Partial Differential Equations
Course SCICOMP 9502B

Assignment 2

Matrix-Vector Computation

Instructor

Dr. Colin Denniston

Submitted by

Tuntun Gaurav

Chemical and Biochemical Engineering

1. Matrix-Vector multiplication

1. Description of Code:

The program is written for matrix-vector multiplication. I have defined two global static values namely pi and e with 16 digits of precision. A is defined dynamically as a 2D matrix with each row occupying a contiguous set of memory spaces and the rows being non-contiguous to each other, while b and c are 1D arrays dynamically initialized as a contiguous block of memory using calloc. The MPI is initialized and the rank and number of processors are taken in to the variable myid and nprocs respectively. The program is written so as to allow it to be generalized for any value of N, size of the array.

The if condition allows only processor 0 namely as the master processor to enter the first condition. First, A and b are initialized and b is broadcast to all the processors as an array of size ncols with double precision values sent from the master node to all the mpi_comm_world. This will send the b matrix to any processor listening for a matrix form the master processor. Further, the master processor sends out the rows of A to all the processor at hand as a 1D array of the next row "rowsent" to processor rowsent+1 till rowsent = nprocs-1. The "for" loop is started at the master processor to receive the results and process the remaining rows to the processors. Master processor listens for the results from the other processors. The sender and tag are saved in the to substitute the answer at the correct position in vector c. It has been assumed that the N>nprocs. Hence, the remaining rows are sent to the rest of the processors with an if condition till the end of number of rows. At the end of which a tag "0" is sent out with MPI_Botton to define the end of process.

The other part is for all the processors other than the master processor. The Bcast is used to listen for b matrix from the master processor. Then, again the processor listens for receiving a row of the matrix A. This will be the row with rowsent+1 value as the rank(myid) for this processor. Each processor receives the buff i.e. row of a with their rank +1. The while loop runs until we achieve the tag "0" value at the end of all the rows in A. The buff is multiplied to the buff which contain the present row of A, and the added giving us the result for the present row. And the code proceeds to sending the result back to the master processor. The master processor receives this answer and the status of the row to position the answer at the correct row value. The next row is sent to the processor for solving. It goes on until all the rows are passed at which time the master code passes tag "0" which ends the process for this processor. The result is printed and mpi_finalize cleans up all mpi state.

1. Sample Input/Output:
   First two tests were performed with Identity an inverse-identity matrix, to check if the program is working fine. Then a 16 digit precision value provided as static global variable is used for calculation and the multiplication is performed to test the accuracy of the code. A is given as a 6x6 matrix with each value given as pi*(i*j) where I and j are the present row and column. The b vector is composed of exponential powered I, where I is the row value. This gives a very high precision calculation, as required.
2. Description of results:
   The identity matrix multiplication gave back the b vector, hence confirming the correct calculation. The expected results given by the inverse matrix multiplication confirms the same.
   The results for the high precision values of A and b multiplication was performed using MATLAB. The results are shown. The method is observed to be accurate for lower decimals but difference is seen at higher values.
3. Main Results: The results are given as below.
   With Identity Matrix

A

1.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	1.000000

That took 0.000597 seconds for processor 2
b 1.000000 C 1.000000
b 2.718218 C 2.718218
b 7.388710 C 7.388710
b 20.084128 C 20.084128
b 54.593042 C 54.593042
b 148.395802 C 148.395802
That took 0.021302 seconds for processor 0
That took 0.029213 seconds for processor 3
That took 0.029326 seconds for processor 1

With Inverse Identity Matrix
A

0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
0.000000	0.000000	0.000000	0.000000	1.000000	0.000000
0.000000	0.000000	0.000000	1.000000	0.000000	0.000000
0.000000	0.000000	1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000	0.000000	0.000000

b 1.000000 C 0.000000
b 2.718218 C 148.395802
b 7.388710 C 54.593042
b 20.084128 C 20.084128
b 54.593042 C 7.388710
b 148.395802 C 2.718218
That took 0.000362 seconds for processor 0
That took 0.007950 seconds for processor 1
That took 0.000150 seconds for processor 2
That took 0.007609 seconds for processor 3

With Diaginal Matrix

A

0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	6.283185	0.000000	0.000000	0.000000	0.000000

0.000000 0.000000 12.566371 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 18.849556 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 25.132741 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000 31.415927
That took 0.000747 seconds for processor 2
That took 0.000145 seconds for processor 1
b 1.000000 C 0.000000
b 2.718218 C 17.079069
b 7.388710 C 92.849274
b 20.084128 C 378.576885
b 54.593042 C 1372.072794
b 148.395802 C 4661.991628
That took 0.000965 seconds for processor 0
That took 0.001797 seconds for processor 3

From Matlab
ans =

1.0e+03 *

0
0.017079468445347
0.092853617429454
0.378603451447757
1.372201176349754
4.662536903327078

4. Analysis: The results from the code were compared with those from Matlab for the same input data and they were matching to the third decimal point for the first answer for the same initial value of pi and exponential terms. However, as the values for calculation increased the accuracy decreased giving minimum for the final value of the result.

2. Matrix-Matrix multiplication

1. Code Description:

The program is written for matrix-matrix multiplication. As earlier, I have defined two global static values namely pi and e with 16 digits of precision. A is defined dynamically as a 2D matrix with each row occupying a contiguous set of memory spaces and the rows being non-contiguous to each other. B and C are defined as 2d matrix of constant length. This ensures that the matrix B is saved in contiguous memory. Another approach for the same is to declare BrowsxBcols length of matrix B dynamically and then parse through it with i*(Bcols) + j for going to the ith row and jth column. The MPI is

initialized and the rank and number of processors are taken in to the variable myid and nprocs respectively. The program is written so as to allow it to be generalized for any value of matrices sizes condition that Brows is equal to Acols as required.

The if condition allows only processor 0 namely as the master processor to enter the first condition. First, A and B are initialized and B matrix is broadcast to all the processors as an array of size Bcols*Brows with double precision values sent from the master node to all the mpi_comm_world. This will send the B matrix to any processor listening for a matrix form the master processor. Further, the master processor sends out the rows of A to all the processor at hand as a 1D array of the next row "rowsent" to processor rowsent+1 till rowsent = nprocs-1. The "for" loop is begins at the master processor to receive the results and process the remaining rows to the processors. Master processor listens for the results from the other processors. The sender and tag values are saved in the to substitute the answer at the correct position in matrix C. It has been assumed that the Arows>nprocs. Hence, the remaining rows are sent to the rest of the processors with an if condition till the end of number of rows. At the end of which a tag "0" is sent out with MPI_Botton to define the end of process.

The other part is for all the processors other than the master processor. The Bcast is used to listen for b matrix from the master processor. Then, again the processor listens for receiving a row of the matrix A. This will be the row with rowsent+1 value as the rank(myid) for this processor. Each processor receives the buff i.e. row of a with their rank +1. The while loop runs until we achieve the tag "0" value at the end of all the rows in A. The row wise multiplication is then performed for the present buff value. And the code proceeds to sending the result back to the master processor. The master processor receives this answer and the status of the row to position the answer at the correct row value. The next row is sent to the processor for solving. It goes on until all the rows are passed at which time the master code passes tag "0" which ends the process for this processor. The result is printed and mpi_finalize cleans up all mpi state.

2. Sample Input/Output:
   A 16 digit precision values for pi and exponential term were provided as static global variable is used to test the accuracy of the code. A is given as a 6x6 and 12x12 matrix with each value given as pi*(i*j) where i and j are the present row and column. The B matrix is composed of exponential powered i+j, where i is the row value and j the column value. This gives a very high precision calculation, as required to test the veracity of the program. Same was performed for a matrix of 12x12.
3. Description of results:
   The results for the high precision values of A and B multiplication was performed using MATLAB. The results are shown. Further, the program was tested with time taken for same calculation using a single processor

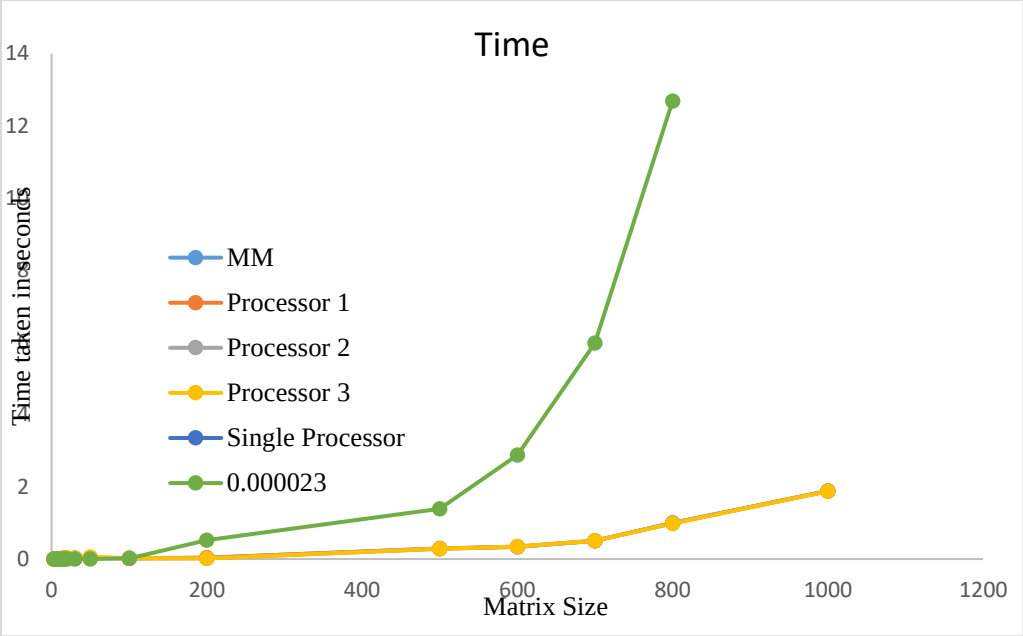

Figure 1. Time taken by the processors in a 4 processor mpi program and that by a single processor program.

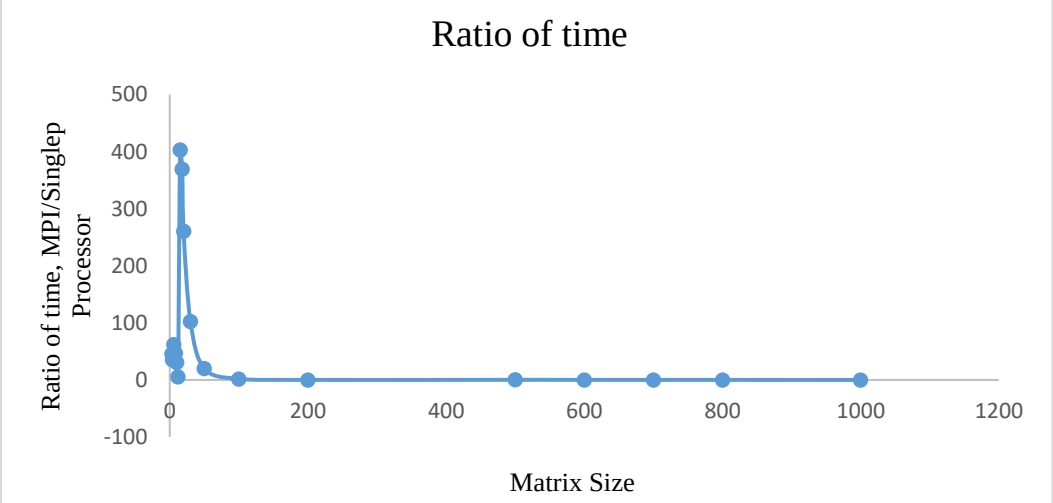

Figure 2. Comparison of time taken by single and multiple processors

4. Main Results: The results are given as below.
   With 6X6 Matrix

A = pi*(i*j) B = e^(i+j)

A

0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	3.141593	6.283185	9.424778	12.566371	15.707963
0.000000	6.283185	12.566371	18.849556	25.132741	31.415927
0.000000	9.424778	18.849556	28.274334	37.699112	47.123890
0.000000	12.566371	25.132741	37.699112	50.265482	62.831853

0.000000 15.707963 31.415927 47.123890 62.831853 78.539816
B
1.000000 2.718218 7.388710 20.084128 54.593042 148.395802
2.718218 7.388710 20.084128 54.593042 148.395802 403.372178
7.388710 20.084128 54.593042 148.395802 403.372178 1096.453614
20.084128 54.593042 148.395802 403.372178 1096.453614 2980.400219
54.593042 148.395802 403.372178 1096.453614 2980.400219 8101.378255
148.395802 403.372178 1096.453614 2980.400219 8101.378255
22021.314189

C

0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
3261.284825 8864.883916 24096.689208 65500.060270 178043.458934
483960.978639
6522.569650 17729.767832 48193.378415 131000.120540 356086.917868
967921.957278
9783.854474 26594.651747 72290.067623 196500.180810 534130.376803
1451882.935917
13045.139299 35459.535663 96386.756831 262000.241081 712173.835737
1935843.914556
16306.424124 44324.419579 120483.446038 327500.301351 890217.294671
2419804.893195

Using Matlab
ans =

1.0e+06 *

Columns 1 through 3

0	0	0
0.003261637308500	0.008866049426719	0.024100421046869
0.006523274616999	0.017732098853438	0.048200842093739
0.009784911925499	0.026598148280156	0.072301263140608
0.013046549233999	0.035464197706875	0.096401684187478
0.016308186542498	0.044330247133594	0.120502105234347

Columns 4 through 6

0	0	0
0.065511736589917	0.178079363123167	0.484069896801265
0.131023473179834	0.356158726246334	0.968139793602530
0.196535209769751	0.534238089369501	1.452209690403795

0.262046946359668 0.712317452492668 1.936279587205060
0.327558682949585 0.890396815615836 2.420349484006325

Comparison: Listed are the difference between the calculated values using matlab and MPI program
For Row 1
0
0
0
0
0
0
For Row 2
1.0e+05 *

0.031128716658974
0.084614551225073
0.230000560495715
0.625191022829583
1.699403750064246
4.619345128441933
For Row 3
1.0e+05 *

0.063741564908974
0.173263390385073
0.470967452565715
1.280191625529583
3.479838339404246
9.458954914831933

For Row 4
1.0e+06 *

0.009635441314897
0.026191222953507
0.071193434461572
0.193519222822958
0.526027292875425
1.429856470122193
For Row 5
1.0e+06 *

0.012896726139897
0.035056106869507
0.095290123672572
0.259019283093958
0.704070751809425
1.913817448761193

For Row 6

1.0e+06 *

0.016158010964897
0.043920990785507
0.119386812879572
0.324519343363958
0.882114210743425
2.397778427400193

5. Analysis:

The method is observed to be more accurate for low values and diverges from the results given by MATLAB for higher values. As evident form the results the difference increases as we travel down the column and the values increase. On going down with more rows, the results diverges from that calculated using matlab. Similar results were observed for the matrix of 12x12 size.

The results for the comparison between the solution using single and multiprocessors are given and the plot shows that initially for Arows less than 200, the time taken by a single processor is lower by a proportion of 5 for Arows <50. However, the time ratio increases with increase in Arows to a value of 200. On further increment of Arows, i.e. for Arows from 20-50, the time difference reduces. The plot is as shown in figure 2. For matrices of size higher than 200x200, the mpi program takes much lower time than with single processor. Hence, using MPI for this data range is suitable.

3. BLAS:

1. Code Description

The basic code is similar to the one used for Matrix-matrix multiplication. The changes made are in solving the two sets of arrays.

The BLAS library is included and the B matrix is initialised as a 1D array of length Bcols*Brows to be used with BLAS. The flow of program is similar to that of Matrix-matrix multiplication up till while loop. Here, the BLAS functions are used to convert the 1D matrix into 2D matrix using gsl_matrix_view. These, temporary matrices are tehn multiplied using gsl_blas_dgemm function and the results are saved in ans 1D matrix. This is then sent back to the master processor as the result. The variation is done to use the BLAS library to perform matrix-vector multiplication.

2. Sample Input/Output:

The same input file as used in the matrix-matrix multiplication is used to see the accuracy. Then the number of rows and columns are increased to see the effect of using BLAS for matrix multiplication.

3. Description of results:

Again comparison for the same results from MATLAB and that using the code from the last problem was performe. The results are shown for the 6x6 matrix while those for 12x12 matrix shoed similar trends.

Further, the program was tested with time taken for same calculation to see the effect of using BLAS library for multiplication. The effect would be more evident in with higher number of processors. Hence, it was increased to see the results for a matrix size of 500x500.

4. Main Results: The results are given as below.

With 6X6 Matrix

A =
pi*(i*j) B = e^(i+j)
A
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 3.141593 6.283185 9.424778 12.566371 15.707963
0.000000 6.283185 12.566371 18.849556 25.132741 31.415927
0.000000 9.424778 18.849556 28.274334 37.699112 47.123890
0.000000 12.566371 25.132741 37.699112 50.265482 62.831853
0.000000 15.707963 31.415927 47.123890 62.831853 78.539816
B
1.000000 2.718218 7.388710 20.084128 54.593042 148.395802
2.718218 7.388710 20.084128 54.593042 148.395802 403.372178
7.388710 20.084128 54.593042 148.395802 403.372178 1096.453614
20.084128 54.593042 148.395802 403.372178 1096.453614 2980.400219
54.593042 148.395802 403.372178 1096.453614 2980.400219 8101.378255
148.395802 403.372178 1096.453614 2980.400219 8101.378255 22021.314189

Rowsent1 1

myid C
0

0.000000 0.000000 0.000000 0.000000 0.000000
0.000000
3261.284825 8864.883916 24096.689208 65500.060270 178043.458934
483960.978639
6522.569650 17729.767832 48193.378415 131000.120540 356086.917868
967921.957278
9783.854474 26594.651747 72290.067623 196500.180810 534130.376803
1451882.935917
13045.139299 35459.535663 96386.756831 262000.241081 712173.835737
1935843.914556
16306.424124 44324.419579 120483.446038 327500.301351 890217.294671
2419804.893195

We achieve these same results without using BLAS library. Hence, the time taken was studied. The Figure 3, shows the comparison for the time taken to solve the same using BLAS and without using BLAS.

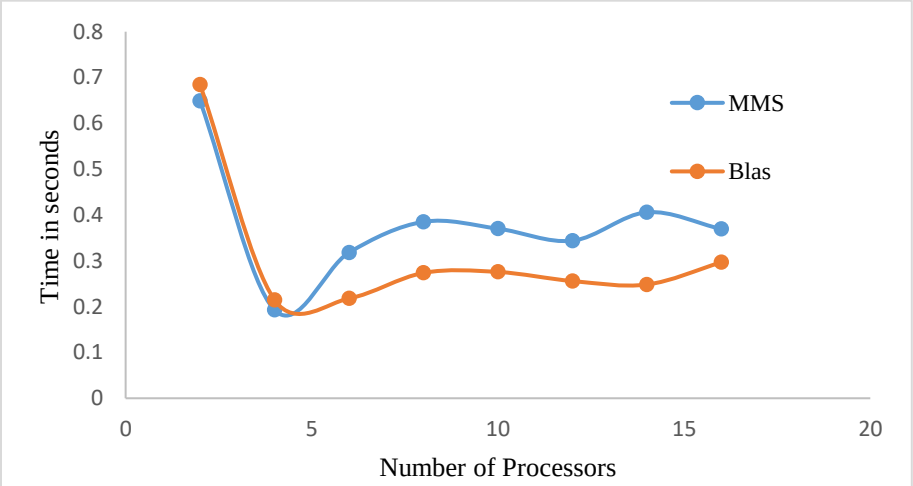

Figure 3. Comparison of Matrix multiplication with using BLAS.

5. Analysis

The accuracy for calculation is same as that for matrix matrix multiplication, hence, it is diverging from that given usng matlab by the same amount.

However, with increasing number of processors, the use of BLAS library was found to be working better with a shorter time for calculations.

4. Timing:

1. Code Description:
   In the code for the matrix multiplication, and that using BLAS and that with just one processor, the MPI_Wtime() function was used to measure the time taken in performing the calculations. The begin time was used after the initialising and the end time as used before finalize. However, no results were printed and no extra loops were brought into time calculation to give the absolute time.
2. Sample Input/Output:
   The size of matrix was varied over a range of 3-1000 to see the variation of time taken to calculate. This was done for the case of 4 processors for MPI based programs and 1 processor for the single processor.
   Based on the results observed in the above plot, matrix size 500x500 was used to perform the comparison between the code using BLAS library with that not using BLAS library.
3. Description of Results:
   The plot of time with matrix size N in figure 1, shows the difference between the single processor and multiprocessor. The figure 3 shows the variation of time when using 500x500 matrix size for program using BLAS and without it. The comparison of using BLAS and without BLAS library for varying matrix size is shown in Figure 4.
4. Main Results:
   The figure 4 shows the variation between the time taken with the size of the matrix. The figure 3 shows the variation between the program using BLAS library and that not using it as a function of the number of processors used.

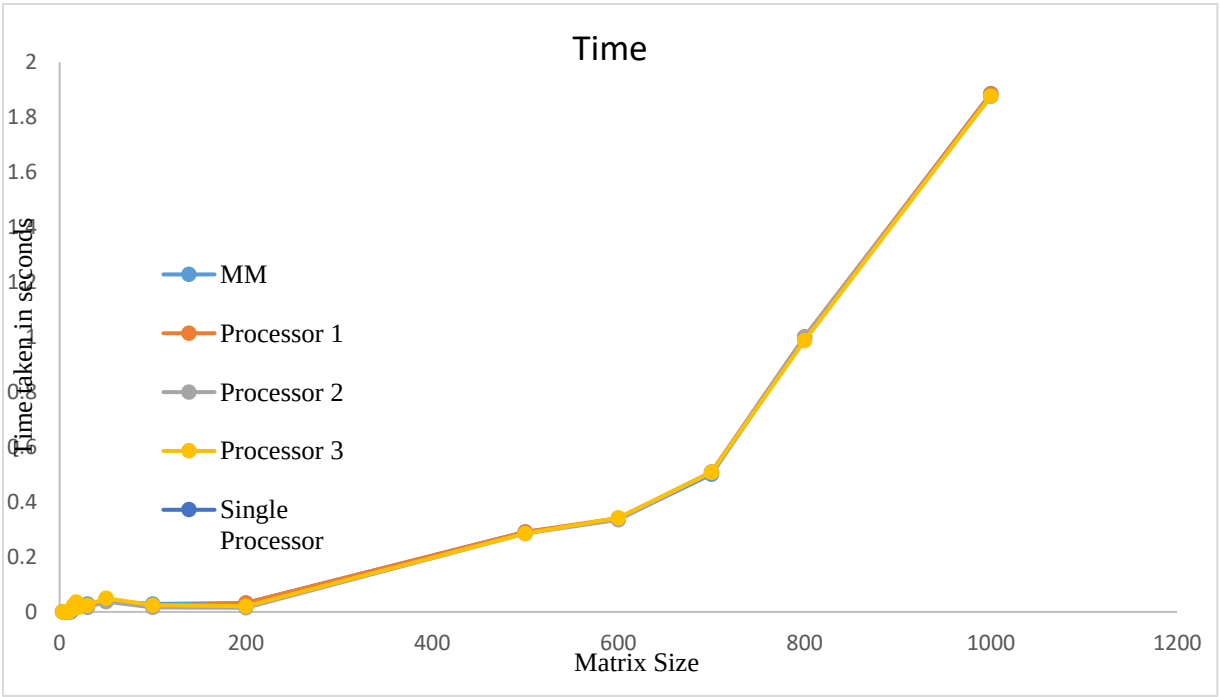

Figure 4. Time taken the matrix size

N	MM	BLAS	One Processor	Time Ratio

3	0.001045	0.021044	0.000023	45.4347826
4	0.00092	0.001497	0.000023	40
6	0.001485	0.001601	0.000026	57.1153846
8	0.001462	0.001075	0.000024	60.9166667
10	0.001172	0.001045	0.000031	37.8064516
12	0.00025	0.00112	0.000038	6.57894737
15	0.01974	0.000283	0.000044	448.636364
18	0.021042	0.023167	0.000049	429.428571
20	0.026819	0.005597	0.000057	470.508772
30	0.028946	0.007261	0.000103	281.029126
50	0.039196	0.023627	0.000283	138.501767
100	0.028946	0.004645	0.001929	15.0057024
200	0.032732	0.011967	0.016197	2.02086806
500	0.290697	0.200191	0.52434	0.55440554
600	0.336528	0.342144	1.389107	0.24226212
700	0.500936	0.590304	2.884109	0.1736883
800	0.9964	0.84796	5.987482	0.16641386
1000	1.886824	1.839969	12.687548	0.14871463

Table 1. Time taken with matrix size variation.

5. Analysis: The Figure 1 clearly shows a variation in speed of calculation performed using MPI for matrix size above 500. Hence 500x500 matrix size was chosen for further analysis. It can be seen that for N<200, the time required for message passing plays a major role in determining the completion. Hence, with increase in N, it becomes less and less critical in determining the total time.
   Table 1 gives the time variation for all the three types. The comparison of time taken with BLAS library included shows very small variation in the range of 0.001 with and without using BLAS library as shown in Figure 4. On observing the individual plots, we see that the slope of Time vs N increase with N at various instances. A polynomial regression of this plot gives a parabolic plot for the time taken with matrix size. This is reasonable with increase in the size of calculation by square terms every time N is increased.

5. Improvements:

Two improvements can be proposed here for decreasing the time taken.

1. Use of MPI_ISend instead of MPI_Send:
   This is useful, as it will leave the processor 1 for further calculations or sending data to the next processor rather than waiting. Also, we can use this processor for calculations and further enhance our speed. This can be done in this case, since we are not varying the matrix A or any row of matrix A, hence no changes will be brought in the values of A. Therefore, use of MPI_Isend will be an enhancement in the code.
2. Sending multiple rows of array A:

This can be done by dividing the number of rows by the number of processors nprocs. If N i.e. the number of rows is a multiple of nprocs, then the rows are equivalently divided in the first attempt, otherwise we can attach the remaining rows with the last processor. This will decrease the number of times, message passing is performed. However, increase in length of the data passed will affect the time. Hence, this is viable choice for only a range of N.

Appendix

1. Matrix-vector Multiplication

```
2. //Identity Matrix vector Multiplication
3. #include <stdio.h>
4. #include "mpi.h"
5. #include <math.h>
6. #define e 2.7182182459046
7. #define pi 3.141592653589793
8. int main(int argc, char** argv)
9. {
10.       int nprocs, myid, nrows,crow, ncols , rowsent,i, j, anstype,
   sender, k, N=6, master = 0 ;
11.       double **A, *b, *c;
12.       double ans=0.0, *buff;
13.       double begintime1, begintime2, endtime1, endtime2;
14.       //printf("Give the size fo Identitiy Matrix\n");
15.       MPI_Status status;
16.       //MPI_Datatype anstype;
17.       // initialize MPI, determine/distribute size of arrays here
18.       // assume A will have rows 0,nrows-1 and columns 0,ncols-1, b
   is 0,ncols-1
19.       // so c must be 0,nrows-1
20.
21.       nrows = N;//for NxN matrix
22.       ncols = N;//for NxN matrix
23.       MPI_Init(&argc, &argv); //Initialize the MPI environment
24.       MPI_Comm_size(MPI_COMM_WORLD, &nprocs);//get the number of
   processes
25.       MPI_Comm_rank(MPI_COMM_WORLD, &myid); //get the rank of the
   processes
26.       begintime1 = MPI_Wtime();
27.       b = (double *) calloc(ncols, sizeof(double));
28.
29.       c = (double *) calloc(ncols, sizeof(double));
30.       buff = (double *) calloc(ncols, sizeof(double));
31.       // Master part
32.       if (myid == master ) {
33.       // Initialize or read in matrix A and vector b here
34.       A = (double **) calloc(nrows, sizeof(double *));
35.       for(k =0;k<nrows;k++){
```

```
36.       A[k] = (double *) calloc(ncols, sizeof(double));
37.       }
38.
39.
40.       printf("A\n");
41.       for (i=0;i<nrows;i++){
42.         for(j=0;j<ncols;j++){
43.           if(i==j) {
44.             A[i][j] = pi*(i+j);
45.           }
46.           printf("%f\t", A[i][j]);//initialising matrix A
47.         }
48.         printf("\n");
49.       }
50.
51.       for(k =0;k<ncols;k++){
52.       b[k] = (double)pow(e,k); //initialising matrix B
53.
54.       }
55.
56.       // send b to every slave process, note b is an array and
   b=&b[0]
57.       MPI_Bcast(b, ncols, MPI_DOUBLE_PRECISION, master,
   MPI_COMM_WORLD);
58.       // send one row to each slave tagged with row number, assume
   nprocs<nrows
59.       rowsent=0;
60.       for (i=1; i< nprocs; i++) {
61.        // Note A is a 2D array so A[rowsent]=&A[rowsent][0]
62.        MPI_Send(A[rowsent], ncols,
   MPI_DOUBLE_PRECISION,i,rowsent+1,MPI_COMM_WORLD);
63.        rowsent++;
64.       }
65.         for (i=0; i<nrows; i++) {
66.        MPI_Recv(&ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
   MPI_ANY_TAG,
67.            MPI_COMM_WORLD, &status);
68.        sender = status.MPI_SOURCE;
69.        anstype = status.MPI_TAG;           //row number+1
70.        c[anstype-1] = ans;
71.        if (rowsent < nrows) {               // send new row
72.
   MPI_Send(A[rowsent],ncols,MPI_DOUBLE_PRECISION,sender,rowsent+1,MPI_
   COMM_WORLD);
73.          rowsent++;
74.        }
75.        else        // tell sender no more work to do via a 0 TAG
76.
   MPI_Send(MPI_BOTTOM,0,MPI_DOUBLE_PRECISION,sender,0,MPI_COMM_WORLD);
77.       }
```

```
78.       }
79.       // Slave part
80.       else {
81.       // slaves recieve b, then compute dot products until done
   message recieved
82.       MPI_Bcast(b, ncols, MPI_DOUBLE_PRECISION, master,
   MPI_COMM_WORLD);
83.       MPI_Recv(buff,ncols,MPI_DOUBLE_PRECISION,master,MPI_ANY_TAG,M
   PI_COMM_WORLD,&status);
84.
85.       while(status.MPI_TAG != 0) {
86.        crow = status.MPI_TAG;
87.        ans=0.0;
88.        for (i=0; i<ncols; i++)
89.           ans+=buff[i]*b[i];
90.        MPI_Send(&ans,1,MPI_DOUBLE_PRECISION, master, crow,
   MPI_COMM_WORLD);
91.        MPI_Recv(buff,ncols,MPI_DOUBLE_PRECISION,master,
   MPI_ANY_TAG,
92.               MPI_COMM_WORLD, &status);
93.       }
94.       }
95.       // output c here on master node
96.       if((myid==master)&&(rowsent>=nrows)){
97.       for(k=0;k<nrows;k++){
98.       printf("b %f\t", b[k]);
99.       printf("C %f\n", c[k]);
100.      }
101.      }
102.      endtime1 = MPI_Wtime();
103.      printf("That took %f seconds for processor %d\n",endtime1-
   begintime1, myid);
104.      MPI_Finalize();
105.      //free any allocated space here
106.      return 0;
107.      }
108.
```

2. **Matrix-Matrix Multiplication**

```
3. //Inverse Matrix Multiplication
4. #include <stdio.h>
5. #include "mpi.h"
6. #include <math.h>
7. #define e 2.7182182459046
8. #define pi 3.141592653589793
9. int main(int argc, char** argv)
10.       {
```

```
11.          int nprocs, anstype, myid, m, rowsent,  sender, Arows=500,
   Acols=500, Brows, Bcols=500, Crows, Ccols, crow , i, j, k, N, master
   = 0 ;
12.               double **A, B[500][500], C[500][500];
13.               double ans[500], buff[500];
14.               MPI_Status status;
15.               double begintime1, begintime2, endtime1, endtime2;
16.               // initialize MPI, determine/distribute size of
   arrays here
17.               // assume A will have rows 0,nrows-1 and columns
   0,ncols-1, b is 0,ncols-1
18.               // so c must be 0,nrows-1
19.          Brows = Acols ;
20.          Crows = Arows;
21.          Ccols = Bcols;
22.          MPI_Init(&argc, &argv); //Initialize the MPI environment
23.          MPI_Comm_size(MPI_COMM_WORLD, &nprocs);//get the number of
   processes
24.          MPI_Comm_rank(MPI_COMM_WORLD, &myid); //get the rank of
   the processes
25.               begintime1 = MPI_Wtime();
26.        if (myid == master ) {
27.          // Initialize or read in matrix A and vector b here
28.                A = (double **) calloc(Arows, sizeof(double *) );
29.                for(k =0;k<Arows;k++){
30.                       A[k] = (double *) calloc(Acols,
   sizeof(double) );
31.                }
32.               printf("A\n");
33.                       for (i=0;i<Arows;i++){
34.                              for(j=0;j<Acols;j++){
35.                                     A[i][j] = pi*(i+j);
36.                                     printf("%f\t", A[i][j]);
37.                              }
38.                              printf("\n");
39.                       }
40.               printf("B\n");
41.                for (i=0;i<Bcols;i++){
42.                       for(j=0;j<Bcols;j++){
43.                              B[i][j] = pow(e, i+j);
44.                              printf("%f\t", B[i][j]);
45.                              }
46.                              printf("\n");
47.               }
48.
49.               // send b to every slave process, note b is an array
   and b=&b[0]
50.               MPI_Bcast(B,Bcols*Brows, MPI_DOUBLE_PRECISION,
   master, MPI_COMM_WORLD);
51.
```

```
52.          // send one row to each slave tagged with row number,
   assume nprocs<nrows
53.          rowsent=0;
54.          for (i=1; i<nprocs; i++) {
55.            // Note A is a 2D array so A[rowsent]=&A[rowsent][0]
56.            MPI_Send(A[rowsent], Acols,
   MPI_DOUBLE_PRECISION,i,rowsent+1,MPI_COMM_WORLD);
57.            rowsent++;
58.          }
59.             for (m=0; m<Arows; m++) {
60.           MPI_Recv(ans, Crows, MPI_DOUBLE_PRECISION,
   MPI_ANY_SOURCE, MPI_ANY_TAG,
61.                       MPI_COMM_WORLD, &status);
62.              sender = status.MPI_SOURCE;
63.            anstype = status.MPI_TAG;            //row number+1
64.                for(i=0;i<Arows;i++){
65.            C[anstype-1][i] = ans[i];
66.                }
67.            if (rowsent < Arows) {                // send new row
68.
   MPI_Send(A[rowsent],Acols,MPI_DOUBLE_PRECISION,sender,rowsent+1,MPI_
   COMM_WORLD);
69.              rowsent++;
70.               }
71.            else        // tell sender no more work to do via a 0
   TAG
72.
   MPI_Send(MPI_BOTTOM,0,MPI_DOUBLE_PRECISION,sender,0,MPI_COMM_WORLD);
73.          }
74.        }
75.
76.        // Slave part
77.        else {
78.          // slaves recieve b, then compute dot products until done
   message recieved
79.               MPI_Bcast(B, Bcols*Brows, MPI_DOUBLE_PRECISION,
   master, MPI_COMM_WORLD);
80.
81.                      MPI_Recv(buff, Acols, MPI_DOUBLE_PRECISION,
   master, MPI_ANY_TAG,
82.                      MPI_COMM_WORLD, &status);
83.
84.                          while(status.MPI_TAG != 0){
85.                                  crow = status.MPI_TAG;
86.                                  for(i=0;i<Bcols;i++){
87.                                         ans[i] = 0.0;
88.                                         for(j=0;j<Acols;j++){
89.                                                 ans[i] +=
   buff[j]*B[j][i];
90.                                          }
```

```
                                }
                        MPI_Send(ans, Bcols, MPI_DOUBLE_PRECISION,
   master, crow, MPI_COMM_WORLD);
                MPI_Recv(buff, Acols, MPI_DOUBLE_PRECISION, master,
   MPI_ANY_TAG,
                                    MPI_COMM_WORLD, &status);
                }
         }
       // output c here on master node
         if((myid==master)&&(rowsent>=Arows)){
       printf("myid C %d\n\n", myid);
       for(i=0;i<Crows;i++){
                for(j=0;j<Ccols;j++){
                        printf("%f\t", C[i][j]);
                        }
                printf("\n");
                }
       }
       endtime1 = MPI_Wtime();
       printf("That took %f seconds for processor %d for N
   %d\n",endtime1-begintime1, myid, Arows);

        MPI_Finalize();
        //free any allocated space here
         return 0;
       }
```

3. Matrix Matrix Multiplication using BLAS

```
//Inverse Matrix Multiplication
#include <stdio.h>
#include "mpi.h"
#include <math.h>
#include <gsl/gsl_blas.h>
#define e 2.7182182459046
       #define pi 3.141592653589793
       int main(int argc, char** argv)
       {
          int nprocs, anstype, myid, m, rowsent,  sender, Arows=500,
   Acols, Brows, Bcols, Crows, Ccols, crow , i, j, k, N, master = 0 ;
                double **A, B[(Arows*Arows)], C[Arows][Arows];
                double ans[Arows], buff[Arows];
                double begintime1, begintime2, endtime1, endtime2;

                MPI_Status status;
                // initialize MPI, determine/distribute size of
   arrays here
```

```
20.              // assume A will have rows 0,nrows-1 and columns
   0,ncols-1, b is 0,ncols-1
21.              // so c must be 0,nrows-1
22.       Acols = Arows;
23.       Bcols = Arows;
24.          Brows = Acols ;
25.          Crows = Arows;
26.          Ccols = Bcols;
27.          MPI_Init(&argc, &argv); //Initialize the MPI environment
28.          MPI_Comm_size(MPI_COMM_WORLD, &nprocs);//get the number of
   processes
29.          MPI_Comm_rank(MPI_COMM_WORLD, &myid); //get the rank of
   the processes
30.       begintime1 = MPI_Wtime();
31.        if (myid == master ) {
32.          // Initialize or read in matrix A and vector b here
33.                A = (double **) calloc(Arows, sizeof(double *) );
34.                for(k =0;k<Arows;k++){
35.                       A[k] = (double *) calloc(Acols,
   sizeof(double) );
36.                }
37.               //printf("A\n");
38.                       for (i=0;i<Arows;i++){
39.                              for(j=0;j<Acols;j++){
40.                              A[i][j] = pi*(i+j);
41.                                      //printf("%f\t", A[i][j]);
42.                              }
43.                              //printf("\n");
44.                       }
45.               //printf("B\n");
46.                for (i=0;i<Bcols;i++){
47.                       for(j=0;j<Bcols;j++){
48.                       B[i*Arows+j] =pow(e, (i+j));
49.       //printf("%f\t", B[i*Arows+j]);
50.                              }
51.                       //printf("\n");
52.               }
53.
54.               // send b to every slave process, note b is an array
   and b=&b[0]
55.               MPI_Bcast(B,Bcols*Brows, MPI_DOUBLE_PRECISION,
   master, MPI_COMM_WORLD);
56.
57.          // send one row to each slave tagged with row number,
   assume nprocs<nrows
58.          rowsent=0;
59.          for (i=1; i<nprocs; i++) {
60.            // Note A is a 2D array so A[rowsent]=&A[rowsent][0]
61.            MPI_Send(A[rowsent], Acols,
   MPI_DOUBLE_PRECISION,i,rowsent+1,MPI_COMM_WORLD);
```

```
62.             rowsent++;
63.           }
64.          // printf("\n\nRowsent1 %d\n\n", rowsent);
65.           for (m=0; m<Arows; m++) {
66.            MPI_Recv(ans, Crows, MPI_DOUBLE_PRECISION,
   MPI_ANY_SOURCE, MPI_ANY_TAG,
67.                       MPI_COMM_WORLD, &status);
68.                         //for(k=0;k<Arows;k++){printf("ans5recvd
   %f\t", ans[k]);}
69.                         //printf("\n\n");
70.             sender = status.MPI_SOURCE;
71.             anstype = status.MPI_TAG;            //row number+1
72.                 for(i=0;i<Arows;i++){
73.             C[anstype-1][i] = ans[i];
74.                 }
75.
76.             if (rowsent < Arows) {                // send new row
77.
   MPI_Send(A[rowsent],Acols,MPI_DOUBLE_PRECISION,sender,rowsent+1,MPI_
   COMM_WORLD);
78.                rowsent++;
79.            }
80.             else        // tell sender no more work to do via a 0
   TAG
81.
   MPI_Send(MPI_BOTTOM,0,MPI_DOUBLE_PRECISION,sender,0,MPI_COMM_WORLD);
82.           }
83.         }
84.
85.         // Slave part
86.         else {
87.           // slaves recieve b, then compute dot products until done
   message recieved
88.                MPI_Bcast(B, Bcols*Brows, MPI_DOUBLE_PRECISION,
   master, MPI_COMM_WORLD);
89.
90.        //for(i=0;i<Brows;i++){
91.                  //  for(j=0;j<Bcols;j++){
92.                        //   printf("%f\t", B[Arows*i+j]);
93.                        //  }
94.                        //   printf("\n");
95.                        //}
96.            MPI_Recv(buff, Acols, MPI_DOUBLE_PRECISION, master,
   MPI_ANY_TAG,
97.                       MPI_COMM_WORLD, &status);
98.               //for(i=0;i<Acols;i++){
99.               //printf("buff %d here %f\n", myid, buff[i]);
100.              //      }
101.              //printf("myid %d\n\n", myid);
102.
```

```
                        while(status.MPI_TAG != 0){
                                crow = status.MPI_TAG;
                             gsl_matrix_view K =
   gsl_matrix_view_array(buff, 1, Arows);
                             gsl_matrix_view L =
   gsl_matrix_view_array(B, Arows, Acols);
                             gsl_matrix_view M =
   gsl_matrix_view_array(ans, 1, Acols);
                             gsl_blas_dgemm (CblasNoTrans,
   CblasNoTrans,
                             1.0, &K.matrix, &L.matrix,
                             0.0, &M.matrix);


         //for(k=0;k<Arows;k++){printf("ans5recvd %f\t", ans[k]);}
                                       //printf("\n\n");
              MPI_Send(ans, Bcols, MPI_DOUBLE_PRECISION, master,
   crow, MPI_COMM_WORLD);
              MPI_Recv(buff, Acols, MPI_DOUBLE_PRECISION, master,
   MPI_ANY_TAG,
                               MPI_COMM_WORLD, &status);
              }
       }
      // output c here on master node
      // if((myid==master)&&(rowsent>=Arows)){
      //printf("myid C %d\n\n", myid);
      //for(i=0;i<Crows;i++){
      //      for(j=0;j<Ccols;j++){
              //      printf("%f\t", C[i][j]);
              //      }
              //printf("\n");
              //}
      //}
      endtime1 = MPI_Wtime();
      printf("That took %f seconds for processor %d for N
   %d\n",endtime1-begintime1, myid, Arows);

       MPI_Finalize();
       //free any allocated space here
        return 0;
      }
```