Christopher Lewis

PA4: Matrix Multiplication

Due: 5/4/2017 - <u>Late</u>

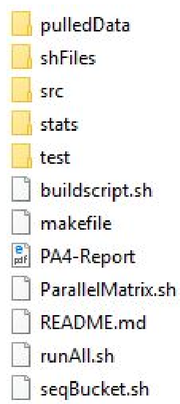
Table of Contents

Overview:

Project 4 requires that I look at matrix multiplication both sequentially and how it looks when parallelized. This is important because matrix multiplication is a N^3 operation so over time the runtime will increase very fast when ran sequentially. Below is a thorough analysis of the sequential and parallel portions of this project.

Project Layout:

Below is the structure of the project.



Inside **build** you will copy a script to the directory that will be called using **./RunAll** below is a sample of what will be found inside the script RunAll which will sbatch the parallel and sequential scripts.

Run All Script

```
#!/bin/bash

BATCHFILES="../build"

for (( a=120; a<=1500000000; a+=120 ))
do

    TEST=$(squeue -o"%.18i %.9P %.20j %.20u %.8T %.10M %.9l %.6D %R")

    while [[ "$TEST" =~ "christopherlewis" ]]
    do
        sleep 1s
        TEST=$(squeue -o"%.18i %.9P %.20j %.20u %.8T %.10M %.9l %.6D %R")
    done

    echo $a
    sbatch par4Cores.sh $a
    squeue
    sleep 1s
done
```

Sequential Script

```
#!/bin/bash
#SBATCH -N1
#SBATCH -n1
#SBATCH --time=00:50:00
#SBATCH --mail-user=christopherlewis@nevada.unr.edu
for (( a=120; a<=5000; a+=120 ))
do
    srun seqMatrix $a seqMatrixTimed.txt
done
```

Parallel Script(s) (Will add later)

Below are two scripts for parallel one is to read in from a file and the other is to no read in from a file.

File IO

```
#!/bin/bash
#SBATCH -N2
#SBATCH -n9
#SBATCH --time=00:01:00
#SBATCH --mail-user=christopherlewis@nevada.unr.edu
srun parMatrix 12 matrixA.txt matrixB.txt >> IOTimed.txt
```

No file IO

```
#!/bin/bash
#SBATCH -N1
#SBATCH -n4
#SBATCH --time=00:30:00
#SBATCH --mail-user=christopherlewis@nevada.unr.edu
srun parMatrix $1 >> 4CoresTimed.txt
```

Sequential Matrix Multiplication

Sequential matrix multiplication is an operation that starts with three dynamically allocated two dimensional arrays of the same size (a, b, and c) and will multiply matrix a with matrix b and will place the result in c. The runtime for each operation is displayed below in table 1 and the graph is displayed in figure 1.

120	0.00568
240	0.027725
360	0.172856
480	0.312501
600	1.442827
720	3.049201
840	6.532457
960	10.94245
1080	16.80198
1200	24.30845
1320	36.26885
1440	44.92302
1560	57.53735
1680	73.54014
1800	91.29215
1920	111.6497
2040	134.2156
2160	161.8084
2280	192.8555
2400	228.7546
2520	266.5984

Table 1: In the table above, right column is the time and the left is the matrix size.

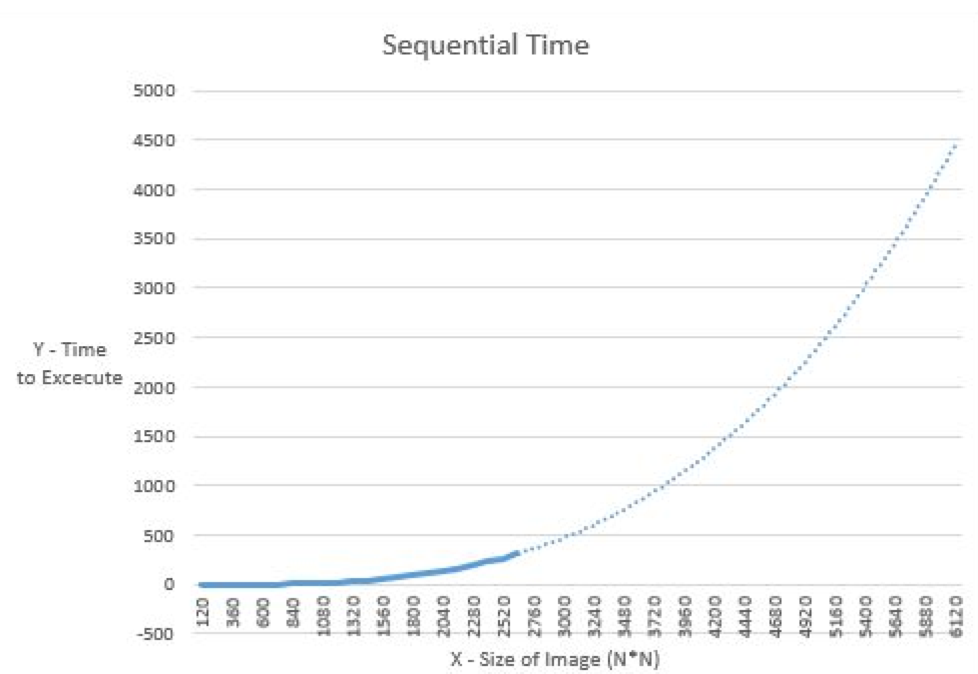


Figure 1: In the graph above, the X-Axis is the size of the data set and the Y-Axis is the time taken to execute. As the numbers get larger, so does the execution time in an N^3 fashion.

Parallel Matrix Multiplication

Parallel matrix multiplication is a little different from sequential. The parallel program will treat each processor as its own group of cells. Within that group of cells it will do the same process as sequential for matrix multiplication but before and e=after there is a slight change to how it is done and that is using cannons algorithm.

Cannons Algorithm (An outline)

row i of matrix a is circularly shifted by i elements to the left.
col j of matrix b is circularly shifted by j elements up.

Repeat n times:
p[i][j] multiplies its two entries and adds to running total.
circular shift each row of a 1 element left
circular shift each col of b 1 element up

Functions Used for Cannon's Algorithm:

Below is the prototypes of functions that will be used to complete Cannon's Algorithm in an efficient manner.

Makes a whole shift left using a send replace functions and uses position to find it out

*void shiftLeft(int *matA, int size, int myProcessor, int numProcessors);*

Makes a whole shift up using a send replace functions and uses position to find it out

*void shiftUp(int *matB, int size, int myProcessor, int numProcessors);*

Finds the id of the processor that is to the left of the current processor (myProcessor)

int getIdLeft(int myProcessor, int numProcessors);

Finds the id of the processor that is above the current processor (myProcessor)

int getIdUp(int myProcessor, int numProcessors);

Finds the id of the processor that is to the right of the current processor (myProcessor)

int getIdRight(int myProcessor, int numProcessors);

Finds the id of the processor that is below the current processor (myProcessor)

int getIdDown(int myProcessor, int numProcessors);

Running Matrix Multiplication in Parallel:

When Running the N-Cubed Algorithm in parallel the results were not surprising since the times should be significantly faster than sequential. The time difference wasn't extraordinary until the matrices got large. Below in Table 2 is a chart of the timing data.

	1	4	9	16
120	0.00568	0.003287	0.00347	0.002149
240	0.027725	0.011488	0.008023	0.005937
360	0.172856	0.025983	0.024481	0.011143
480	0.312501	0.058708	0.04223	0.024034
600	1.442827	0.121096	0.078649	0.044788
720	3.049201	0.370554	0.121475	0.068853
840	6.532457	0.447835	0.233694	0.114718
960	10.94245	0.667046	0.414298	0.164118
1080	16.80198	1.33654	0.635659	0.227855
1200	24.30845	2.93289	0.626435	0.318077
1320	36.26885	4.03392	1.03498	0.427834
1440	44.92302	6.18912	1.14496	0.834744
1560	57.53735	9.39915	1.63868	0.694643
1680	73.54014	13.3093	2.59202	1.03507
1800	91.29215	16.9694	4.68184	1.33403
1920	111.6497	22.3763	5.65852	1.55246
2040	134.2156	72.8219	7.81143	4.08309
2160	161.8084	36.6655	9.72191	2.91686
2280	192.8555	42.9863	12.9557	3.5877
2400	228.7546	49.7794	16.3795	6.21412
2520	266.5984	59.6934	21.0928	6.5802
2640	316.2932	75.6682	25.1479	8.39685
2760		79.6308	30.6232	11.3424
2880		94.6961	36.6024	13.389
3000		107.218	42.1941	16.2732
3120		121.086	47.9459	19.9749
3240		133.538	55.4333	24.3433
3360		147.679	62.2927	28.7896
3480		167.039	69.7187	31.8812
3600		183.352	78.6876	35.911
3720		206.233	86.669	41.1639
3840		231.018	97.668	47.1234
3960		287.189	114.234	52.2804
4080		273.527	111.701	150.306
4200		301.241	130.308	63.6696
4320		327.321	141.864	71.2124
4440			154.423	77.9532
4560			161.286	85.0375
4680			181.015	92.8951
4800			196.689	101.653
4920			213.71	109.116
5040			228.96	118.283
5160			247.555	128.119
5280			268.489	148.197
5400			283.991	149.719
5520			305.28	161.395

Table 2: In the table above, The far left column is the matrix size (N*N) and the top row is the number of cores. The other cells are the times taken to finish.

After the data was put into a chart to get a look at how the timing looked in a 3-D surface the results were very neat since the execution limit of 300 seconds (5 Minutes) with more processors yielded a larger matrix size. Below in Figure 2 shows a 3-D surface of the timing.

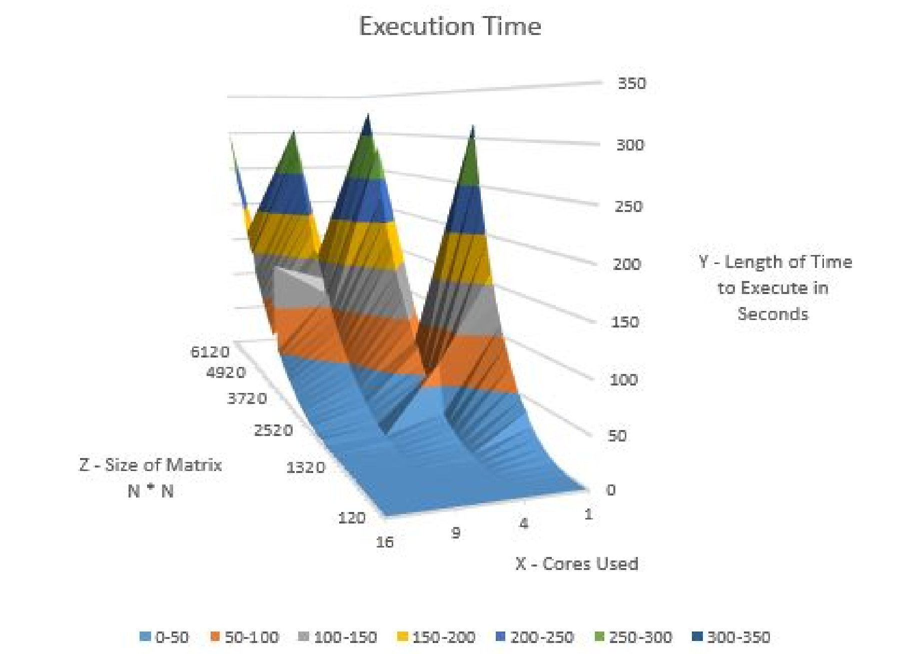


Figure 2: In the graph above, the X-Axis is the number of cores used, the Y Axis is the execution time in seconds, and the Z Axis is the size of the matrix N * N.

Speed Up

After the parallel portion of the program was complete the next piece of information that is important to look at is speed up. Speed up is given as Ts/Tp below in Table 3 is a structure of speed up data.

Speedup		4	9	16
	120	1.728019	1.636888	2.64309
	240	2.413388	3.45569	4.669867
	360	6.652658	7.060823	15.51252
	480	5.322971	7.399976	13.00245
	600	11.91474	18.34514	32.21459
	720	8.228763	25.10147	44.28567
	840	14.58675	27.95304	56.94361
	960	16.40435	26.41203	66.6743
	1080	12.57125	26.43238	73.73979
	1200	8.288225	38.80443	76.42316
	1320	8.990969	35.04304	84.77318
	1440	7.258386	39.23545	53.81652
	1560	6.121548	35.11201	82.8301
	1680	5.52547	28.37175	71.04847
	1800	5.37981	19.4992	68.43336
	1920	4.98964	19.73125	71.91791
	2040	1.843066	17.18195	32.87108
	2160	4.413095	16.64368	55.47347
	2280	4.48644	14.88576	53.75462
	2400	4.595367	13.96591	36.81207
	2520	4.466128	12.63931	40.51524
	2640	4.180002	12.57732	37.66808

Table 3: In the table above, the top row is the number of cores and the far left column is the number of data points (Matrix size N*N). The data in the middle is speed up for each task.

When looking at the data in a surface the data has extraordinary speed up around 720*720 - 2400*2400. Below in Figure 3 is the 3-D surface of the speed up times.

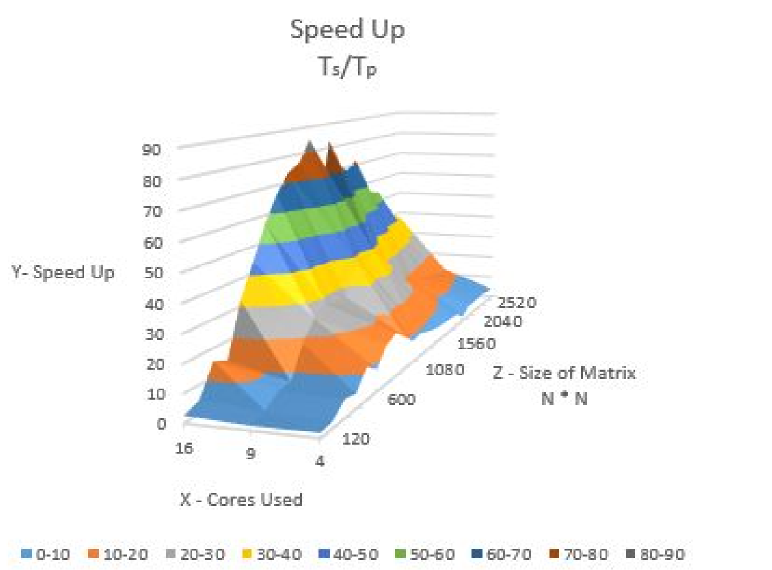


Figure 3: Diagram showing the speed up using each processor configuration where X is the number of cores, Y is the speed up, and Z is the size of matrix N * N.

Efficiency:

Along with looking at the speed up of a parallel program the next piece of data to look at is the efficiency. Efficiency is similar to how speed up is calculated but it is calculated as Ts/(Tp* # Cores). Below in Table 4 is a graph showing the efficiency.

Efficiency		4	9	16
	120	0.432005	0.181876	0.165193
	240	0.603347	0.383966	0.291867
	360	1.663164	0.784536	0.969532
	480	1.330743	0.82222	0.812653
	600	2.978684	2.038349	2.013412
	720	2.057191	2.789052	2.767854
	840	3.646687	3.105893	3.558976
	960	4.101086	2.93467	4.167144
	1080	3.142813	2.936931	4.608737
	1200	2.072056	4.311603	4.776448
	1320	2.247742	3.893671	5.298324
	1440	1.814596	4.359494	3.363533
	1560	1.530387	3.901335	5.176881
	1680	1.381368	3.152416	4.440529
	1800	1.344953	2.166578	4.277085
	1920	1.24741	2.192361	4.494869
	2040	0.460767	1.909105	2.054443
	2160	1.103274	1.849298	3.467092
	2280	1.12161	1.653973	3.359664
	2400	1.148842	1.551768	2.300754
	2520	1.116532	1.404368	2.532203
	2640	1.045001	1.39748	2.354255

Table 4: In the table above, the top row is the number of cores and the far left column is the number of data points (Matrix size N*N). The data in the middle is efficiency.

When looking at the data in a surface the data has spikes in efficiency around 720*720 - 2400*2400. This range is the exact same as the speed up. Below in figure 4 shows a surface of the efficiency.

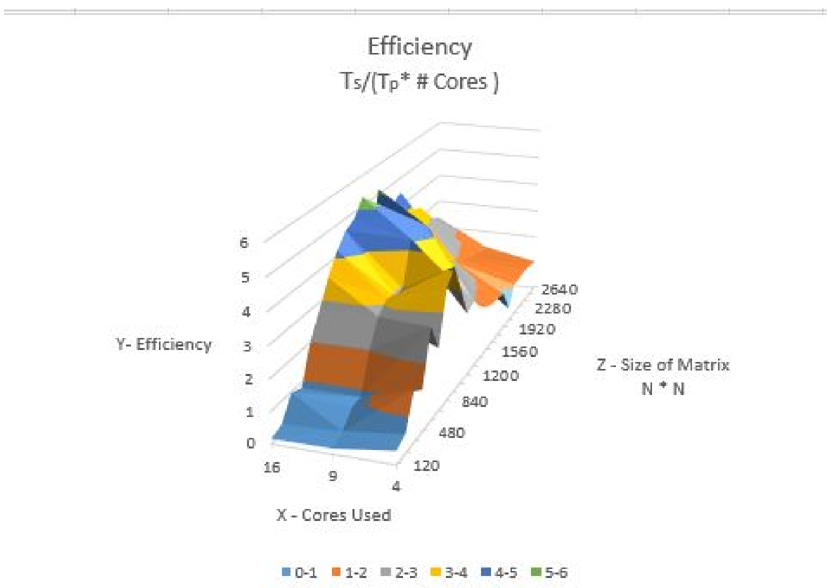


Figure 4: Diagram showing the efficiency using each processor configuration where X is the number of cores, Y is the efficiency, and Z is the size of matrix N * N.

Conclusion/Discussion:

When looking at the sequential time vs the parallel time the speed up was extraordinary. The speed up is due to the fact that even though there is a large of communication the amount of computation is quite large. During runtime the time taken to execute has small spikes this is due to maybe cache and some thrashing occurring. Overall, the speed up was very high when compared to sequential time. As for parallel, when it came to large matrices the speed up started to go down toward the end but the ideal amount of cores for this project seems to be 16 as it got the most speed up and efficiency. Below in the appendix is a small log of changes made halfway through this project and the source code for this project.

Appendix

Appendix 1:

In the midway point of the assignment we had a code review which included suggested changes to make for the assignment. Below is a list of changes made from the midway point.

- Added in the ability to read in from a file
- Added more functionality
- Optimized runtime
- Can switch between file I/O and reading in from a file.
- Improved message passing and fixed major loop error.

Appendix 2:

Source Code:

```
/*
Project 4: Matrix Multiplication
Name: #################
Due: April 20th, 2017
*/

// Header files
#include <iostream>
#include <mpi.h>
#include <fstream>
#include <vector>
#include <sys/time.h>
#include <stdlib.h>
#include <time.h>
#include <vector>
#include <algorithm>
#include <math.h>

using namespace std;

// tags and const that are important to the project
#define TERMINATE_TAG 0
#define DATA_TAG 1
#define RESULT_TAG 2
#define MY_MPI_DATA_TAG 3
#define MASTER 0
#define MAX_NUM 100000
#define M_A_DATA 10
#define M_B_DATA 11

// function prototype for calculation

/*
Function to do all computations for master
*/
void master( char **argv, int argc );

/*
Function that takes all data from master thats been sent and does its own calculations
*/
void slave( int taskId );

/*
Makes a whole shift left using a send replace functions and uses position to find it out
*/
void shiftLeft( int *matA, int size, int myProcessor, int numProcessors );

/*
Makes a whole shift up using a send replace functions and uses position to find it out
*/
void shiftUp( int *matB, int size, int myProcessor, int numProcessors );

/*
Finds the id of the processor that is to the left of the current processor (myProcessor)
*/
int getIdLeft( int myProcessor, int numProcessors );

/*
Finds the id of the processor that is above the current processor (myProcessor)
*/
```

```
int getIdUp( int myProcessor, int numProcessors );

/*
Finds the id of the processor that is to the right of the current processor (myProcessor)
*/
int getIdRight( int myProcessor, int numProcessors );

/*
Finds the id of the processor that is below the current processor (myProcessor)
*/
int getIdDown( int myProcessor, int numProcessors );

/*
Generates random numbers for arrays A and B. Sets C to 0
*/
void genNumbers( int *arrayA, int *arrayB, int *arrayC, int sizeN, int argc, char **argv );

/*
Generates zeroes for C (Needed for Slave)
*/
void genZeroes( int **arrayC, int sizeN );



void initShift(int myrank, int numCores, int left, int up, int right, int down, int size, int root, int* dataA, int* dataB);


//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////


// main function
int main( int argc, char **argv ) {
  /*
  */
// /
  int rank;
  int numProcessors;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &numProcessors );

  FILE *masterFp;
 //  masterFp = fopen("main.txt", "a+");

  //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
  if( rank == 0 ) // Master
  {
    master(argv, argc);
  }

  //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
  else
  {
    slave(rank);
  }
  //
  //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
  MPI_Finalize();

  return 0;
}

////////////// THIS IS FOR THE INCLUDED FUNCTION ///////////
```

```
void master(char **argv, int argc )
  {

  FILE *masterFp;
  // masterFp = fopen("timing.txt", "a+");
  clock_t startProgram, endProgram;
  int myRank, numProcessors;
  MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
  MPI_Comm_size( MPI_COMM_WORLD, &numProcessors );



  // Initialize Variables
  int sizeN = atoi(argv[1]);
  int sumMatrixDimension = sizeN/(int)sqrt(numProcessors);
  int fullMatrixDimension = sizeN;
  int rowOriginForSubMatrix;
  int colOriginForSubMatrix;
  int processNum;
  int subMatrixSize = sumMatrixDimension*sumMatrixDimension;
  int shiftAmnt = MASTER; // For Initial Shift
  int shifts;
  int loopAmnt;


  // Init Size and populate array with rand Numbers


    int *arrayA = new int [sizeN*sizeN];
    int *arrayB = new int [sizeN*sizeN];
    int *arrayC = new int [sizeN*sizeN];

    int *sendArrayA = new int [subMatrixSize];
    int *sendArrayB = new int [subMatrixSize];

    int *myArrayA = new int [subMatrixSize];
    int *myArrayB = new int [subMatrixSize];

  genNumbers( arrayA, arrayB, arrayC, sizeN, argc, argv);



  // go though the rows with offset
  for(int verticalOffset = 0 ; verticalOffset < (int)sqrt(numProcessors); verticalOffset++)
    {

    // go thorugh the columns using the offset
    for(int horizontalOffset = 0; horizontalOffset < (int)sqrt(numProcessors); horizontalOffset++)
      {

      //go trhough the rows of submatrix
      for(int row = 0; row < sumMatrixDimension; row++)
        {
          ////
        // go throguh the cols of submatrix
        for(int col = 0; col < sumMatrixDimension; col++)
          {

          // get location in the submatrix
          rowOriginForSubMatrix = ((verticalOffset * sumMatrixDimension) * fullMatrixDimension);
          colOriginForSubMatrix = (horizontalOffset*sumMatrixDimension);

          //load sub array by going through the submatrix by its location found
          sendArrayA[row*sumMatrixDimension + col] = arrayA[(rowOriginForSubMatrix + (row*fullMatrixDimension)) +
(colOriginForSubMatrix + col)];
          sendArrayB[row*sumMatrixDimension + col] = arrayB[(rowOriginForSubMatrix + (row*fullMatrixDimension)) +
```

```
(colOriginForSubMatrix + col)];

                }
              }

          // send dtata to processes
          processNum = verticalOffset*(int)sqrt(numProcessors) + horizontalOffset;



          //if the process num is MASTER put data in myArray<A or B>
          if(processNum == MASTER)
          {
             for(int i = 0; i < subMatrixSize; i++)
             {
                myArrayA[i] = sendArrayA[i];
                myArrayB[i] = sendArrayB[i];
             }
          }

          //actually send to other processes
          else if(processNum > MASTER){


             MPI_Send(sendArrayA, subMatrixSize, MPI_INT, processNum, M_A_DATA, MPI_COMM_WORLD);
             // MPI_Barrier(MPI_COMM_WORLD);


             MPI_Send(sendArrayB, subMatrixSize, MPI_INT, processNum, M_B_DATA, MPI_COMM_WORLD);

            // MPI_Barrier(MPI_COMM_WORLD);


          }


       }
      // MPI_Barrier(MPI_COMM_WORLD);

   }

// ADD A BARRIER



////////////////////////////////////////////////////

            // START THE MATRIX MATH HERE

////////////////////////////////////////////////////

   int twoDimSplit = (int)(subMatrixSize/(int)sqrt(subMatrixSize));


   int **myA = new int *[twoDimSplit];
   int **myB = new int *[twoDimSplit];
   int **myC = new int *[twoDimSplit];

   for( int i = 0; i < twoDimSplit; i++ )
   {
   myA[i] = new int [twoDimSplit];
   myB[i] = new int [twoDimSplit];
   myC[i] = new int [twoDimSplit];
   }
```

```
    genZeroes(myC, (int)(subMatrixSize/(int)sqrt(subMatrixSize)) );


    MPI_Barrier(MPI_COMM_WORLD);

    startProgram = clock();


//////////////////////
for( loopAmnt = 0; loopAmnt < (int)sqrt(numProcessors); loopAmnt++ )
    {

 /////////////

 ///   CONVERT 2D ARAYS

//////


int offset = subMatrixSize/(int)sqrt(subMatrixSize);
int offsetTimesJ;

    for( int i = 0; i < subMatrixSize/(int)sqrt(subMatrixSize); i++)
    {
        for( int j = 0; j < subMatrixSize/(int)sqrt(subMatrixSize); j++)
            {
                offsetTimesJ = i * offset;

                //Add offset * y to the lenggth
                myA[i][j] = arrayA[ offsetTimesJ + j]; ///////// May need to sway I and J



                myB[i][j] = arrayB[ offsetTimesJ + j]; /////////


            }
    }
    // Optimize Vars Later
    // MULT NUMBER


    int loopLength = (int)sqrt(subMatrixSize);


    for (int i = 0; i < loopLength; i++)
    {
        for (int j = 0; j < loopLength; j++)
        {
            for (int k = 0; k < loopLength; k++)
            {
                myC[i][j] = myC[i][j] + myA[i][k] * myB[k][j];
            }
        }
    }
    //////

    // Put into 1D array for passing
    for( int i = 0; i < subMatrixSize/(int)sqrt(subMatrixSize); i++)
    {
        for( int j = 0; j < subMatrixSize/(int)sqrt(subMatrixSize); j++)
            {
            offsetTimesJ = i * offset;
```

```
                    arrayA[offsetTimesJ + j] = myA[i][j];
                    arrayB[offsetTimesJ + j] = myB[i][j];
          }
    }


//////////////////////////////////////////////////////////////////////////////////////////////
    // Do final shift (Shift Amount is made by task id % sqrtNumP)
        shiftLeft( arrayA, subMatrixSize, myRank, numProcessors );
        shiftUp( arrayB, subMatrixSize, myRank, numProcessors );
//////////////////////////////////////////////////////////////////////////////////////////////
    }



MPI_Barrier(MPI_COMM_WORLD);

endProgram = clock();


    cout << sizeN << " " << ((float)(endProgram-startProgram)/CLOCKS_PER_SEC) << endl;



///////////////
    // DELETE THESE SAVAGE BEASTS


        delete arrayA;
      delete arrayB;
        delete arrayC;

        delete sendArrayA;
        delete sendArrayB;

        delete myArrayA;
        delete myArrayB;

        arrayA = NULL;
       arrayB = NULL;
       arrayC = NULL;

       sendArrayA = NULL;
       sendArrayB = NULL;

       myArrayA = NULL;
       myArrayB = NULL;


    }

void slave( int taskId )
    {


    FILE *fp;
   // fp = fopen("slave.txt", "a+");

    int myRank;

    MPI_Comm_rank( MPI_COMM_WORLD, &myRank );

//////////////////////////////

    int numProcessors;
```

```
// int shiftAmnt = taskId; // For Initial Shift
int shiftAmnt = myRank;
int shifts;
int subMatrixSize;
MPI_Status status;
int loopAmnt;


MPI_Comm_size( MPI_COMM_WORLD, &numProcessors );

MPI_Probe(MASTER, M_A_DATA, MPI_COMM_WORLD, &status );

MPI_Get_count( &status, MPI_INT, &subMatrixSize );

int *arrayA = new int [subMatrixSize];

 MPI_Recv( arrayA, subMatrixSize, MPI_INT, 0, M_A_DATA, MPI_COMM_WORLD, MPI_STATUS_IGNORE );




MPI_Probe(MASTER, M_B_DATA, MPI_COMM_WORLD, &status );

MPI_Get_count( &status, MPI_INT, &subMatrixSize );


int *arrayB = new int [subMatrixSize];



 MPI_Recv( arrayB, subMatrixSize, MPI_INT, MASTER, M_B_DATA, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
// MPI_Barrier(MPI_COMM_WORLD);
//    INIT 2D ARAYS


int SmallBoxDimension = ((int)sqrt(subMatrixSize));

int **myA = new int *[SmallBoxDimension];
int **myB = new int *[SmallBoxDimension];
int **myC = new int *[SmallBoxDimension];


for( int i = 0; i < SmallBoxDimension; i++ )
{
myA[i] = new int [SmallBoxDimension];
myB[i] = new int [SmallBoxDimension];
myC[i] = new int [SmallBoxDimension];
}



genZeroes(myC, (int)sqrt(subMatrixSize) );


//

int left = getIdLeft(myRank, numProcessors);
int right = getIdLeft(myRank, numProcessors);
int down = getIdDown(myRank, numProcessors);
int up = getIdUp(myRank, numProcessors);

////////////////////////////////////////////////////////////////////////////////////////////
// Initial shift (Shift Amount is made by task id % sqrtNumP)
```

```
MPI_Barrier(MPI_COMM_WORLD);


initShift( myRank, numProcessors, left, up, right, down, subMatrixSize, (int)sqrt(numProcessors), arrayA, arrayB);

////////////////////////////////////////////////////////////////////////////////////////

//    Loop for the rest of the multiplication

for( loopAmnt = 0; loopAmnt < (int)sqrt(numProcessors); loopAmnt++ )
   {


//    CONVERT 2D ARAYS
   int offset = (int)sqrt(subMatrixSize);
   int offsetTimesY;

   for( int i = 0; i < (int)sqrt(subMatrixSize); i++)
   {
      for( int j = 0; j < (int)sqrt(subMatrixSize); j++)
         {
            offsetTimesY = i * offset;




            //Add offset * y to the lenggth
            myA[i][j] = arrayA[ offsetTimesY + j];
            myB[i][j] = arrayB[ offsetTimesY + j];

         }
   }

   // Optimize Vars Later
   // MULTIPLY THE NUMBERS

   int loopLength = (int)sqrt(subMatrixSize);


   for (int i = 0; i < loopLength; i++)
   {
      for (int j = 0; j < loopLength; j++)
      {
         for (int k = 0; k < loopLength; k++)
         {
            myC[i][j] = myC[i][j] + myA[i][k] * myB[k][j];

         }
      }

   }

   // int offset = (int)sqrt(subMatrixSize);
   // int offsetTimesY;
   // offsetTimesY = i * offset;

   // Put into 1D array for passing
   for( int i = 0; i < subMatrixSize/(int)sqrt(subMatrixSize); i++)
   {
      for( int j = 0; j < subMatrixSize/(int)sqrt(subMatrixSize); j++)
         {
         offsetTimesY = i * offset;


         arrayA[offsetTimesY + j] = myA[i][j];
```

```
          arrayB[offsetTimesY + j] = myB[i][j];

          }
   }


////////////////////////////////////////////////////////////////////////////////////////////////////
   // Do final shift (Shift Amount is made by task id % sqrtNumP)
       shiftLeft( arrayA, subMatrixSize, taskId, numProcessors );
       shiftUp( arrayB, subMatrixSize, taskId, numProcessors );
////////////////////////////////////////////////////////////////////////////////////////////////////

   }

MPI_Barrier(MPI_COMM_WORLD);


   }

void initShift(int myrank, int numCores, int left, int up, int right, int down, int size, int root, int* dataA, int* dataB)
{

   int indexCol = (myrank % root);
   int indexRow = (myrank/root);
   int count;

   for(count = 0; count < indexRow; count++)
   {
      shiftLeft( dataA, size, myrank, numCores );
   }

   for(count = 0; count < indexCol; count++)
   {
      shiftUp( dataB, size, myrank, numCores );
   }
}

void shiftLeft( int *matA, int size, int myProcessor, int numProcessors )
   {
   int destProcessor;
   int recvProcessor;
   MPI_Status status;


   destProcessor = getIdLeft( myProcessor, numProcessors );
   recvProcessor = getIdRight( myProcessor, numProcessors );



   MPI_Sendrecv_replace(matA, size, MPI_INT, destProcessor, M_A_DATA, recvProcessor, M_A_DATA, MPI_COMM_WORLD,
&status);

   }

void shiftUp( int *matB, int size, int myProcessor, int numProcessors )
   {
   int destProcessor;
   int recvProcessor;
   MPI_Status status;

   destProcessor = getIdUp( myProcessor, numProcessors );
   recvProcessor = getIdDown( myProcessor, numProcessors );
```

```
   MPI_Sendrecv_replace(matB, size, MPI_INT, destProcessor, M_B_DATA, recvProcessor, M_B_DATA, MPI_COMM_WORLD,
&status);

   }

int getIdUp( int myProcessor, int numProcessors )
   {
   int idAboveMe;

   if(( myProcessor < (int)sqrt(numProcessors)))
      {
         idAboveMe = myProcessor + (numProcessors - (int)sqrt(numProcessors));
      }

   else{

      idAboveMe = myProcessor - (int)sqrt(numProcessors);
   }

   return idAboveMe;
   }


int getIdDown( int myProcessor, int numProcessors )
   {
   int idBelowMe;

   if( myProcessor < ( numProcessors - (int)sqrt(numProcessors)) )
      {
         idBelowMe = myProcessor + (int)sqrt(numProcessors);


      }
   else{

      idBelowMe = myProcessor % (int)sqrt(numProcessors);
   }

   return idBelowMe;
   }

int getIdLeft( int myProcessor, int numProcessors )
   {
   int idToMyLeft;

   if(( myProcessor % (int)sqrt(numProcessors)) == 0 )
      {
         idToMyLeft = myProcessor + ((int)sqrt(numProcessors) - 1);
      }

   else{

      idToMyLeft = myProcessor - 1;

      }

   return idToMyLeft;
   }

int getIdRight( int myProcessor, int numProcessors )
   {
   int idToMyRight;

   if( (myProcessor % (int)sqrt(numProcessors)) >= ((int)sqrt(numProcessors) - 1) )
      {
         idToMyRight = myProcessor - ( (int)sqrt(numProcessors) - 1 );
```

```
    }

  else{
    idToMyRight = myProcessor + 1;

  }
  return idToMyRight;
  }

void genNumbers( int *arrayA, int *arrayB, int *arrayC, int sizeN, int argc, char **argv )
  {

  ifstream fin;



  if( argc < 5 )
  {
  for( int index = 0; index < (sizeN * sizeN); index++ )
    {
      //srand(time(NULL));
        //srand(time(NULL));
         arrayA[index] = rand()%100;

         //srand(time(NULL));
         arrayB[index] = rand()%100;

         // Set array C to 0 (For proper calculations)
         arrayC[index] = 0;
    }
  }
  else{

  fin.clear();
  fin.open(argv[2]);
  int number;
  fin >> number;

  for( int index = 0; index < (sizeN * sizeN); index++ )
    {
    fin >> number;
      //srand(time(NULL));
        //srand(time(NULL));
         arrayA[index] = number;
         // Set array C to 0 (For proper calculations)
         arrayC[index] = 0;
    }
  fin.close();

  // Open second file though this is very unoptimized... NOT RECOMMENDED FOR EVERY DAY USE

  fin.clear();
  fin.open(argv[3]);
  fin >> number;

  for( int index = 0; index < (sizeN * sizeN); index++ )
    {
    fin >> number;
      //srand(time(NULL));
        //srand(time(NULL));
         arrayB[index] = number;
         // Set array C to 0 (For proper calculations)
    }
  fin.close();

  }
```

```
   }

void genZeroes( int **arrayC, int sizeN )
   {
   for( int index = 0; index < sizeN; index++ )
      {
         for( int j = 0; j < sizeN; j++ )
             arrayC[index][j] = 0;
      }

   }
```