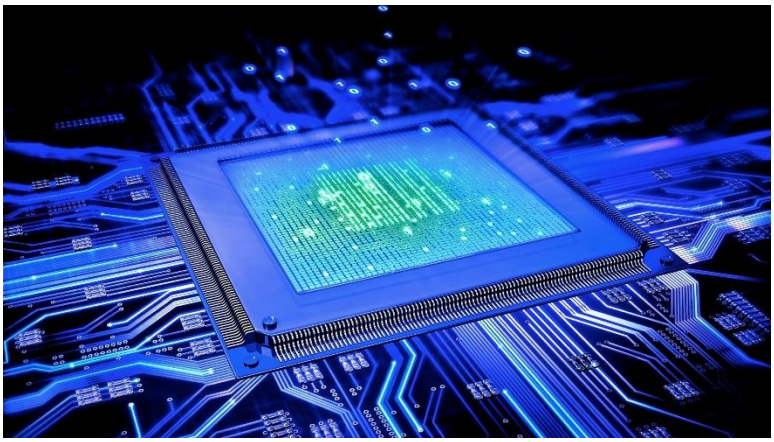

Homework #3

CPE 512

KYLE RAY

October 10, 2017

Contents

Part 1 Serial Test

CODE

- See Appendix A for the source code

OUTPUT

```
uahcls01@dmcvlogin1:Hw3> ./mm_mult_serial 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time=1e-06 seconds
```

Part 2 Parallel Test

CODE

- See Appendix B for the source code

OUTPUT NP = 1

```
uahcls01@dmcvlogin2:Hw3> mpirun -np 1 ./mm_mult_mpi 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
```

```
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time=1.7e-05 seconds
```

OUTPUT NP = 2

```
uahcls01@dmcvlogin2:Hw3> mpirun -np 2 ./mm_mult_mpi 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time=9.9e-05 seconds
```

OUTPUT NP = 3

```
uahcls01@dmcvlogin2:Hw3> mpirun -np 3 ./mm_mult_mpi 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time=0.000154 seconds
```

OUTPUT NP = 4

```
uahcls01@dmcvlogin2:Hw3> mpirun -np 4 ./mm_mult_mpi 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
```

```
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time=0.00015 seconds
```

OUTPUT NP = 5

```
uahcls01@dmcvlogin2:Hw3> mpirun -np 5 ./mm_mult_mpi 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time=0.004219 seconds
```

OUTPUT NP = 6

```
uahcls01@dmcvlogin2:Hw3> mpirun -np 6 ./mm_mult_mpi 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
```

```
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time=0.000299 seconds
```

OUTPUT NP = 7

```
uahcls01@dmcvlogin2:Hw3> mpirun -np 7 ./mm_mult_mpi 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time=0.004195 seconds
```

OUTPUT NP = 8

```
uahcls01@dmcvlogin2:Hw3> mpirun -np 8 ./mm_mult_mpi 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time=0.000227 seconds
```

I originally ran the program with np = 2, 4, and 8 first and then proceeded to go back and run the rest of the cases after reading the homework description more closely. My connection to the dmc seemed slow on the night I ran np = 1, 3, 5, 6, and 7 which I think was the cause of the substantial increase in completion time.

Part 3 Timing Analysis

SERIAL

```
Matrix Size =  256, time = 0.107185 seconds
Matrix Size =  512, time = 0.862933 seconds
Matrix Size =  768, time = 4.647209 seconds
Matrix Size = 1024, time = 26.243087 seconds
Matrix Size = 1280, time = 46.751129 seconds
Matrix Size = 1536, time = 106.47022 seconds
Matrix Size = 1792, time = 180.9898 seconds
Matrix Size = 2048, time = 371.17542 seconds
Matrix Size = 2304, time = 490.22311 seconds
Matrix Size = 2560, time = 665.90334 seconds
Matrix Size = 2816, time = 950.09028 seconds
Matrix Size = 3072, time = 1154.8234 seconds
Matrix Size = 3328, time = 1447.418 seconds
Matrix Size = 3584, time = 2131.6487 seconds
```

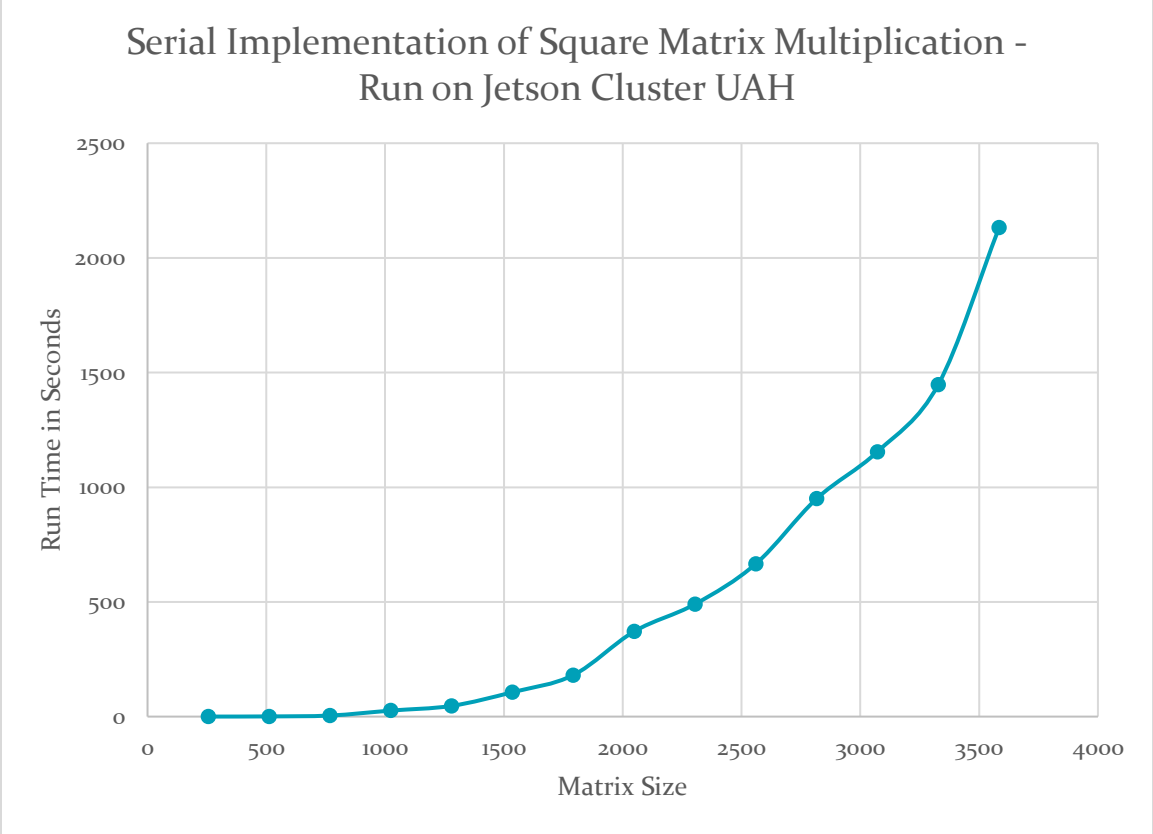

Figure 1. Serial Matrix Multiplication Run Time Characteristics

The base algorithm behaves as expected. We know that the algorithm used has an order notation of $O(n^3)$ and looking at **Figure 1** it can be seen that the run time is exponentially increasing as the matrix size increases.

PARALLEL

NP = 2

```
Matrix Size =  256, time = 0.069588 seconds
Matrix Size =  512, time = 0.510835 seconds
Matrix Size =  768, time = 2.1731 seconds
Matrix Size = 1024, time = 13.790599 seconds
Matrix Size = 1280, time = 23.50243 seconds
Matrix Size = 1536, time = 55.698489 seconds
Matrix Size = 1792, time = 97.612683 seconds
Matrix Size = 2048, time = 188.87857 seconds
Matrix Size = 2304, time = 244.05105 seconds
Matrix Size = 2560, time = 347.84285 seconds
Matrix Size = 2816, time = 474.08665 seconds
Matrix Size = 3072, time = 576.89958 seconds
Matrix Size = 3328, time = 736.07543 seconds
Matrix Size = 3584, time = 1175.1372 seconds
```

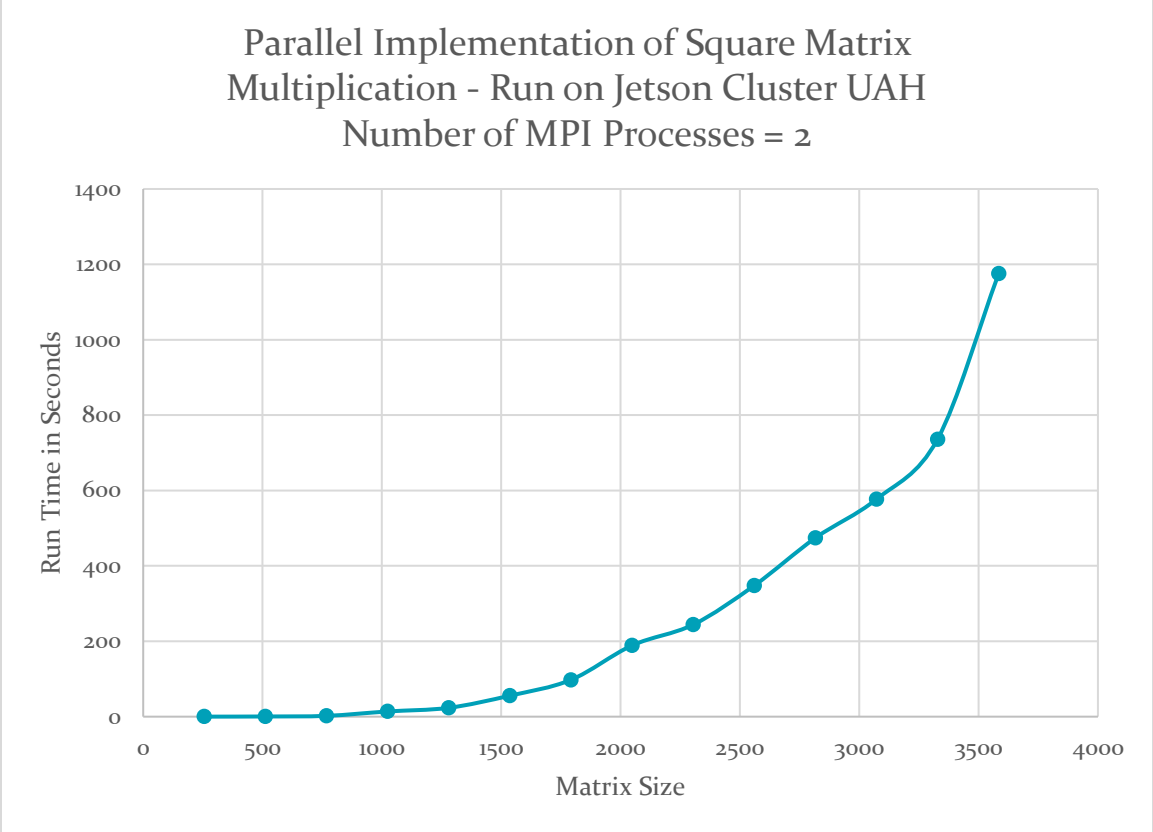

Figure 2. Parallel Matrix Multiplication Run Time Characteristics, NP(2)

Comparing **Figure 2** with **Figure 1** it seems that utilizing one more process and splitting the work evenly between the two has cut the completion time almost in half for every case, which is in line with what we would expect to see.

NP = 4

```
Matrix Size =  256, time = 0.054693 seconds
```

```
Matrix Size =  512, time = 0.289523 seconds
Matrix Size =  768, time = 1.221861 seconds
Matrix Size = 1024, time = 6.843977 seconds
Matrix Size = 1280, time = 11.904123 seconds
Matrix Size = 1536, time = 27.318558 seconds
Matrix Size = 1792, time = 48.401298 seconds
Matrix Size = 2048, time = 97.835424 seconds
Matrix Size = 2304, time = 121.94584 seconds
Matrix Size = 2560, time = 172.05514 seconds
Matrix Size = 2816, time = 236.1049 seconds
Matrix Size = 3072, time = 303.81367 seconds
Matrix Size = 3328, time = 370.86322 seconds
Matrix Size = 3584, time = 584.54626 seconds
```

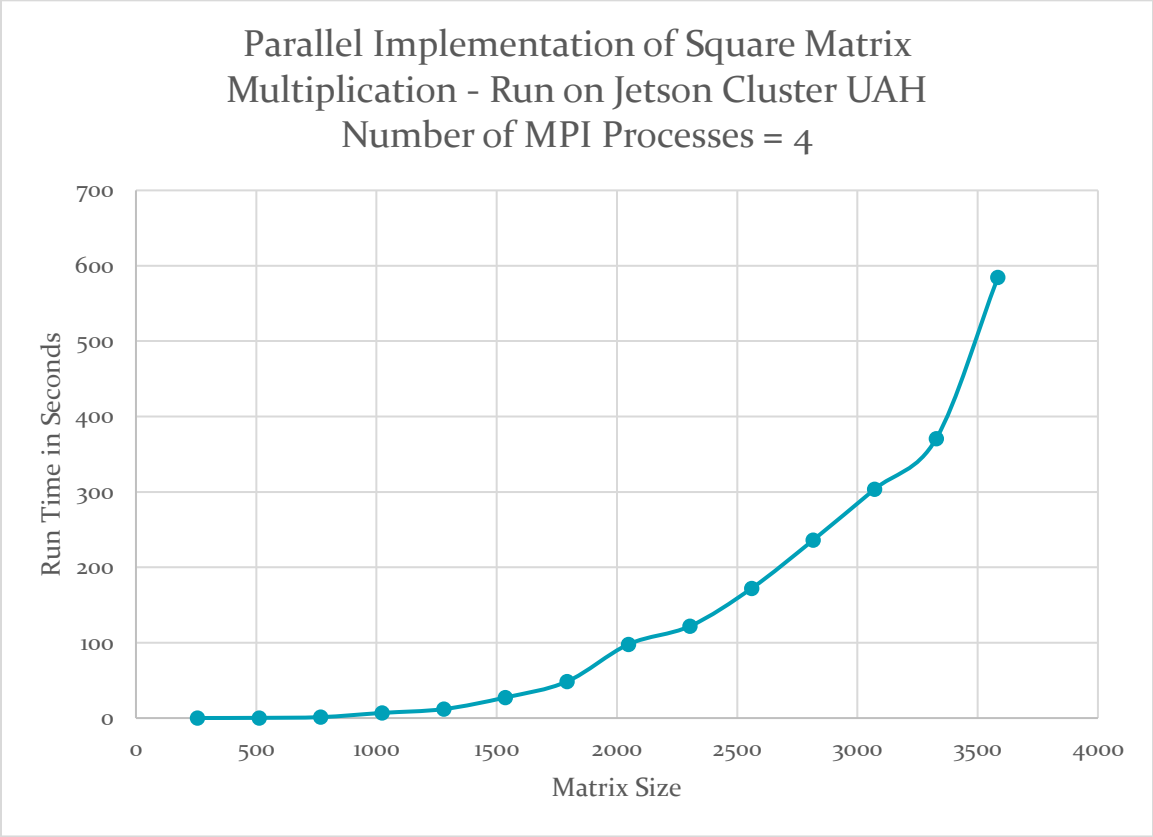

Figure 3. Parallel Matrix Multiplication Run Time Characteristics, NP(4)

NP = 8

```
Matrix Size =  256, time = 0.042516 seconds
Matrix Size =  512, time = 0.196104 seconds
Matrix Size =  768, time = 0.726653 seconds
Matrix Size = 1024, time = 3.569868 seconds
Matrix Size = 1280, time = 6.044348 seconds
Matrix Size = 1536, time = 13.962634 seconds
Matrix Size = 1792, time = 24.515898 seconds
Matrix Size = 2048, time = 49.485545 seconds
```

```
Matrix Size = 2304, time = 61.588715 seconds
Matrix Size = 2560, time = 88.130733 seconds
Matrix Size = 2816, time = 120.32 seconds
Matrix Size = 3072, time = 156.76207 seconds
Matrix Size = 3328, time = 187.06935 seconds
Matrix Size = 3584, time = 287.17158 seconds
```

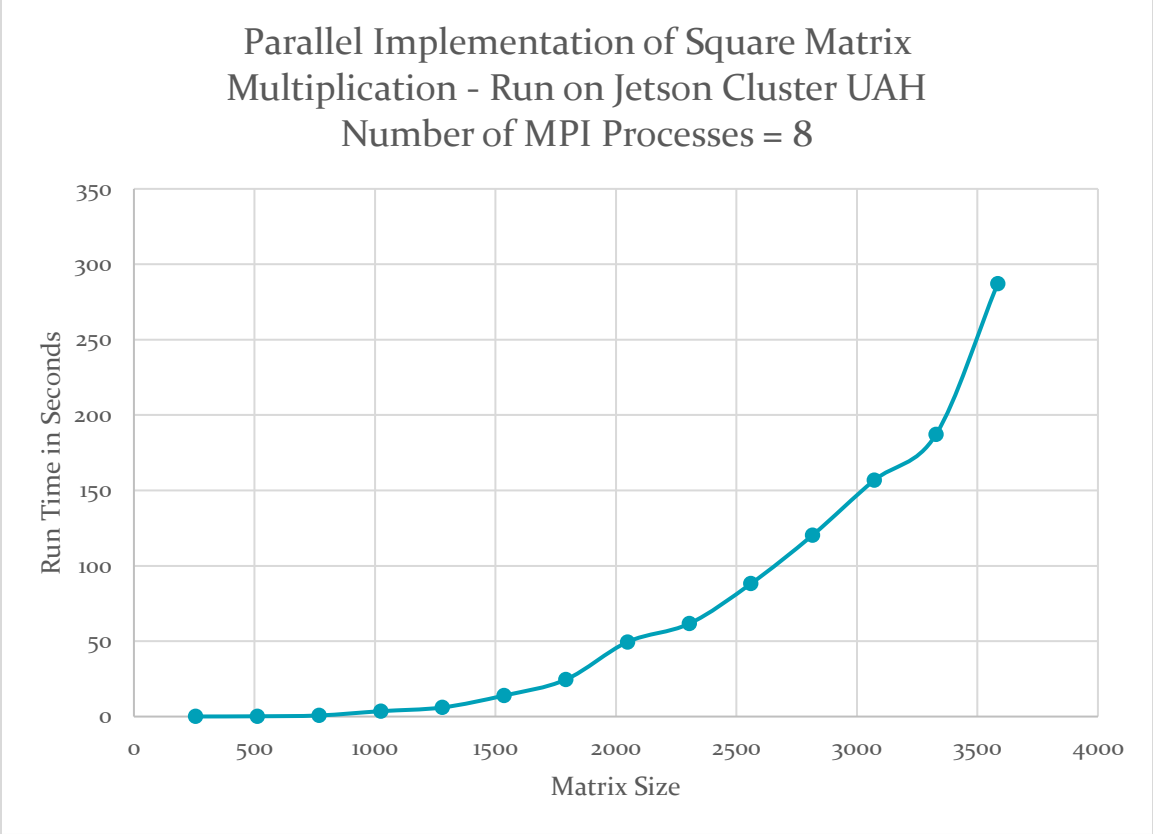

Figure 4. Parallel Matrix Multiplication Run Time Characteristics, NP(8)

Examining the above data, we can see a trend forming. Every time the number of processes is doubled we are seeing the completion time cut almost in half for each run. The above implementations are behaving just like the serial portion but because the work is being distributed and has parallelized the program, it is able to finish in a timelier manner. We can see from each graph, of each parallel run, that the run time is increasing at an exponential rate as matrix size increases which is consistent with the behavior of the serial program. The algorithm is executing $n * \frac{n*n}{p}$ times, where p is the number of processes. Therefore, the order notation for each implementation in an MPI process is $O(n^2)$ but overall the algorithm is still $O(n^3)$.

Below are the graphs for relative speed up and efficiency with regards to the serial version of the program.

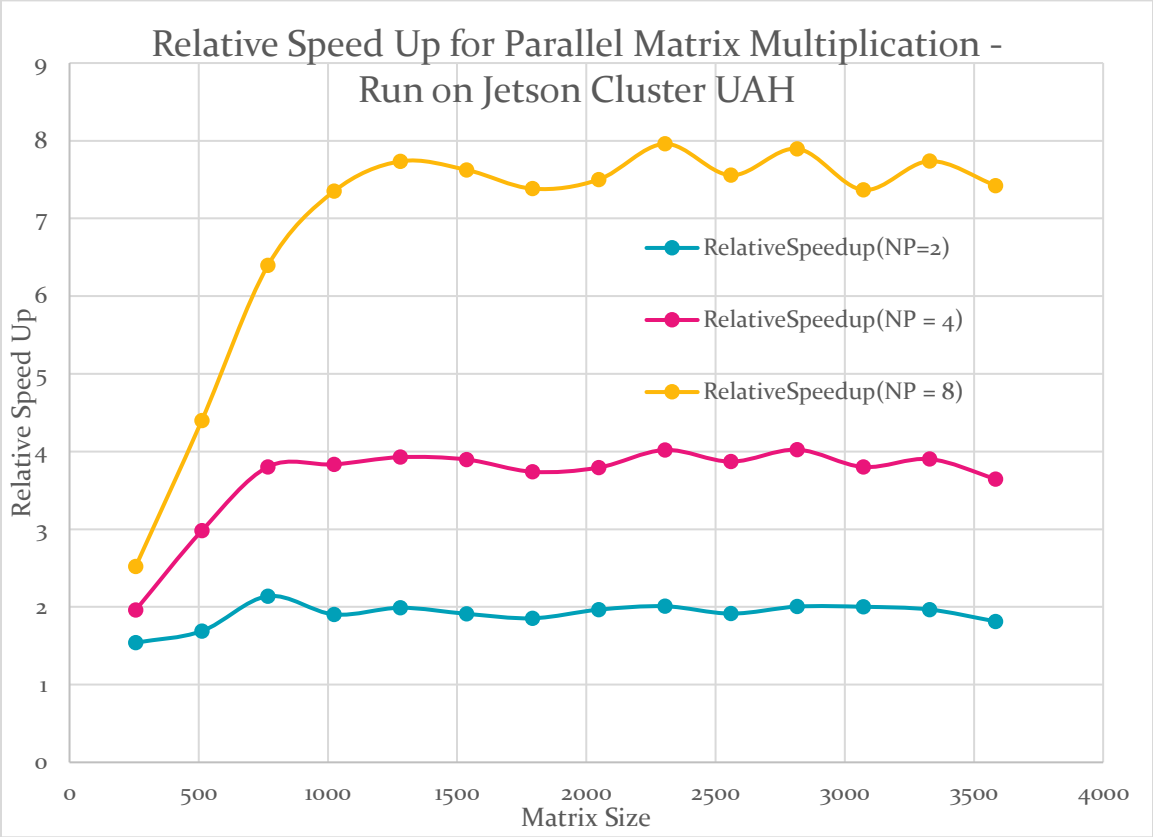

Figure 5. Parallel Matrix Multiplication Relative Speed Up, NP(2, 4, 8)

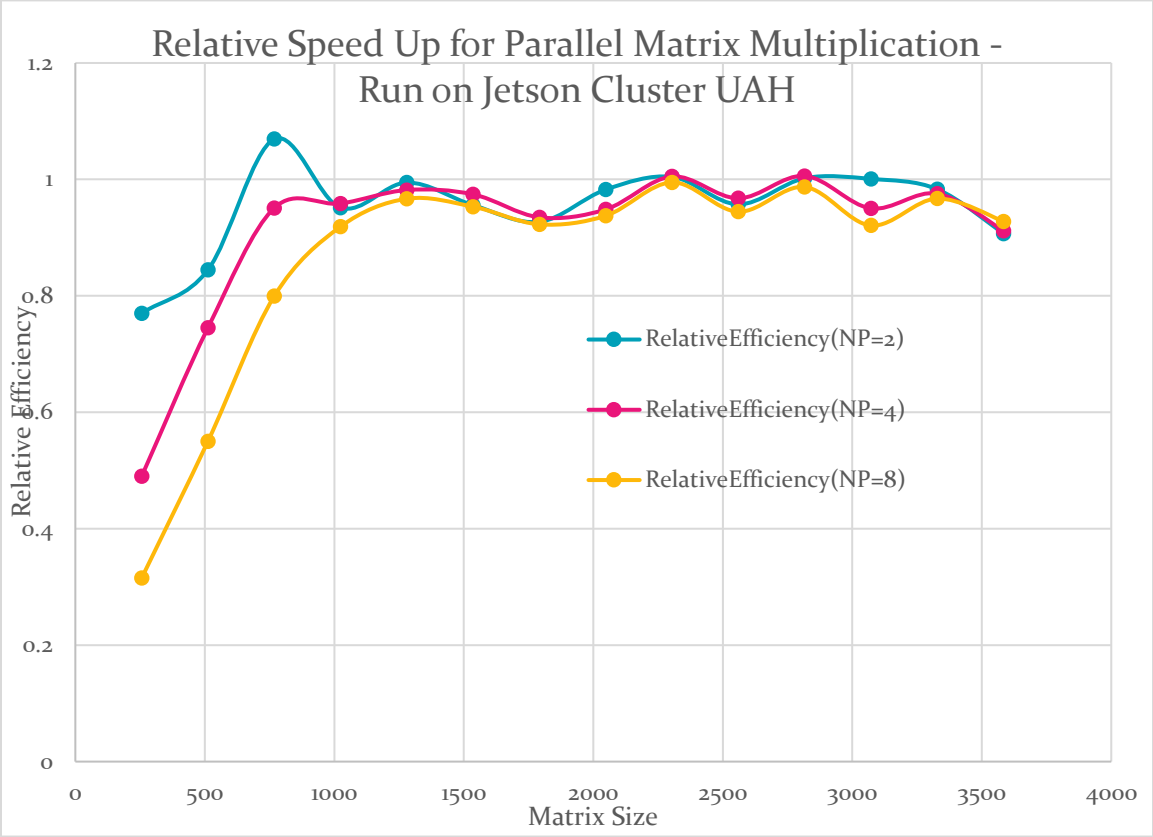

Figure 5. Parallel Matrix Multiplication Relative Efficiency, NP(2, 4, 8)

Appendix

A) SERIAL CODE

```
*/

using namespace std;
#include <iostream>
#include <iomanip>
#include <sstream>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

#define MX_SZ 320
#define SEED 2397           /* random number seed */
#define MAX_VALUE  100.0    /* maximum size of array elements A, and B */

/* copied from mpbench */
#define TIMER_CLEAR     (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec =
0)
#define TIMER_START     gettimeofday(&tv1, (struct timezone*)0)
#define TIMER_ELAPSED   ((tv2.tv_usec-tv1.tv_usec)+((tv2.tv_sec-
tv1.tv_sec)*1000000))
#define TIMER_STOP      gettimeofday(&tv2, (struct timezone*)0)
struct timeval tv1,tv2;

/*
This declaration facilitates the creation of a two dimensional
dynamically allocated arrays (i.e. the lxm A array, the mxn B
array, and the lxn C array).  It allows pointer arithmetic to
be applied to a single data stream that can be dynamically allocated.
To address the element at row x, and column y you would use the
following notation:  A(x,y),B(x,y), or C(x,y), respectively.
Note that this differs from the normal C notation if A were a
two dimensional array of A[x][y] but is still very descriptive
of the data structure.
*/
float *a,*b,*c;
#define A(i,j) *(a+i*dim_m+j)
#define B(i,j) *(b+i*dim_n+j)
#define C(i,j) *(c+i*dim_n+j)

/*
   Routine to retrieve the data size of the numbers array from the
   command line or by prompting the user for the information
*/
void get_index_size(int argc,char *argv[],int *dim_l,int *dim_m,int *dim_n) {
   if(argc!=2 && argc!=4) {
      cout<<"usage:  mm_mult_serial [l_dimension] <m_dimension n_dimmension>"
           << endl;
      exit(1);
   }
   else {
      if (argc == 2) {
         *dim_l = *dim_n = *dim_m = atoi(argv[1]);
      }
      else {
         *dim_l = atoi(argv[1]);
```

```
         *dim_m = atoi(argv[2]);
         *dim_n = atoi(argv[3]);
      }
   }
   if (*dim_l<=0 || *dim_n<=0 || *dim_m<=0) {
      cout<<"Error: number of rows and/or columns must be greater than 0"
          << endl;
      exit(1);
   }
}

/*
   Routine that fills the number matrix with Random Data with values
   between 0 and MAX_VALUE
   This simulates in some way what might happen if there was a
   single sequential data acquisition source such as a single file
*/
void fill_matrix(float *array,int dim_m,int dim_n)
{
   int i,j;
   for(i=0;i<dim_m;i++) {
      for (j=0;j<dim_n;j++) {
         array[i*dim_n+j]=drand48()*MAX_VALUE;
      }
   }
}

/*
   Routine that outputs the matrices to the screen
*/
void print_matrix(float *array,int dim_m,int dim_n)
{
   int i,j;
   for(i=0;i<dim_m;i++) {
      for (j=0;j<dim_n;j++) {
         cout << array[i*dim_n+j] << " ";
      }
      cout << endl;
   }
}

/*
   MAIN ROUTINE: summation of a number list
*/

int main( int argc, char *argv[])
{
   float dot_prod;
   int dim_l,dim_n,dim_m;
   int i,j,k;

   /*
   get matrix sizes
   */
   get_index_size(argc,argv,&dim_l,&dim_m,&dim_n);

   // dynamically allocate from heap the numbers in the memory space
   // for the a,b, and c matrices
   a = new (nothrow) float[dim_l*dim_m];
   b = new (nothrow) float[dim_m*dim_n];
   c = new (nothrow) float[dim_l*dim_n];
   if(a==0 || b==0 || c==0) {
```

```
      cout <<"ERROR:  Insufficient Memory" << endl;
      exit(1);
    }

    /*
       initialize numbers matrix with random data
    */
    srand48(SEED);
    fill_matrix(a,dim_l,dim_m);
    fill_matrix(b,dim_m,dim_n);

    /*
      output numbers matrix
    */
   // cout << "A matrix =" << endl;
   // print_matrix(a,dim_l,dim_m);
   // cout << endl;

    //cout << "B matrix =" << endl;
    //print_matrix(b,dim_m,dim_n);
    //cout << endl;

    /*
    Start recording the execution time
    */
    TIMER_CLEAR;
    TIMER_START;

    // multiply local part of matrix
    for (i=0;i<dim_l;i++) {
       for (j=0;j<dim_n;j++) {
          dot_prod = 0.0;
          for (k=0;k<dim_m;k++) {
             dot_prod += A(i,k)*B(k,j);
          }
          C(i,j) = dot_prod;
       }
    }

    /*
       stop recording the execution time
    */
    TIMER_STOP;

    //cout << "C matrix =" << endl;
    //print_matrix(c,dim_l,dim_n);
    //cout << endl;
    cout << "time=" << setprecision(8) <<  TIMER_ELAPSED/1000000.0
         << " seconds" << endl;

}
```

B) PARALLEL CODE

```
/*****************************************************************/
/* Matrix Matrix Multiplication Program Example -- serial version */
/* September 2016 -- B. Earl Wells -- University of Alabama       */
/*                                    in Huntsville              */
/*****************************************************************/
// mm_mult_serial.cpp
// compilation:
//   gnu compiler
//      g++ mm_mult_serial.cpp -o mm_mult_serial -O3 -lm
// Note: to compile a parallel MPI program version which is named
//   mm_mult_mpi.cpp
//   then execute the following command
//      gnu compiler
//         mpic++ mm_mult_mpi.cpp -o mm_mult_MPI_gnu -lm  -O3
/*
   This program is designed to perform matrix matrix multiplication
   A x B = C, where A is an lxm matrix, B is a m x n matrix and
   C is a l x n matrix. The program is designed to be a template
   serial program that can be expanded into a parallel multiprocess
   and/or a multi-threaded program.

   The program randomly assigns the elements of the A and B matrix
   with values between 0 and a MAX_VALUE. It then multiples the
   two matrices with the result being placed in the C matrix.
   The program prints out the A, B, and C matrices.

   The program is executed using one or three command line parameters.
   These parameters represent the dimension of the matrices. If only
   one parameter is used then then it is assumed that square matrices are
   to be created and multiplied together that have the specified
   dimension. In cases where three command line parameters are entered
   then the first parameter is the l dimension, the second the m, and
   the third is the n dimension.

   To execute:
   mm_mult_serial [l_parameter] <m_parameter n_parameter>

   Editted by Kyle Ray
   October 12, 2017
   Homework #3

   Changed the serial version of the program to utilize the MPI library to
   perform the matrix multiplication in parallel on a number of MPI processes
   passed in on the command line.
*/

using namespace std;
#include <iostream>
#include <iomanip>
#include <sstream>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
//#include <time.h>
#include <mpi.h>

#define MX_SZ 320
#define SEED 2397           /* random number seed */
#define MAX_VALUE  100.0    /* maximum size of array elements A, and B */
```

```
/* copied from mpbench */
#define TIMER_CLEAR     (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec =
0)
#define TIMER_START     gettimeofday(&tv1, (struct timezone*)0)
#define TIMER_ELAPSED   ((tv2.tv_usec-tv1.tv_usec)+((tv2.tv_sec-
tv1.tv_sec)*1000000))
#define TIMER_STOP      gettimeofday(&tv2, (struct timezone*)0)
struct timeval tv1,tv2;

// Defines so that I can compile the code in visual studio
//#define srand48(s) srand(s)
//#define drand48() (((double)rand())/((double)RAND_MAX))


/*
This declaration facilitates the creation of a two dimensional
dynamically allocated arrays (i.e. the lxm A array, the mxn B
array, and the lxn C array).  It allows pointer arithmetic to
be applied to a single data stream that can be dynamically allocated.
To address the element at row x, and column y you would use the
following notation:  A(x,y),B(x,y), or C(x,y), respectively.
Note that this differs from the normal C notation if A were a
two dimensional array of A[x][y] but is still very descriptive
of the data structure.
*/
float *a, *b, *c;
float *group_a, *group_c;
#define A(i,j) *(a+i*dim_m+j)
#define B(i,j) *(b+i*dim_n+j)
#define C(i,j) *(c+i*dim_n+j)

/*
   Routine to retrieve the matrix dimensions of the arrays from the
   command line.
*/
void get_index_size(int argc, char *argv[], int *dim_l, int *dim_m, int *dim_n,
int rank) {
  if (argc != 2 && argc != 4) {
    if (rank == 0)
    {
      cout << "usage:  mm_mult_serial [l_dimension] <m_dimension n_dimmension>"
        << endl;
      MPI_Finalize();
      exit(1);
    }
  }
  else {
    if (argc == 2) {
      *dim_l = *dim_n = *dim_m = atoi(argv[1]);
    }
    else {
      *dim_l = atoi(argv[1]);
      *dim_m = atoi(argv[2]);
      *dim_n = atoi(argv[3]);
    }
  }
  if (rank == 0)
  {
    if (*dim_l <= 0 || *dim_n <= 0 || *dim_m <= 0) {
      cout << "Error: number of rows and/or columns must be greater than 0"
        << endl;
      MPI_Finalize();
```

```
      exit(1);
    }
  }
}

/*
   Routine that fills the number matrix with Random Data with values
   between 0 and MAX_VALUE
   This simulates in some way what might happen if there was a
   single sequential data acquisition source such as a single file
*/
void fill_matrix(float *array, int dim_m, int dim_n)
{
  int i, j;
  for (i = 0; i < dim_m; i++) {
    for (j = 0; j < dim_n; j++) {
      array[i*dim_n + j] = drand48()*MAX_VALUE;
    }
  }
}

/*
   Routine that outputs the matrices to the screen
*/
void print_matrix(float *array, int dim_m, int dim_n)
{
  int i, j;
  for (i = 0; i < dim_m; i++) {
    for (j = 0; j < dim_n; j++) {
      cout << array[i*dim_n + j] << " ";
    }
    cout << endl;
  }
}

// Routine to get the base number of multiplies for a ceratin process
int getBaseMult(int num_mults, int numtasks, int rank)
{
  // Calculate the base number of multiplies for each task
  int base = num_mults / numtasks;

  // Calculate the extra if it is not an even distribution
  int extra = num_mults % numtasks;

  // If there are any extra assign them to the first tasks up to rank of extra
  if (extra != 0)
  {
    // If rank is less than the number of extra items, then this process gets
an extra multiply to process
    if (rank < extra)
      base = (num_mults / numtasks) + 1;
  }

  return base;
}

/* ONE-TO-ALL SCATTER ROUTINE
Routine to divide the number of rows to each process based on the number
of multiplies that each process must perform.  Each process will receive the
row up to number of multiplies in a sequential order.  This method will also
keep up
```

```
with the current column slider and pass that to the process so it knows where
to
start it's set of multiplications.
The partial arrays for each process are stored in the group_a array.
*/
void scatter(float* a, float* b, float *group_a, int root, int rank, int
numtasks,
  int dim_l, int dim_m, int dim_n, int* start_column)
{
  MPI_Status status;
  int type = 234;

  // How many multiplies will the entire operation require?
  int num_mults = 0;
  if (dim_l == 1)
    num_mults = dim_n;
  else if (dim_n == 1)
    num_mults = dim_l;
  else
    num_mults = dim_l * dim_n;

  // Variables to keep up with what row and column we are reading from
  int begin_row = 0;
  int begin_column = 0;

  // Root process does all of the work
  if (rank == root)
  {
    // Loop over the MPI tasks
    for (int mpi_task = 0; mpi_task < numtasks; mpi_task++)
    {
      // Get the number of multiplies
      int base = getBaseMult(num_mults, numtasks, mpi_task);

      // Each task gets at least one row to work with
      int num_rows = 1;

      // Variables to keep up with navigating the matrices
      int count = base;
      int curr_col = begin_column;

      // Calculate the number of rows that we need to send to the process.
      while (count > 0)
      {
        // Get the current distance from the end of dim_n
        int diff = dim_n - curr_col;

        // If we have more multiplies to process then left for this row, we
must add another row
        if (count > diff)
        {
          num_rows++;
          count -= diff;
          curr_col = 0;
          continue;
        }

        count -= dim_n;
      }

      // If the current loop iteration doesn't correspond to the root then we
must send the column and number of rows to the process
```

```
      if (mpi_task != root)
      {
        MPI_Send(&begin_column, 1, MPI_INT, mpi_task, type, MPI_COMM_WORLD); //
send the column slider location
        MPI_Send(&num_rows, 1, MPI_INT, mpi_task, type, MPI_COMM_WORLD); //
send the number of rows for correct sizing
      }

      // Local Buffer variables
      float* local_a = new float[num_rows*dim_m];

      // Reset our column check and multiply count
      curr_col = begin_column;
      count = base;

      // Row assignment
      for (int r = 0; r < num_rows; r++)
      {
        // Fill up the local matrix with values from the main A matrix
        for (int t = 0; t < dim_m; t++)
        {
          // If this is the root, go ahead and store it in the buffer
          if (mpi_task == root)
          {
            group_a[r*dim_m + t] = a[begin_row*dim_m + t];
          }
          else // Store it in the local that will be sent to the other
processes
          {
            local_a[r*dim_m + t] = a[begin_row*dim_m + t];
          }
        }

        // Calculate the distance from the end of dim_n for this set of row
multiplications
        int diff = dim_n - curr_col;

        // If we still have more to process we must add the next row to this
processes variables
        if (diff <= count)
        {
          begin_row++;
          count -= diff;
          curr_col = 0;
        }
      }

      // Column Assignment
      // This logic will keep up with what column each process should start
performing calculations
      // Base is the updated base number of multiplies each process must
perform
      int temp_column = base % dim_n; // Get the leftover after applying number
of multiplies
      begin_column += temp_column; // Update the current column index
      if (begin_column >= dim_n) // Account for wrapping around dim_n
        begin_column = begin_column % dim_n;

      // Send the data to the other processes
      if (mpi_task != root)
      {
```

```
        MPI_Send(local_a, num_rows*dim_m, MPI_FLOAT, mpi_task, type,
MPI_COMM_WORLD);
      }

      delete[] local_a;
    }
  }
  else
  {
    // Calculate the base number of multiplies for each task
    int base = getBaseMult(num_mults, numtasks, rank);

    // I must send the number of rows as well
    int num_rows = 0;

    // Receive smaller arrays as well as the starting dim_n column from the
root process
    MPI_Recv(start_column, 1, MPI_INT, root, type, MPI_COMM_WORLD, &status);
    MPI_Recv(&num_rows, 1, MPI_INT, root, type, MPI_COMM_WORLD, &status);
    MPI_Recv(group_a, num_rows*dim_m, MPI_FLOAT, root, type, MPI_COMM_WORLD,
&status);
  }
}

// All to one gather routine
// Each process will send their calculated sub matrix back to the root process
void gather(float* c, float* group_c, int num_mults, int root, int rank, int
numtasks, int dim_l, int dim_n)
{
  MPI_Status status;
  int type = 123;

  // Calculate the base number of multiplies for each task
  int base = getBaseMult(num_mults, numtasks, rank);

  if (rank == root)
  {
    int curr_ind = 0;
    // Piece back together the matrix
    for (int mpi_task = 0; mpi_task < numtasks; mpi_task++)
    {
      if (mpi_task == root)
      {
        for (int i = 0; i < base; i++)
        {
          // Copy what the root has
          c[curr_ind] = group_c[i];
          curr_ind++;
        }
      }
      else
      {
        // Receive from the processes

        // Calculate the base number of multiplies for each task
        int base = getBaseMult(num_mults, numtasks, mpi_task);

        float* temp = new float[base];

        MPI_Recv(temp, base, MPI_FLOAT, mpi_task, type, MPI_COMM_WORLD,
&status);
```

```
        for (int i = 0; i < base; i++)
        {
          //cout << "Group_c Item " << temp[i] << endl;
          c[curr_ind] = temp[i];
          curr_ind++;
        }

        delete[] temp;
      }
    }
  }
  else
  {
    // Send the matrix to the root
    MPI_Send(group_c, base, MPI_FLOAT, root, type, MPI_COMM_WORLD);
  }
}

/*
   MAIN ROUTINE: summation of a number list
*/

int main(int argc, char *argv[])
{
  float dot_prod;
  int dim_l, dim_n, dim_m;
  int i, j, k;

  int num_mults, group_size, num_group;
  int numtasks, rank, num;
  int start_column;
  MPI_Status status;

  // Main Routine

  MPI_Init(&argc, &argv); // initalize MPI environment
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks); // get total number of MPI
processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get unique task id number


  // get matrix sizes
  get_index_size(argc, argv, &dim_l, &dim_m, &dim_n, rank);

  // The root process fills the matrices and then passes them to the othe
processes
  if (rank == 0)
  {
    // dynamically allocate from heap the numbers in the memory space
    // for the a,b, and c matrices
    a = new (nothrow) float[dim_l*dim_m];
    b = new (nothrow) float[dim_m*dim_n];
    c = new (nothrow) float[dim_l*dim_n];
    if (a == 0 || b == 0 || c == 0)
    {
      cout << "ERROR:  Insufficient Memory 1" << endl;
      MPI_Abort(MPI_COMM_WORLD, 1);
    }


    /*
       initialize numbers matrix with random data
```

```
    */
    srand48(SEED);
    fill_matrix(a, dim_l, dim_m);
    fill_matrix(b, dim_m, dim_n);

    /*
      output numbers matrix
    */
    //cout << "A matrix =" << endl;
    //print_matrix(a, dim_l, dim_m);
    //cout << endl;

    //cout << "B matrix =" << endl;
    //print_matrix(b, dim_m, dim_n);
    //cout << endl;
  }
  else
  {
    b = new (nothrow) float[dim_m*dim_n];
  }

  // Start recording the execution time
  if (rank == 0)
  {
    TIMER_CLEAR;
    TIMER_START;
  }
  MPI_Bcast(b, dim_m*dim_n, MPI_FLOAT, 0, MPI_COMM_WORLD);

  // broad cast the data size, which is really the number of multiplies
  if (dim_l == 1)
    num_mults = dim_n;
  else if (dim_n == 1)
    num_mults = dim_l;
  else
    num_mults = dim_l * dim_n;

  int base = getBaseMult(num_mults, numtasks, rank);

  // Each process has a local array set for local calculations
  group_a = new (nothrow) float[dim_l*dim_m];
  group_c = new (nothrow) float[dim_l*dim_n];


  start_column = 0;

  if (group_a == 0 || group_c == 0)
  {
    cout << "ERROR:  Insufficient Memory 2" << endl;
    MPI_Abort(MPI_COMM_WORLD, 1);
  }

  // Scatter the Data
  // The root process needs to scatter the correct amount of data to each
process.
  scatter(a, b, group_a, 0, rank, numtasks, dim_l, dim_m, dim_n,
&start_column);

  // Each process will start working on the data here
  int startIndex = 0;
  int row = 0;
  int col = start_column;
```

```
  for(int i = 0; i < base; i++)
  {
    group_c[i] = 0;
    for (int j = 0; j < dim_m; j++)
    {
      group_c[i] += group_a[dim_m*row + j] * b[j*dim_n + col];
    }

    // Keep up with the column so we know when to bump the row
    if (start_column != 0 && (start_column % (dim_n - 1) == 0))
    {
      start_column = 0;
      row++;
    }
    else if ((start_column + 1) != dim_n) // can't exceed the dim_n for current
column
    {
      start_column++;
    }

    col = start_column;
  }

  // Gather
  gather(c, group_c, num_mults, 0, rank, numtasks, dim_l, dim_n);

  /*
  Start recording the execution time
  */
 // TIMER_CLEAR;
 // TIMER_START;

  /*
     stop recording the execution time
  */
  //TIMER_STOP;

  if (rank == 0)
  {
    TIMER_STOP;
    //cout << "C matrix =" << endl;
    //print_matrix(c, dim_l, dim_n);
    //cout << endl;

    cout << "time=" << setprecision(8) <<  TIMER_ELAPSED/1000000.0
         << " seconds" << endl;
  }

 // if (rank == 0)
   // cout << "Made it to the cleanup" << endl;

  // Clear out memory
  if (rank == 0)
  {
    delete[] a;
    delete[] b;
    delete[] c;
  }

  delete[] group_a;
  delete[] group_c;
```

```
  // Terminate MPI Program -- perform necessary MPI housekeeping
  // clear out all buffers, remove handlers, etc.
  MPI_Finalize();
}
```

C) RUN TIME CHARACTERISTICS ON ONE GRAPH

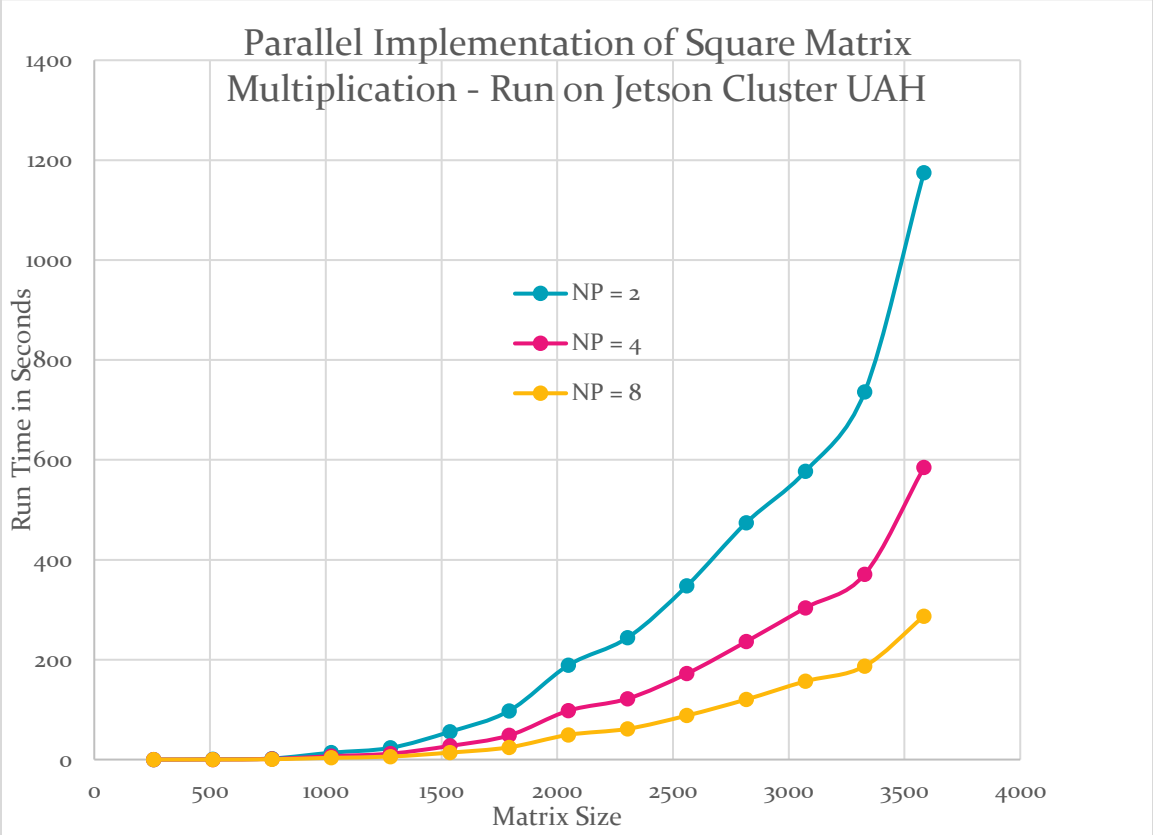

Figure 6. Parallel Matrix Multiplication Run Time Characteristics, NP(2, 4, 8)