**Non-Blocking MPI**

**Final Project**

**CPEN/CPSC 435**

**Group 4:**

# Introduction:

# Parallel computing allows for partitioning and disturbing data sets to multiple processes, allowing for increased processing performance. Compared to a traditional single program, single process platform a single program multiple process reduces latency wait times. Instead of having to wait for sequential computations to finish, independent functions can run concurrently. Using a master slave model, this project will implement a message passing interface utilizing a non-blocking send and receive structure. The algorithm is designed to compute matrix multiplication and record the processing time of the master. The time data recorded will be compared with a similar matrix multiplication program utilizing a blocking MPI send and receive. Using a dynamically sized matrix, time data is recorded for analyzed determining the performance of the algorithm reflective of changing computational sizes.

# Implementation:

We first sent all of Matrix B from the Master node to all Slave nodes using MPI\_Send and MPI\_Irecv. Next, we defined a loop for the Slaves in which they continually check to see if work is available for them. This is handled by sending an MPI\_Request to the Master and receiving a confirmation or a kill signal if no work is available. If rows from Matrix A are received via MPI\_Irecv, then the matrix multiplication is called on the supplied rows and the results are then sent back to the Master using MPI\_Isend.

On the Master’s end, there is a loop that continually checks for a request from any node. If it receives a status indicating a Slave is idle, it sends back a set number of rows from Matrix A. Otherwise, if it receives a status indicating work is complete, it saves the results to Matrix C. Results are saved via MPI\_Recv, and rows are sent via MPI\_Send.

# Performance Analysis:

* Fixed matrix of size 512 comparing, non-blocking verses blocking results:

Analysis of the non-blocking MPI programs shows a slightly longer processing time as compared to the blocking program created for assignment #6. As seen in the comparative graphs there is a roughly 0.2-0.3 second delay in performance obtained by the non-blocking program.

* Adjustments to matrix size with set Node: 4, Rows Per Task: 2:

With a set number of nodes and rows per task the results displayed increase in time with the growing size of the matrix computation. These results explain the effects created with a blocking master and non-blocking slave program. As the graph indicates there is an exponential growth reflective to increased matrix size. These results are accurate to the computational time delays with increased data sizes.

# Discussion and Conclusion:

This project was not without its difficulties, one of the more significate troubleshooting scenarios arouse with configuring the logic for the slave function when it had finished with the current task. This disconnect was remedied with the use of MPI\_Probe in the master function. Probing allows the program to check if the slave was returning data, in turn then being ready for a new partitioned data set, if remaining date was available. Another question that came to light in the process of developing this project was, what was the correct number to divided matrix A partitions into for the most efficient output. The concern was with proper load balancing, when sending large numbers of matrix A rows to the slave, there was a potential difference in process return times per node. Although thought to be nominal, our concern was with the scalability of the project. With increased partitioned sizes would the load balancing discontinuities be more apparent? With testing, the resulted showed to be nominal as previously hypothesized. When looking at the difference between blocking and Non-blocking send and receive it is important to properly include to MPI\_Wait, as with the increased scale of the matrixes there will be delays in processing times. By using properly placed waits the program will guarantee the processes all have been received prior to continuing, in the effort to eliminate false data.

In conclusion, when using data sets as exampled in this analysis, the superior performance was achieved with the use of the blocking MPI program. With 2 nodes there was the greatest performance disparity between non-blocking and blocking, a near different of 0.45th of second. This size of difference in performance would have a detrimental impact on a program running multiple sequential matrix multiplication processes. When choosing a program style, the choice for a blocking algorithm when doing matrix multiplication would be simpler to implement. Coupled with the performance results and the simpler implementation blocking MPI is in conclusion best choice.

# Contributions:

* Mathew Lee: Mathew was the lead developer on this project, as he is the strongest of the team using the MPI process. He contributed in developing the logical structure for this project. He collaboratively troubleshot ideas and concerns, drawing out logic using the whiteboard. He openly discussed his thoughts with the team as we worked through the complication that arose.
* Trevor Greenside: Trevor coordinated the lines of communications, he set up the GitHub for this project, with each file he had created working templates for all the necessary MPI initialized headers and body code. With each templates preset we as a team could focus just on placing the logic together inside the bodies. This proved to be a valuable group time saver and sped up the logical development process significantly. Trevor also created the script file again creating a more efficient project.
* Thomas Hughes: Thomas’ contribution to the project came from the data analysis, most of the project write up, presentation power point slide. He assisted in verifying and validating the matrix multiplication implementation, running multiple tests and sharing the data for presenting.