Lab 3 – Multi Process Matrix Multiply

Draven Schilling

10-2-19

CS 3841

Introduction:

In this lab, we created a programmatic representation of a standard math matrix. We were then able to take two matrixes and multiply them together to get the resulting matrix. Matrix multiplication is required to use multiprocessing in order to increase the speed of execution and processing.

Design:

For my design, I wanted a clean interface for creating, storing, manipulating, and destructing matrixes. So I decided to create a matrix struct containing the dimensions of the matrix along with an integer pointer of the cell data (in order). The next logical thing to do is create a method that will create a matrix and populate it's data with a file's information so I then created a "createMatrix" method which takes a file name and returns a pointer to a malloc'ed matrix struct that's populated with that files data.

After being able to create the matrix I thought it would be useful to create a method that would print the matrix to the console so that I could verify that matrix creation was functionally correct. So then I created "printMatrix" which did just that, print the matrix dimensions followed by a visual representation of the data.

Next, I knew I needed to be able to free the memory allocated to the matrix structs so I created a "deleteMatrix" method used for deleting a matrix after it's use had run out.

Finally, it was time to create the multiply matrix method, I decided to not start with the non-multiprocessing solution because I was short on time and I was confident in my ability to create and debug a multiprocessing solution to start. With that in mind, I started my multiply matrix method by creating a new result matrix of the correct size and creating the pids for each child process. Next I created a loop to go thought each of the pids and execute a child process subroutine. The child subroutine would take each matrix along with a designated cell to calculate and perform pointer arithmetic to offset and perform the calculation. The child would then pipe the result back to the parent and destruct itself. It was at this point I realized I needed to create another struct type to store the child's results before piping it back to the parent. The parent doesn't know which processes finish in which order so it wouldn't know which results to put in which result cells. So the new struct would hold a cell number and it's associated result. After piping this result back to the parent, the parent would collect all the child process data and fill the result matrix. The extra malloc'ed memory world be then freed and the result matrix returned.

Build Instructions:

1. Execute make to build the makefile
2. Run $./test matx maty (where matx is a standard matrix file and maty is another standard matrix file.
3. The results of the program will print matrix x followed by matrix y and finally the result matrix.

Analysis:

Based on the timing results, the multiprocessing solution is significantly slower than the non-multiprocessing solution. I believe this is because the overhead for creating multiple processes is not worth it for calculating the result of one cell (unless the input matrixes are very large). A better way would be to use shared memory such that you don't need to create the overhead for a pipe. Not sure how much of a performance increase it would be, but it would help a bit.

If a matrix contains an entire row or column of zeros the resulting multiplication element is zero. How could we optimize the number of processes needed if this happens a lot in our input data?

- We could detect when this is the case in an input matrix and skip creating forks for those processes and jump to filling the result matrix with zero's.

Conclusion:

The most challenging part of this lab was forking n number of times and debugging how to properly use and destroy a child process. It took me not very long to create and print matrix's, and the calculation for matrix multiplication took a bit of thinking. The parts that took the longest were figuring out how to send a result struct though a pipe as well as destructing a child once it was done.

I like the refresher on reading from files and that this lab was more open ended then the first (such that we were not given and restricted to a provided header file). I liked the idea of multiprocessing, but I feel like the implementation was not doing it justice such as its effects not being that substantial.