Parallel Matrix Multiplication Study

John Nehls

10/11/15

Note: This project may be found on GitHub at https://github.com/JohnNehls/ParallelMatrixMultiplication

Contents

1 Introduction

This document will serve as a comparison of serial, OpenMP parallel, and MPI parallel code that accomplishes the same task: matrix multiplication. Along with comparing the total matrix multiplication times of the codes, we will look at the ratio of time spent calculating the multiplication to the time the parallel tool spends communicating data. Then, we attempt to explain why the parallelization schemes scale the way they do.

2 Parallel Matrix Multiplication

The scheme used to distribute the matrix multiplication of matrices A and B

$$AB = C \tag{1}$$

is the same approach as the one displayed in Lecture 2 of the course. Thus, to offer a brief summary of the approach, we will use slightly modified figures of ones shown in class (which seem to be from an excellent introduction to parallel computing [1])

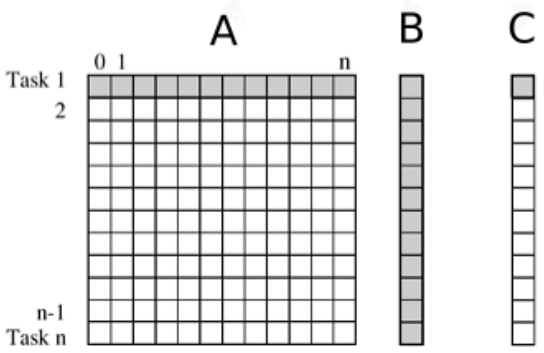

Figure 1:

In figure 1 we see how the matrix multiplication is split across n tasks. Task 1 computes the the first element of C using the information in dark gray: the first row of A and all of matrix B. This is a general way to see the how the computation may be split in to n tasks. In practice often each task will calculate more than one quantity, this is the concept of granularity. In figure 2 we see how the multiplication of equation 1 is split over 4 tasks.

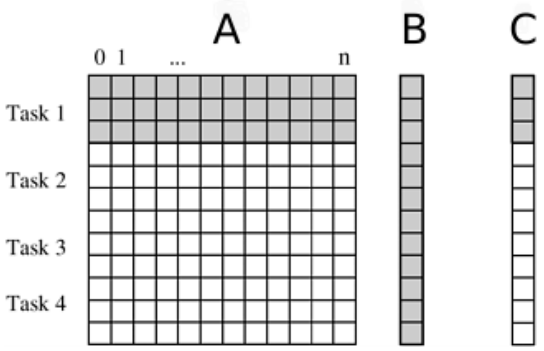

Figure 2:

The parallelization scheme used in our MPI and OpenMP codes is the same as shown here: each task sees one or more rows of A, all of B, and calculates a subset of C. For a more detailed treatment, I encourage you to check out the book or visit the website hyper linked [1].

3 Results

3.1 Details of the Experiments

To carry out the comparisons of serial to OpenMP to MPI, we calculated the multiplication of random-valued, square matrices of dimensions 240x240 and 2400x2400. The parallel codes computed these matrices using 2, 4, 8, 16, 32, 64, and 128 parallel processes. Each result was averaged over 10 runs. The Arizona Center of Mathematical Sciences at the University of Arizona provided me with time on a Silocon Graphics (SGI) UV2000 which contains, after recent upgrades, 128 Intel Xeon E5-4617 "Sandy Bridge" 6 core Processors at 2.9GHz[2].

3.2 Total Matrix Calculation Time

First, to understand the total matrix multiplication timing data, we look at the postcode for the serial, OpenMP, and MPI total matrix multiplication times. The important detail in this code is that the creation and filling of the matrices is excluded from the timing.

```
1  int main(int argc, char *argv[]) {
2    A,B = CreateMatrices();        /* Excluded from timing!!!  */
3    FillMatricesRandomly(A, B);    /* Excluded from timing!!!  */
4
5    start = currentTime()
6
7    // Matrix Multiplication
8
9    end = currentTime()
10   double matrixCalculationTime = end - start;
11
12   return 0;
13 }
```

3.2.1 Results

To get the point, lets look at the total multiplication time results.

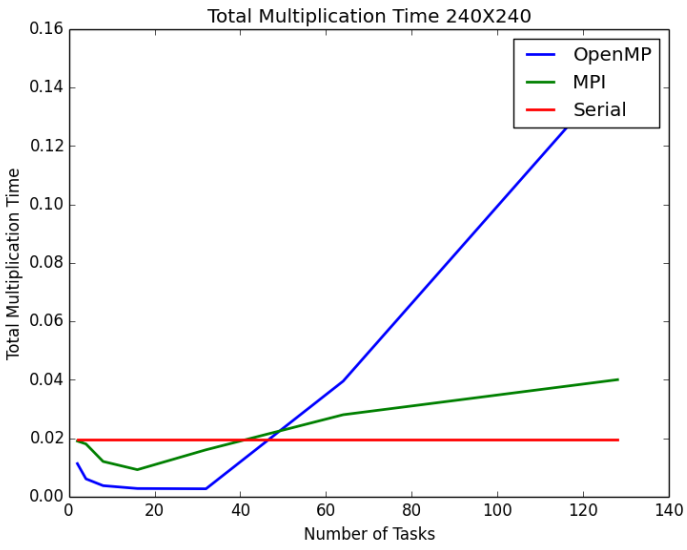

Figure 3:

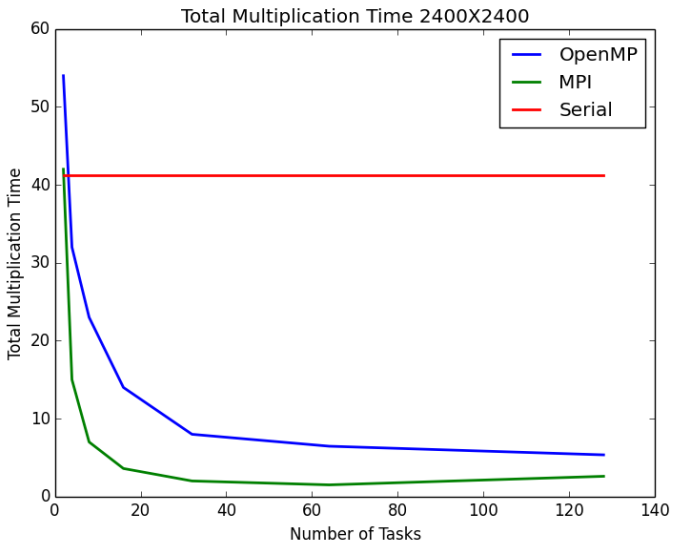

Figure 4:

Now to explain the behavior, we will us Amdahl's Law. Amdahl's Law states

$$speedup = 1/(P/N + S) \tag{2}$$

where P = parallel fraction, N = number of processors and S = serial fraction [3]. Extending this a little further, we find the total multiplication time

$$totaltime = time0/speedup = time0 \times (P/N + S), \tag{3}$$

where $time0$ is the amount of time it takes for the serial code to execute the task at hand.

To explain the dependence of total matrix multiplication time on the number of tasks, we plot Amdahl's Law in the form shown in equation 3 with a parallel fraction of 1 in figures 5 and 6, showing the ideal case, which, it should be said, should be the correct parallel fraction since no serial actions are timed.

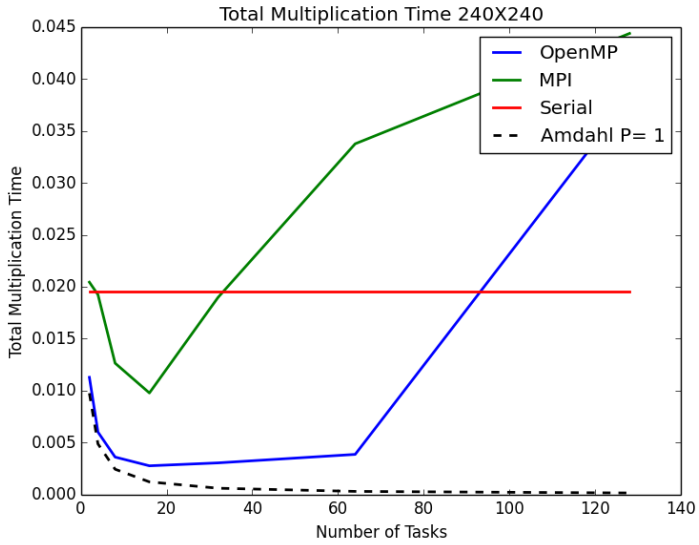

Figure 5:

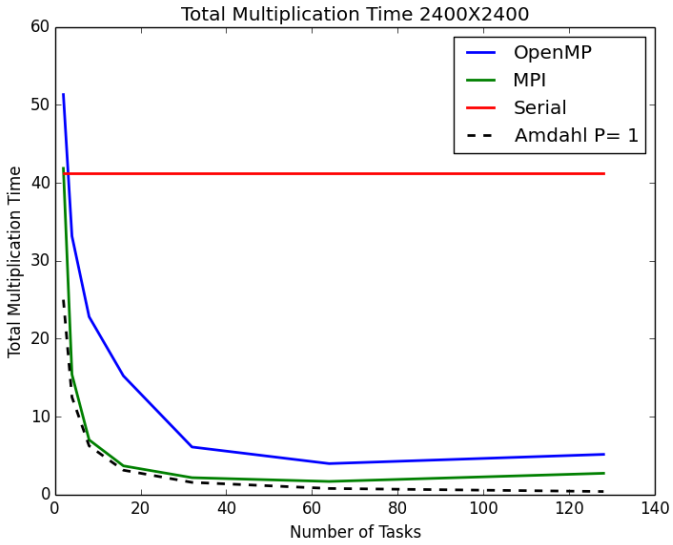

Figure 6:

First of all we see that in figure 5 the experiment is very different from the theory, not even displaying the same shape, where as, for computation of the larger matrices, figure 6, the theory predicts the behavior of the experiment well. The simple explanation of this is that the slaves(or threads) in the 240x240 were spending more time communicating than computing, which, when brought to the extreme, here a granularity of about 50 rows, the computation begins to take longer in parallel than it did in for the serial code. For the 2400x2400 case, this extreme saturation did not appear. We will visit this again after we look at the ratio of computation over communication times in the next results section, which should confirm or deny this explanation for the far from theoretical result in figure 5.

3.3 Ratio of Calculation Time Over Communication Time

Here we calculate the ratio of the computation time over the communication time by assuming

$$T_{total} = T_{comp} + T_{comm}, \tag{4}$$

where T_{total} is the total matrix multiplication time, T_{comp} is the compuation time, and T_{comm} is the communication time of the parrallel tool, to be true. If so, we can say that we can find the communication time by

$$T_{comm} = T_{total} - T_{comp}. \tag{5}$$

So how do we aquire the compuation time? Here we provide the pseudo code for aquiring the computation time from the MPI code.

```
1  int main(int argc, char *argv[]) {
2    /* initialize MPI */
3    A,B = CreateMatrices();       /* Excluded from timing!!!  */
4
5    if (rank == 0) {  /* Master initializes work*/
6      FillMatricesRandomly( A, B);        /* Excluded from timing!!!  */
7
8      //send data to each slave needs to caclulate its part of the workload
9      MPI_Isend(&informationToSlave);
10   }
11
12   if (rank > 0) {   /* work done by slaves (not rank = 0)*/
13     //recieve data from master to caclulate part of the workload
14     MPI_Recv(&information);
15
16       //FOR THE COMPUTATIONAL TIME: start time for local time: the amount of time to do matrix
             calculation for this process
17     localTimeSaver = MPI_Wtime();
18
19     /* DO Matrix Multiplication-- Loops that do the computation  */
20
21     //FOR THE COMPUTATIONAL TIME:calculate local time: the amount of time to do matrix
         calculation for this process
22     localTimeSaver = MPI_Wtime() - localTimeSaver; // caclulates the time spent on calculation
23
24     //send back the matrix calc data to master
25     MPI_Isend(&calulatedMatrixInfo);
26
27     //FOR THE COMPUTATIONAL TIME:localTimeSaver to master
28     MPI_Isend(&localTimeSaver);)
29   }
30   if (rank == 0) { /* master gathers processed work*/
31
32     //receive matrix data from slaves
33     MPI_Recv(&CalculatedMatrixInformation);
34
35     //FOR THE COMPUTATIONAL TIME
36     // find the longest local calulation (which we take as the total amount of calculation time,
         the rest comming from communication.
37     MPI_Recv(&localComputationTimeForEachSlave);
38     double maxLocalMultiplicationTime = Max(localComputationTimeForEachSlave)
39   }
40
41   return 0;
42 }
```

Here we provide the pseudo code for acquiring the average computation time from the OpenMP code. I am unsure if this is an accurate measure, but it is my best guess on how to do the timing, so I decided to in this report. Note, again, that the average computation time is found, not the computation time.

```
int main(int argc, char *argv[]) {
  A,B = CreateMatrices();        /* Excluded from timing!!!  */
  FillMatricesRandomly(A, B);    /* Excluded from timing!!!  */

  omp_set_num_threads(numThreads); // set the number of threads

  // Matrix Multiplication
#pragma omp parallel for private(val) reduction(+:sum)
  for (int i = 0; i < A.rows(); i++) {//iterate through rows of A (parallelized loop)

    val = omp_get_wtime(); // for each thread get Start time

    for (int j = 0; j < B.cols(); j++) {//iterate through columns of B
      for (int k = 0; k < B.rows(); k++) {//iterate through rows of B
        C(i,j) += (A(i,k) * B(k,j));
      }
    }
    sum +=  omp_get_wtime() - val;//add up ALL OF THE COMPUTATION TIMES
  }
  AverageComputationTime = sum/numberOfThreads

  return 0;
}
```

Now that we have the the ratio of the computation time over the communication time (comp/comm), we can look at the results in figures 7 and 8 and see that our intuition was true from the previous results section when trying to describe why the the total multiplication time was so large for the 240x240 parallel runs. The (comp/comm) becomes much less than one for both MPI and OpenMPfor the 240x240, and, in the 2400x2400 case, only just approaches one, keeping it near Amdahl's Law equation 2.

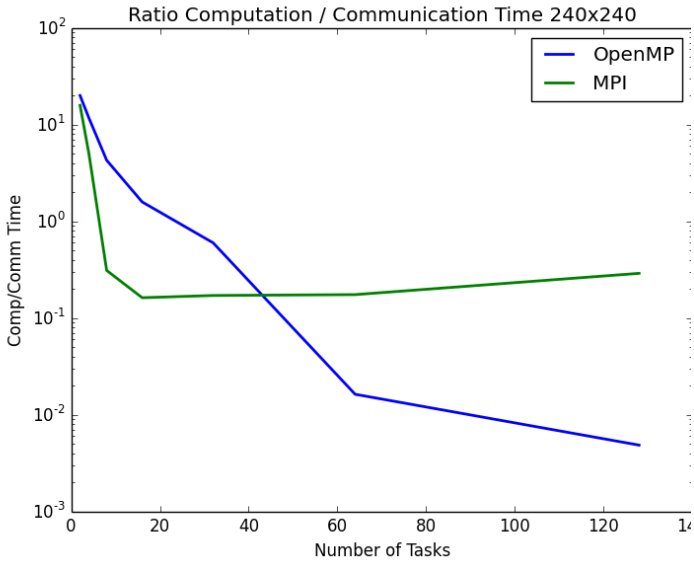

Figure 7:

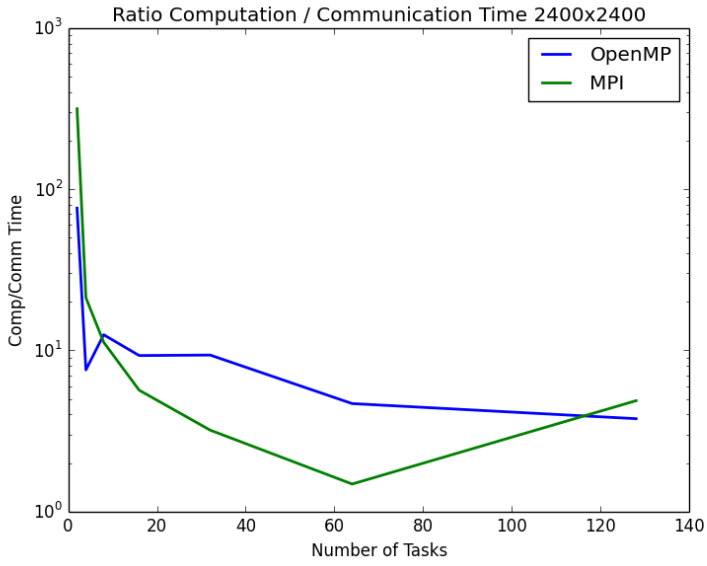

Figure 8:

Appendices

A Serial Code

```
#include<stdio.h>
#include<cstdlib>
#include<time.h>
#include<iostream>
#include<matrix.h>

/* Function Prototypes */
void FillMatricesRandomly(Matrix<double> &A, Matrix<double> &B );
void PrintMatrices( Matrix<double> &A, Matrix<double> &B, Matrix<double> &C );

/* Global variables that could be inputs #TODO */
int randomHigh = 100;           /* the upper bound of the random numbers that fill A and B*/
int randomLow = 0;              /* the lower bound of the random numbers that fill A and B*/

int main(int argc, char *argv[]) {
    std::cout << "Starting a serial matrix multiplication. \n  " << std::endl;
  // Read in the input:  N
  if ( argv[1]== NULL ){ // check input is supplied
    std::cout << "ERROR: The program must be executed in the following way  \n\n  \t \"./serial.
        exe N \"  \n\n where N is an integer. \n \n " << std::endl;
    return 1;
  }
  int N = atoi(argv[1]);  // The dimensions of the matrices MUST be specified at runtime.
  std::cout << "The matrices are: " << N<<"x"<<N<< std::endl;
  // for simplicity, all matricies will be NxN
  int numberOfRowsA = N;  int numberOfColsA = N;  int numberOfRowsB = N;  int numberOfColsB = N;

  //Declare matrices: Matix class is a 2D vetor
  Matrix<double> A = Matrix<double>(numberOfRowsA, numberOfColsA);
  Matrix<double> B = Matrix<double>(numberOfRowsB, numberOfColsB);
  Matrix<double> C = Matrix<double>(numberOfRowsA, numberOfColsB);

  FillMatricesRandomly(A, B);   /* Excluded from timing!!!  */

  struct timespec start, end;
  clock_gettime(CLOCK_MONOTONIC, &start); // start timing

  // Matrix Multiplication
  for (int i = 0; i < A.rows(); i++) {//iterate through rows of A
    for (int j = 0; j < B.cols(); j++) {//iterate through columns of B
      for (int k = 0; k < B.rows(); k++) {//iterate through rows of B
        C(i,j) += (A(i,k) * B(k,j));
      }
    }
  }

  clock_gettime(CLOCK_MONOTONIC, &end); // end timing
  double matrixCalculationTime = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec)*1e
      -9;
  std::cout << "\nTotal multplication time = " << matrixCalculationTime << std::endl;
//   PrintMatrices(A, B, C);
  return 0;
}

void FillMatricesRandomly(Matrix<double> &A, Matrix<double> &B){
/* initialize the random number generator with the current time */
  srand( time( NULL ));
  for (int i = 0; i < A.rows(); i++) {
    for (int j = 0; j < A.cols(); j++) {
      A(i,j) = rand() % (randomHigh - randomLow) + randomLow;
    }
  }
  for (int i = 0; i < B.rows(); i++) {
    for (int j = 0; j < B.cols(); j++) {
      B(i,j) = rand() % (randomHigh - randomLow) + randomLow;
    }
  }
}
```

```
67
68
69  void PrintMatrices(Matrix<double> &A, Matrix<double> &B, Matrix<double> &C){
70    for (int i = 0; i < A.rows(); i++) {
71      std::cout <<"\n"<<std::endl;
72      for (int j = 0; j < A.cols(); j++)
73        std::cout << A(i,j) << " ";
74    }
75      std::cout <<"\n\n"<<std::endl;
76    for (int i = 0; i < B.rows(); i++) {
77      std::cout <<"\n"<<std::endl;
78      for (int j = 0; j < B.cols(); j++)
79        std::cout << B(i,j) << " ";
80    }
81    std::cout <<"\n\n"<<std::endl;
82    for (int i = 0; i < C.rows(); i++) {
83      std::cout <<"\n"<<std::endl;
84      for (int j = 0; j < C.cols(); j++)
85        std::cout << C(i,j) << " ";
86    }
87    std::cout <<"\n\n"<<std::endl;
88  }
```

B OpenMP Code

```
#include<cstdlib>
#include<time.h>
#include<iostream>
#include<matrix.h>
#include<omp.h>


/* Function Prototypes */
void FillMatricesRandomly(Matrix<double> &A, Matrix<double> &B );
void PrintMatrices( Matrix<double> &A, Matrix<double> &B, Matrix<double> &C );

/* Global variables that could be inputs #TODO */
int randomHigh = 100;           /* the upper bound of the random numbers that fill A and B*/
int randomLow = 0;              /* the lower bound of the random numbers that fill A and B*/

int main(int argc, char *argv[]) {
    std::cout << "Starting an OpenMP parallel matrix multiplication. \n  " << std::endl;
  // Read in the two inputs: NumberOfOMPthreads and N
  if ( argv[1]== NULL || argv[2] == NULL){ // check if inputs were supplied
    std::cout << "ERROR: The program must be executed in the following way  \n\n  \t \"./omp.exe
        NumberOfThreads N \"  \n\n where NuberOfThreads and N are integers. \n \n " << std::endl;
    return 1;
  }
  int numThreads = atoi(argv[1]);
  std::cout << "The number of OpenMP threads: " << numThreads << std::endl;
  omp_set_dynamic(0);           // do not allow the number of threads to be set internally
  omp_set_num_threads(numThreads); // set the number of threads

  int N = atoi(argv[2]);  // The dimensions of the matrices MUST be specified at runtime.
  std::cout << "The matrices are: " << N<<"x"<<N<< std::endl;
  // for simplicity, all matricies will be NxN
  int numberOfRowsA = N;  int numberOfColsA = N;  int numberOfRowsB = N;  int numberOfColsB = N;
  //Declare matrices: Matix class is a 2D vetor
  Matrix<double> A = Matrix<double>(numberOfRowsA, numberOfColsA);
  Matrix<double> B = Matrix<double>(numberOfRowsB, numberOfColsB);
  Matrix<double> C = Matrix<double>(numberOfRowsA, numberOfColsB);

  FillMatricesRandomly(A, B);   /* Excluded from timing!!!  */

  struct timespec start, end;
  clock_gettime(CLOCK_MONOTONIC, &start); // start timing

  // Used to calculate the longest a process spends calculating its part of the workload.
  // double sumLocalTime = 0;
  double sum = 0;
  double val = 0;
  // Matrix Multiplication
#pragma omp parallel for private(val) reduction(+:sum)
  for (int i = 0; i < A.rows(); i++) {//iterate through rows of A (parallelized loop)
    val = omp_get_wtime();
    for (int j = 0; j < B.cols(); j++) {//iterate through columns of B
      for (int k = 0; k < B.rows(); k++) {//iterate through rows of B
        C(i,j) += (A(i,k) * B(k,j));
      }
    }
    sum +=  omp_get_wtime() - val;
  }

  clock_gettime(CLOCK_MONOTONIC, &end); // end timing
  double totalMatrixCalculationTime = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec)
      *1e-9;
  std::cout << "Total multplication time = " << totalMatrixCalculationTime << std::endl;
  std::cout << "average multplication time = " << sum/numThreads << std::endl;
  std::cout << "Approximate Communication time = " << totalMatrixCalculationTime - sum/numThreads
       << std::endl;
//   PrintMatrices(A, B, C);
  return 0;
}

void FillMatricesRandomly(Matrix<double> &A, Matrix<double> &B){
/* initialize the random number generator with the current time */
```

```
69    srand( time( NULL ));
70    for (int i = 0; i < A.rows(); i++) {
71      for (int j = 0; j < A.cols(); j++) {
72        A(i,j) = rand() % (randomHigh - randomLow) + randomLow;
73      }
74    }
75    for (int i = 0; i < B.rows(); i++) {
76      for (int j = 0; j < B.cols(); j++) {
77        B(i,j) = rand() % (randomHigh - randomLow) + randomLow;
78      }
79    }
80  }
81
82  void PrintMatrices(Matrix<double> &A, Matrix<double> &B, Matrix<double> &C){
83    for (int i = 0; i < A.rows(); i++) {
84      std::cout <<"\n"<<std::endl;
85      for (int j = 0; j < A.cols(); j++)
86        std::cout << A(i,j) << " ";
87    }
88      std::cout <<"\n\n"<<std::endl;
89    for (int i = 0; i < B.rows(); i++) {
90      std::cout <<"\n"<<std::endl;
91      for (int j = 0; j < B.cols(); j++)
92        std::cout << B(i,j) << " ";
93    }
94    std::cout <<"\n\n"<<std::endl;
95    for (int i = 0; i < C.rows(); i++) {
96      std::cout <<"\n"<<std::endl;
97      for (int j = 0; j < C.cols(); j++)
98        std::cout << C(i,j) << " ";
99    }
100   std::cout <<"\n\n"<<std::endl;
101
102 }
```

C MPI Code

```
#include<cstdlib>
#include<ctime>
#include<mpi.h>
#include<algorithm>
#include<matrix.h>

// Function Prototypes
void FillMatricesRandomly(Matrix<double> &A, Matrix<double> &B );
void PrintMatrices( Matrix<double> &A, Matrix<double> &B, Matrix<double> &C );

/* MPI Send and Recieve Tags */
#define ROW_START_TAG 0 //tag for communicating the start row of the workload for a slave
#define ROW_END_TAG 1   //tag for communicating the end row of the workload for a slave
#define A_ROWS_TAG 2    //tag for communicating the address of the data to be worked on to slave
#define C_ROWS_TAG 3    //tag for communicating the address of the calculated data to master
#define LOCAL_TIME_TAG 4//tag for communicating the address of the local matrix calculation time
    to master

// Instantiate global variables used in the parallelization
int rank;                   // mpi: process id number
int nProcesses;             // mpi: number of total processess
MPI_Status status;          // mpi: store status of a MPI_Recv
MPI_Request request;        // mpi: capture request of a MPI_Isend
int rowStart, rowEnd;       // which rows of A that are calculated by the slave process
int granularity;            // granularity of parallelization (# of rows per processor)

//Used to calculate totalmultiplication time: communication + calculations
double start_time, end_time;
// Used to calculate the longest a process spends calculating its part of the workload.
double localTimeSaver;

/* Global variables that could be inputs #TODO */
int randomHigh = 100;     // the upper bound of the random numbers that fill A and B
int randomLow = 0;        // the lower bound of the random numbers that fill A and B

int main(int argc, char *argv[]) {
  // ***** Handle the input size of the Matrices, N, where matrices A, B, and C will be NxN *****
  if ( argv[1]== NULL ){// check if the input was supplied
   std::cout << "ERROR: The program must be executed in the following way  \n\n  \t \"mpirun -n
       NumberOfProcesses mpi.exe N \"  \n\n where N is an integer. \n \n " << std::endl;
   return 1;
  }
  int N = atoi(argv[1]);  // The dimensions of the matrices MUST be specified at runtime.
  // for simplicity, all matricies will be NxN
  int numberOfRowsA = N;  int numberOfColsA = N;  int numberOfRowsB = N;  int numberOfColsB = N;
  //Declare matrices: Matix class is a 2D vetor
  Matrix<double> A = Matrix<double>(numberOfRowsA, numberOfColsA);
  Matrix<double> B = Matrix<double>(numberOfRowsB, numberOfColsB);
  Matrix<double> C = Matrix<double>(numberOfRowsA, numberOfColsB);

  // MPI:
  MPI_Init(&argc, &argv);                     /* initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);       /* store the rank */
  MPI_Comm_size(MPI_COMM_WORLD, &nProcesses); /* store the number of processes */

  if (rank == 0) {  /* Master initializes work*/
    std::cout << "Starting an MPI parallel matrix multiplication. \n  " << std::endl;
    std::cout << "The matrices are: " << N<<"x"<<N<< std::endl;
    FillMatricesRandomly( A, B);        /* Excluded from timing!!!  */

    /* Begin Timing: used for total multiplication time: communication + calculations  */
    start_time = MPI_Wtime();
    for (int i = 1; i < nProcesses; i++) { /* for each slave */
      // calculate granularity (-1 comes from excluding the master process)
      granularity = (numberOfRowsA / (nProcesses - 1));
      rowStart = (i - 1) * granularity;
      if (((i + 1) == nProcesses) && ((numberOfRowsA % (nProcesses - 1)) != 0)) {//if rows of [A]
          cannot be equally divided among slaves
        rowEnd = numberOfRowsA; //last slave gets all the remaining rows
      } else {
        rowEnd = rowStart + granularity; //rows of [A] are equally divisable among slaves
```

```
69          }
70          //send the low bound, without blocking, to the intended slave
71          MPI_Isend(&rowStart, 1, MPI_INT, i, ROW_END_TAG, MPI_COMM_WORLD, &request);
72          //next send the upper bound without blocking, to the intended slave
73          MPI_Isend(&rowEnd, 1, MPI_INT, i ,ROW_START_TAG, MPI_COMM_WORLD, &request);
74          //finally send the allocated row granularity of [A] without blocking, to the intended slave
75          MPI_Isend(&A(rowStart,0), (rowEnd - rowStart) * numberOfColsA, MPI_DOUBLE, i, A_ROWS_TAG,
                MPI_COMM_WORLD, &request);
76        }
77      }
78      //broadcast B (MPI_Bcast: Broadcasts a message from the process with rank "root" to all other
            processes of the communicator)
79      MPI_Bcast(&B(0,0), numberOfRowsB*numberOfColsB, MPI_DOUBLE, 0, MPI_COMM_WORLD);
80
81      if (rank > 0) {    /* work done by slaves (not rank = 0)*/
82        //receive low bound from the master
83        MPI_Recv(&rowStart, 1, MPI_INT, 0, ROW_END_TAG, MPI_COMM_WORLD, &status);
84        //next receive upper bound from the master
85        MPI_Recv(&rowEnd, 1, MPI_INT, 0, ROW_START_TAG, MPI_COMM_WORLD, &status);
86        //finally receive row granularity of [A] to be processed from the master
87        MPI_Recv(&A(rowStart,0), (rowEnd - rowStart) * numberOfColsA, MPI_DOUBLE, 0, A_ROWS_TAG,
              MPI_COMM_WORLD, &status);
88
89        // start time for local time: the amount of time to do matrix calculation for this process
90        localTimeSaver = MPI_Wtime();
91        /* Matrix Multiplication  */
92        for (int i = rowStart; i < rowEnd; i++) {//the given set of rows of A (parallelized loop)
93          for (int j = 0; j < B.cols(); j++) {//iterate through columns of [B]
94            for (int k = 0; k < B.rows(); k++) {//iterate through rows of [B]
95              C(i,j) += (A(i,k) * B(k,j));
96            }
97          }
98        }
99        // calculate local time: the amount of time to do matrix calculation for this process
100       localTimeSaver = MPI_Wtime() - localTimeSaver;
101
102       //send back the low bound first without blocking, to the master
103       MPI_Isend(&rowStart, 1, MPI_INT, 0, ROW_END_TAG, MPI_COMM_WORLD, &request);
104       //send the upper bound next without blocking, to the master
105       MPI_Isend(&rowEnd, 1, MPI_INT, 0, ROW_START_TAG, MPI_COMM_WORLD, &request);
106       //finally send the processed granularity of data without blocking, to the master
107       MPI_Isend(&C(rowStart,0), (rowEnd - rowStart) * numberOfColsB, MPI_DOUBLE, 0, C_ROWS_TAG,
              MPI_COMM_WORLD, &request);
108       //send back the local calculation time without blocking, to the master
109       MPI_Isend(&localTimeSaver, 1, MPI_DOUBLE, 0, LOCAL_TIME_TAG, MPI_COMM_WORLD, &request);
110
111
112     }
113
114     if (rank == 0) { /* master gathers processed work*/
115       for (int i = 1; i < nProcesses; i++) {// untill all slaves have handed back the processed
              data
116         //receive low bound from a slave
117         MPI_Recv(&rowStart, 1, MPI_INT, i, ROW_END_TAG, MPI_COMM_WORLD, &status);
118         //receive upper bound from a slave
119         MPI_Recv(&rowEnd, 1, MPI_INT, i, ROW_START_TAG, MPI_COMM_WORLD, &status);
120         // //receive processed data from a slave
121         MPI_Recv(&C(rowStart,0), (rowEnd - rowStart) * numberOfColsB, MPI_DOUBLE, i, C_ROWS_TAG,
                MPI_COMM_WORLD, &status);
122
123       }
124       end_time = MPI_Wtime(); //end time of the total matrix matrix multiplication
125       double totalMultiplicationTime = end_time - start_time;
126
127       // find the longest local calulation (which we take as the total amount of calculation time,
              the rest comming from communication.
128       std::vector<double> LocalMultiplicationTimes = std::vector<double>(nProcesses);
129       for (int i = 1; i < nProcesses; i++) {
130         MPI_Recv(&LocalMultiplicationTimes[i], 1, MPI_DOUBLE, i, LOCAL_TIME_TAG, MPI_COMM_WORLD, &
                status);
131       }
132       double maxLocalMultiplicationTime = *std::max_element(LocalMultiplicationTimes.begin(),
              LocalMultiplicationTimes.end());
133
```

```
134     // print out the results
135     std::cout <<"Total multiplication time =  " << totalMultiplicationTime <<"\n"<< std::endl;
136     std::cout <<"Longest multiplication time =  " << maxLocalMultiplicationTime  <<"\n"<< std::
            endl;
137     std::cout <<"Approximate communication time =  " << totalMultiplicationTime -
            maxLocalMultiplicationTime  <<"\n\n"<< std::endl;
138
139     // PrintMatrices(A, B, C);     // for debugging
140
141   }
142   MPI_Finalize(); //finalize MPI operations
143   return 0;
144 }
145
146 void FillMatricesRandomly(Matrix<double> &A, Matrix<double> &B){
147 /* initialize the random number generator with the current time */
148   srand( time( NULL ));
149   for (int i = 0; i < A.rows(); i++) {
150     for (int j = 0; j < A.cols(); j++) {
151       A(i,j) = rand() % (randomHigh - randomLow) + randomLow;
152     }
153   }
154   for (int i = 0; i < B.rows(); i++) {
155     for (int j = 0; j < B.cols(); j++) {
156       B(i,j) = rand() % (randomHigh - randomLow) + randomLow;
157     }
158   }
159 }
160
161
162 void PrintMatrices(Matrix<double> &A, Matrix<double> &B, Matrix<double> &C){
163   for (int i = 0; i < A.rows(); i++) {
164     std::cout <<"\n"<<std::endl;
165     for (int j = 0; j < A.cols(); j++)
166       std::cout << A(i,j) << " ";
167   }
168     std::cout <<"\n\n"<<std::endl;
169   for (int i = 0; i < B.rows(); i++) {
170     std::cout <<"\n"<<std::endl;
171     for (int j = 0; j < B.cols(); j++)
172       std::cout << B(i,j) << " ";
173   }
174   std::cout <<"\n\n"<<std::endl;
175   for (int i = 0; i < C.rows(); i++) {
176     std::cout <<"\n"<<std::endl;
177     for (int j = 0; j < C.cols(); j++)
178       std::cout << C(i,j) << " ";
179   }
180   std::cout <<"\n\n"<<std::endl;
181 }
```

D Matrix.h

```
#ifndef MATRIX_H_
#define MATRIX_H_

#include <vector>
#include <complex>

using std::vector;
using std::complex;

// simple wrapper of a vector
template <class T>
class Matrix {
public:
  // create an empty matrix
  Matrix(int numrows, int numcols)
    :Nrow(numrows), Ncol(numcols), elements(Nrow*Ncol) {}

  // construct it from existing data
  Matrix(int numrows, int numcols, T* data)
    :Nrow(numrows), Ncol(numcols), elements(data, data+numrows*numcols) {}

  int rows() {return Nrow;}
  int cols() {return Ncol;}

  // access to elements, col is the fast axis
  T operator() (int row, int col) const {return elements[Ncol*row + col];}
  T& operator() (int row, int col) {return elements[Ncol*row + col];}


  // get raw pointer to elements[0]
  T* data() {return elements.data();}
  const vector<T>& elem() {return elements;}

private:
  int Nrow, Ncol;
  vector<T> elements;
};


#endif // MATRIX_H_
```

References

[1] Introduction to Parallel Computing." Second Edition - Grama, Gupta, et al. - 2003 Available online here: http://parallelcomp.uw.hu/ch03lev1sec1.html

[2] http://www.acms.arizona.edu/CompResources/index.html

[3] AFIPS '67 (Spring) Proceedings of the April 18-20, 1967, spring joint computer conference Pages 483-485