threads play a big role in increasing performance when there is a larger zoom.

Similar to matrix checksum program, we found that running on a remote computer is much faster for this program. We suspect that this is because the remote computer is able to perform better when the program is more computationally expensive as there are more cores in the remote computer. By distributing smaller tasks to remote computer and have them send back the result, the performance will increase compared to computing everything in a local computer.