

Steve's Matrix Mutiply Verification Challenge Analysis

Simone Atzeni
simone@cs.utah.edu

June 24, 2009

Abstract

In this report, there is an analysis of the *Matrix Multiplication: Correctness* by Stephen F. Siegel [3] presented in the *Challenge Problem Session (EC)² 2009 (Exploiting Concurrency Efficiently and Correctly)* [1] workshop. The problem consists in a parallel C/MPI-based implementation of matrix-matrix multiplication based on an example from [2]. The target of this challenge is to prove that the program produces the correct output for any given input, and showing that it terminates, does not deadlock, uses MPI correctly, etc.. In order to verify the correctness of the program, the parallel version is compared with the sequential version and will be showed the they are *functionally equivalent*. The correctness of the program will be verified with the assertion method and using the *ISP (In-Situ Model Checker)* Tool. Besides, in this report will be proposed various parallel versions of the program using different MPI functions. Each version, will be analyzed under a point of view of performance (execution times, number of processors, etc.) and with the help of ISP Tool will be checked the presence of deadlock, memory leak, assert violations, etc..

1 Analysis

Analyzing the program *Steves Matrix Mutiply Verication Challenge*, I tryed to create other versions of it (see Appendix A for the code of each program). So, I modied it using different MPI functions to see the behaviors of the programs. We have 7 different version of *Steves Matrix Mutiply Verication Challenge*, which are:

Original it uses *MPI_Send* and *MPI_Recv*, the master process give a matrixs row to avery process and they return the results

- *iSend* another version like Original but using Non-Blocking Send and Receive, so *MPI_Isend* and *MPI_Wait*

Deterministic it uses *MPI_Send* and *MPI_Recv* but master knows which process will return the results anytime

- *iSendD* another version like Determinstic but using *MPI_Isend* and *MPI_Wait*

More Rows per Process it changes the distribution of the rows among the processes, master send at each process the max numbers of rows (number of rows / number of processes)

Scatter it uses *MPI_Scatter* ad *MPI_Gather* functions, so the arrays is divided automatically among the processes

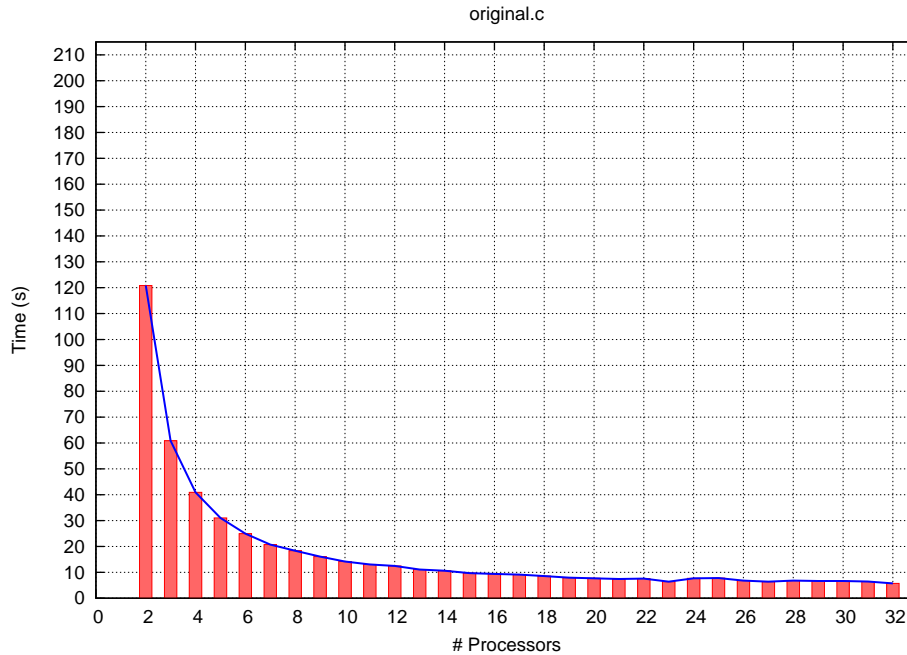
One-Sided Communication it uses RMA (Remote Memory Access), so each process can access at masters memory, takes the rows using *MPI_Get*, computes the result and return it to the master using *MPI_Put*

I ran these versions with big matrixs, matrix A and matrix B respectively 1024x2048 and 2048x1024, so the consequent matrix C is 1024x1024. To run the programs I used the Raven Cluster, thru Raven Server. For each execution, every program saved the number of processors used and the execution time with that number of processors. Every program was run using a number of processors between 2 and 32, but for each number of processor, one is the master, the other attend to calculate the result.

2 Program Versions

2.1 Original Version

The executions of the *Original Version* have showed the follow results:



In the original version, master process sends a row to each process, they compute the results and return it to master, that stores it and send a new row at each process until the rows are finished. So master sends a small message toward the other processors and it seems be quite fast. How we can see in the graphic, with the increasing of the number of processors, the execution time decreasing. Maybe its a obvious situation, because the matrix calculation is parallelized since attending more processors. The graphics look like a *Inverse Exponential Function*, so we can say that with a big number of processors the execution time converges to 0. Anyway, when the number of processors exceed 23 processors, the execution time is approximately constant around 6 seconds. So increasing processors exceeded 23 does not improve widely the execution times, maybe because increasing the number of processors, growing up the number of messages that produce more overhead, slowing down the execution times.

2.2 Original vs. Sequential Version

As pointed out in [1] by Siegel [3] the goal of the challenge is to prove that the parallel program products the correct output, so that is functionally equivalent to the sequential version. In this section will be analyzed the parallel version embedded in the sequential version. At the end of the computation of both programs, the process 0 call the assert function equaling each cell of the Sequential Version's matrix result with the correspondent cell of the Parallel Version's matrix result. Running this program with ISP Tool, it can check every assert violation, so if the result of the Sequential Program is different from the result of the Parallel Program, ISP stops the execution and report the error.

Below is showed the output of ISP after the execution of the the program that embeds the two version, sequential and parallel.. ISP does not report errors, so the program is free from assert violation (this means that the result of the two program is the same), deadlock, memory leak, etc.. In the see Appendix A is showed the code of the program. For the example have been a matrix A (size $1024 * 2048$), a matrix B (size $2048 * 1024$) and a matrix C to keep the result (size $1024 * 1024$), the number of processes is only 2, because just with 3 processes ISP would spawn an huge number of interleaving since the matrix's size is very big.

Output of ISP

ISP - Insitu Partial Order

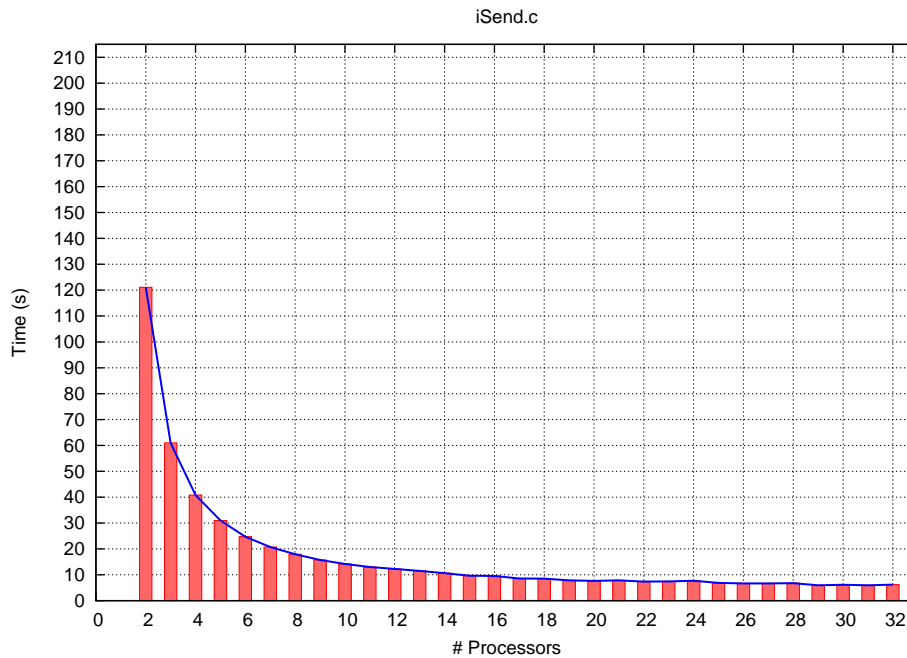
Command: ./a.out
Number Procs: 2
Server: localhost:9999
Blocking Sends: Disabled

Started Process: 14056
INTERLEAVING :1

ISP detected no deadlocks!
Total Interleavings: 1

2.3 Original Non-Blocking Version

The executions of the *Original Non-Blocking Version* have showed the follow results:



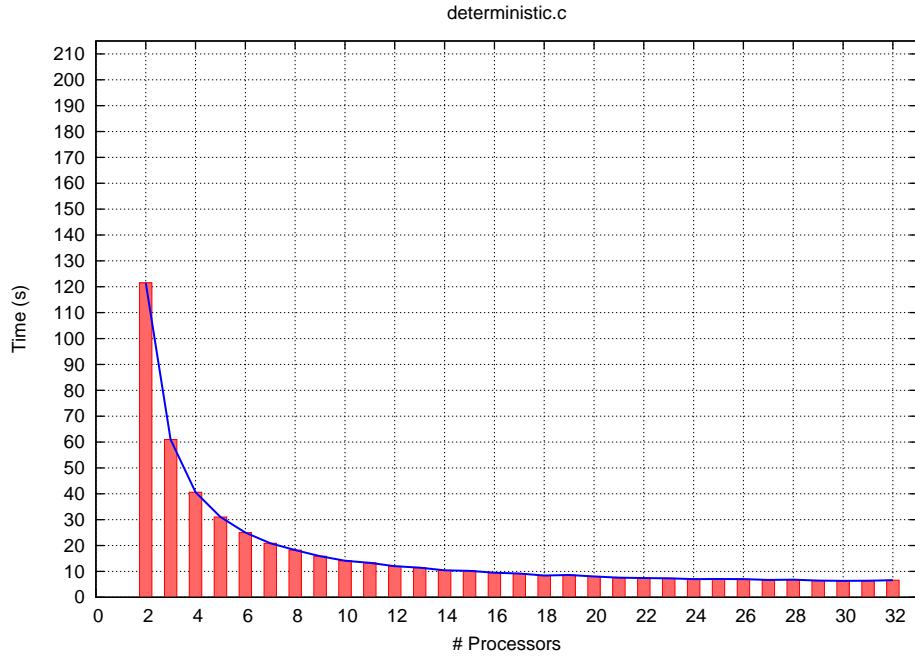
This version results the faster, but we can see the same behavior of the others version, when the number of processors exceed 23-24.

Running ISP, with the *Original and Original Non-Blocking Version*, it spawns a huge number of interleaving (> 50000) and it does not find deadlocks.

2.4 Deterministic Version

The *Deterministic Version* is different from previous version because have been removed the wildcards *MPI_ANY_SOURCE* and have been inserted the exact source from which master will receive the message.

The executions of the *Deterministic Version* have showed the follow results:

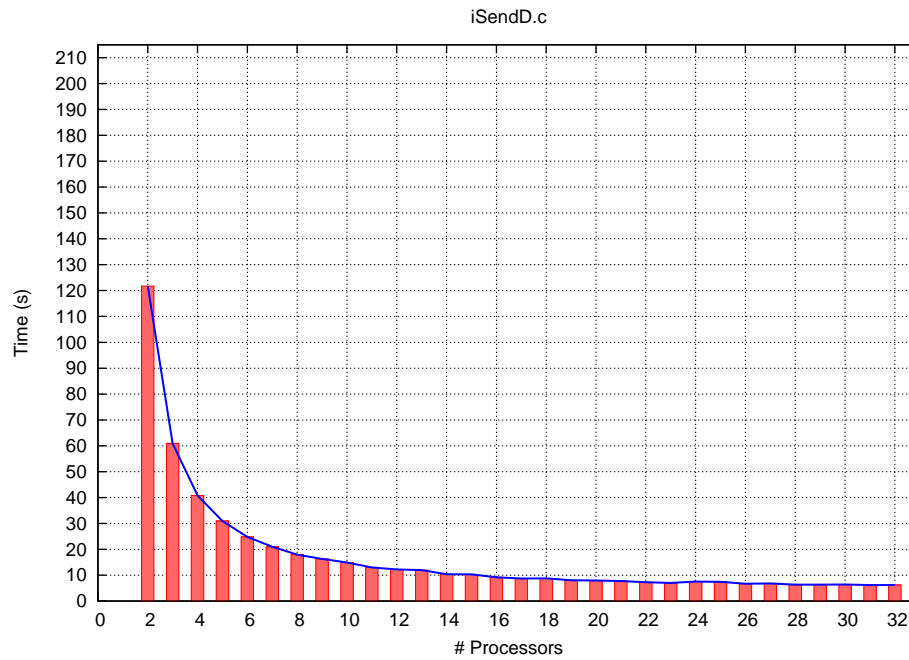


Running ISP with Deterministic Version the number of interleavings is the same whatever is the number of processors because in the code there isn't the wildcard *MPLANY_SOURCE* but master knows who will send the message, furthermore ISP does not point out any deadlock. Below, is shown a screenshot of ISP run with 5 processors.

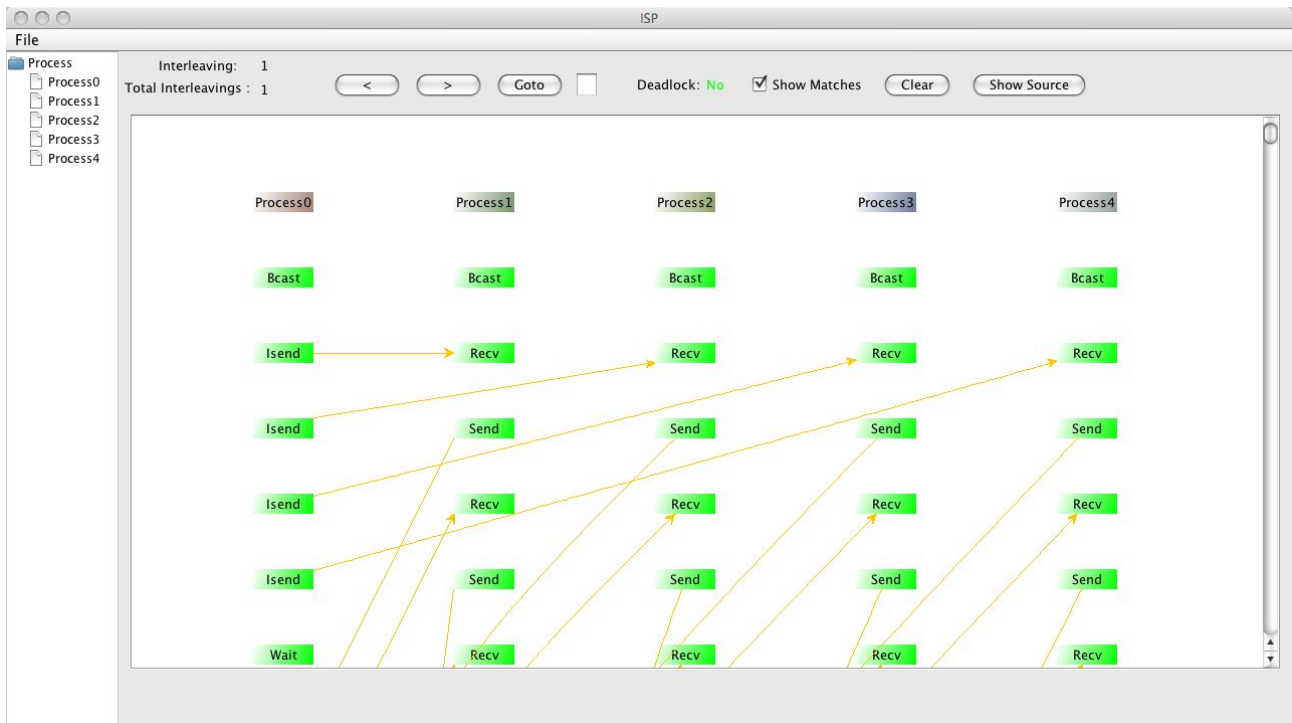


2.5 Deterministic Non-Blocking Version

The executions of the *Deterministic Non-Blocking Version* have showed the follow results:

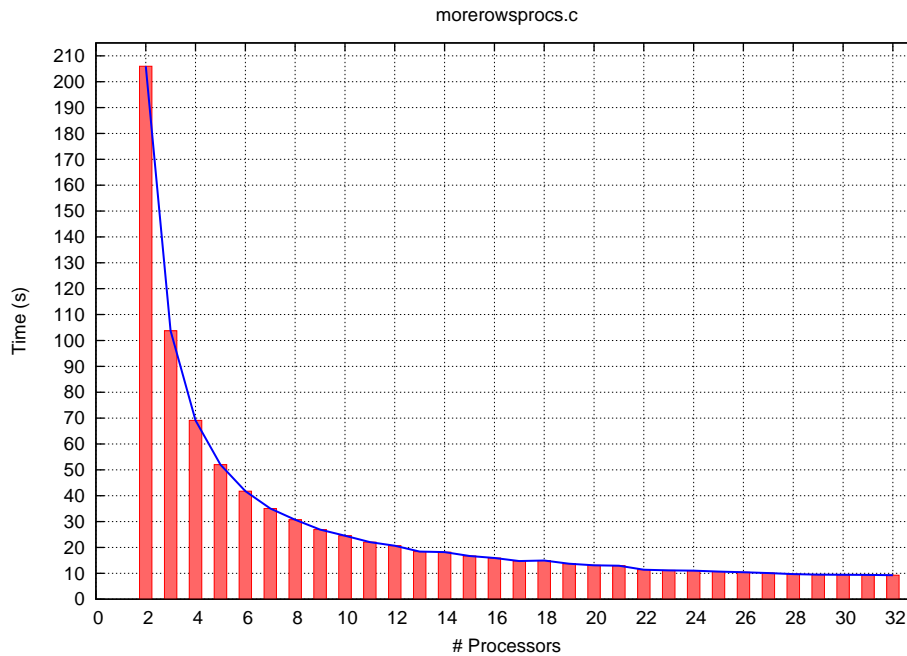


This version has the same behavior of the *Deterministic Version*, ISP does not detect any deadlock, and like the previous version, there are not wildcards, so it runs with only one interleaving.



2.6 More Rows per Process Version

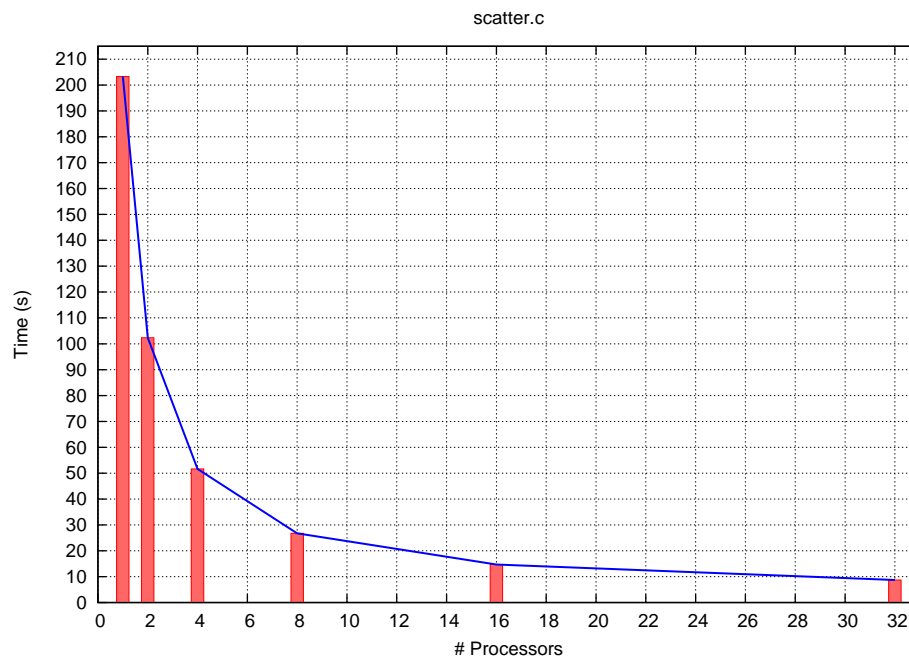
In this version, master processors divide all matrix rows among the processors. So it creates a big message with information about a given number of rows. For example if the matrix have 10 rows and the program is run with 3 processors, master process gives 5 rows to a process and the other 5 rows to other process. They compute the results and give back it to master process that stores it. Increasing the matrix size grow up the size of the message, so may be this is a reason for which this version is more slow than previous versions, because sending a big message take much time. The executions of the *More Rows per Processor Version* have showed the follow results:



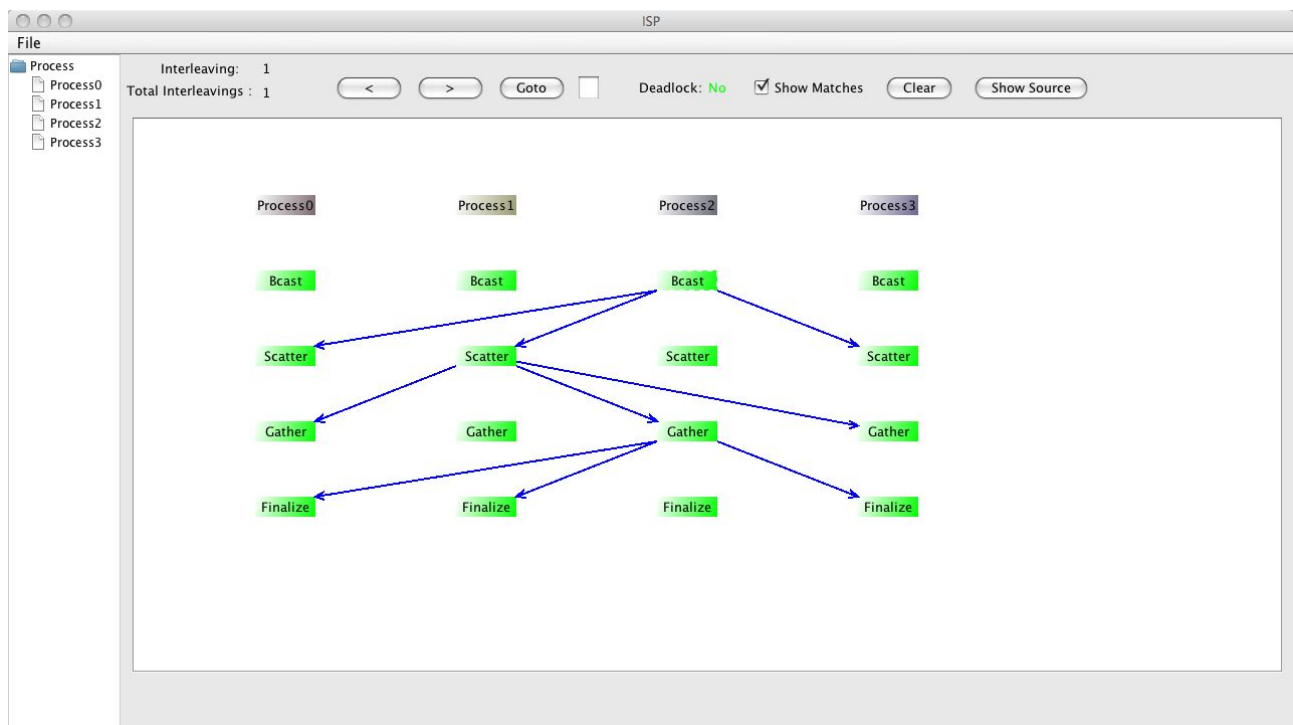
How we can see in the graphics, the execution time is double respect to original version. But increasing the number of processors the times difference is very close. Running ISP with this version it doesnt point out deadlocks but the number of interleavings is very high.

2.7 Scatter and Gather Version

The executions of the *Scatter-Gather Version* have showed the follow results:

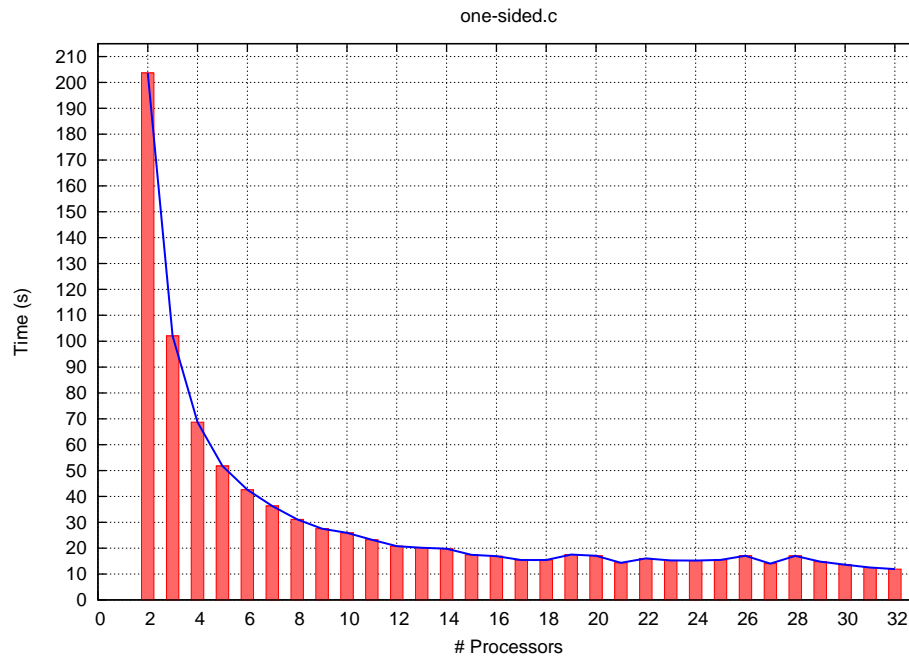


This version uses the *MPI_Scatter* and *MPI_Gather* functions, so the arrays that represents matrix A is divided perfectly and automatically among all processors. Also in this version, if we have a big matrix, master sends toward every processors a huge quantity of information so the size message is big and it can slow down the computation. ISP in this case spawns only one interleaving and it notifies no deadlock. The image below shows a screenshot of ISP program:



2.8 One-Sided Communications Version

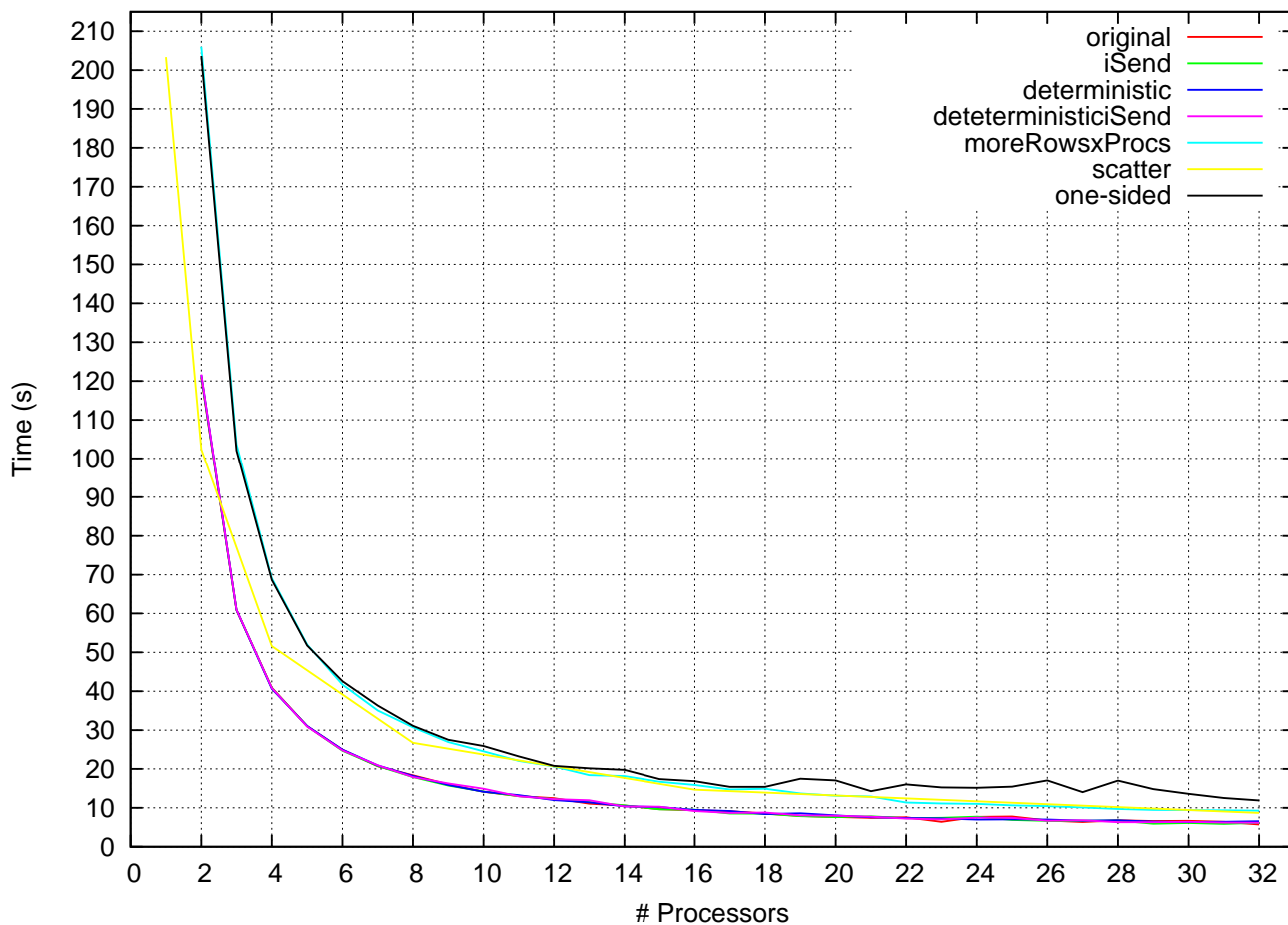
The executions of the *One-Sided Communications Version* have showed the follow results:



Finally, the last version, is much different from the others. It uses *One-Sided Communications Paradigm*, so the process master stores information about matrix A and B. It creates a window, using *MPI.Win_create*, that links space address of matrix A, B and C. So master process shares its memory with the others processors. Then each process using *MPI.Get* can obtain info about matrix A and B, computes a piece of result and stores it in the space address of matrix C using *MPI.Put*. All this operations are synchronized thru *MPI.Win_fence* function.

3 Compare and Conclusion

The graphic below shows the execution times of all versions of the program. We can notice that more or less, all versions are mutually equivalent to, but obviously there are versions with higher performance than others. The first four versions, are that with higher performance, *Scatter-Gather Version* and *More Rows per Process Version* initially with a small number of processors are very different but with the increasing of the number of processors they converge to the same values. Finally, the worst version is the *One-Sided Communications Version*, that with a small number of processors move together the *More Rows per Process Version*, but with the increasing of the number of processors it gets worse in respect to the other versions.



References

- [1] (EC)² 2009: Challenge problems. <http://vsl.cis.udel.edu/ec2/challenge.html>, 2009.
- [2] William Gropp, Ewing Lusk, and Anthony Skjellum. Using mpi: Portable parallel programming with the message-passing interface. <http://www.mcs.anl.gov/research/projects/mpi/usingmpi/>.
- [3] Stephen F. Siegel. <http://www.cis.udel.edu/~siegel/>.

A Appendix - Code

This Appendix shows the code of the all versions.

A.1 Original Version

```
0 #include <stdlib.h>
1 #include <stdio.h>
2 #include <mpi.h>
3 // #include <isp.h>
4
5 #define MAX_ROWS 1024
6 #define MAX_COLS 2048
7 #define MAX_BROWS MAX_COLS
8 #define MAX_BCOLS 1024
9 #define MAX_CROWS MAX_ROWS
10 #define MAX_CCOLS MAX_BCOLS
11 #define MAX_A_ENTRIES (MAX_ROWS * MAX_COLS)
12 #define MAX_B_ENTRIES (MAX_BROWS * MAX_BCOLS)
13 #define MAX_C_ENTRIES (MAX_ROWS * MAX_BCOLS)
14
15 int main(int argc, char *argv[]) {
16
17     float *a, *b, *c;
18     float *buffer, *ans;
19
20     int myid, master, numprocs, ierr;
21     int i, j, numsent, sender;
22     int anstype, row, arows, acols, brows, bcols, crows, ccols;
23     MPI_Status status;
24     FILE *times;
25     double t_start, t_end, start, end;
26     char namefile[10];
27
28     MPI_Init(&argc, &argv);
29     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
30     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
31
32     master = 0;
33     arows = MAX_ROWS;
34     acols = MAX_COLS;
35     brows = MAX_BROWS;
36     bcols = MAX_BCOLS;
37     crows = arows;
38     ccols = bcols;
39
40     a = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
41     b = (float *) malloc(MAX_B_ENTRIES * sizeof(float));
42     c = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
43     buffer = (float *) malloc(MAX_COLS * sizeof(float));
44     ans = (float *) malloc(MAX_COLS * sizeof(float));
45
46     if (myid == master) {
47
48         /* create the times file */
49         times = fopen("original-times.log", "a");
50
51         if(times == NULL) {
52             printf("Error Open File!");
53             return(0);
54         }
55
56         /* read a, b */
57
58         for (i = 0; i < MAX_ROWS; i++)
59             for (j = 0; j < MAX_COLS; j++)
60                 a[i*MAX_COLS + j] = (float) (i*10 + j);
61
62         for (i = 0; i < MAX_BROWS; i++)
63             for (j = 0; j < MAX_BCOLS; j++)
64                 b[i*MAX_BCOLS + j] = (float) (i*10 + j);
65
66         /* finished reading */
67         numsent = 0;
```

```

68
69      /* Total Start */
70      t_start = MPI_Wtime();
71
72      MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
73      for (i = 0; i < numprocs-1; i++) {
74          for (j = 0; j < acols; j++) {
75              buffer[j] = a[i*acols+j];
76          }
77
78          /* Start Time */
79          start = MPI_Wtime();
80
81          MPI_Send(buffer, acols, MPI_FLOAT, i+1, i+1, MPI_COMM_WORLD);
82          numsent++;
83      }
84      for (i = 0; i < crows; i++) {
85          MPI_Recv(ans, ccols, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
86                 MPI_COMM_WORLD, &status);
87
88          /* End Time */
89          end = MPI_Wtime();
90
91          /* Writes results */
92          sender = status.MPI_SOURCE;
93
94          anstype = status.MPI_TAG - 1;
95          for (j = 0; j < ccols; j++) {
96              c[anstype*ccols+j] = ans[j];
97          }
98          if (numsent < arows) {
99              for (j = 0; j < acols; j++) {
100                 buffer[j] = a[numsent*acols+j];
101             }
102             MPI_Send(buffer, acols, MPI_FLOAT, sender, numsent+1, MPI_COMM_WORLD);
103             numsent++;
104             }
105             else {
106                 MPI_Send(buffer, 1, MPI_FLOAT, sender, 0, MPI_COMM_WORLD);
107             }
108         }
109
110         /* Total End */
111         t_end = MPI_Wtime();
112
113         fprintf(times, "\n%d %lg", numprocs, t_end - t_start);
114
115         fclose(times);
116         /*
117         for (i = 0; i < MAX_CROWS; i++) {
118             for (j = 0; j < MAX_CCOLS; j++) {
119                 printf("%f\t", c[i*MAX_CCOLS + j]);
120                 printf("\n");
121             }
122         */
123         printf("\n\nDone! %e TIME: %g", c[MAX_C_ENTRIES - 1], t_end - t_start);
124         printf("\n\n");
125     }
126 }
127 else {
128     MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
129     while (1) {
130         MPI_Recv(buffer, acols, MPI_FLOAT, master, MPI_ANY_TAG,
131                MPI_COMM_WORLD, &status);
132         if (status.MPI_TAG == 0) break;
133         row = status.MPI_TAG - 1;
134         for (i = 0; i < bcols; i++) {
135             ans[i] = 0.0;
136             for (j = 0; j < acols; j++) {
137                 ans[i] += buffer[j]*b[j*bcols+i];
138             }
139         }
140         MPI_Send(ans, bcols, MPI_FLOAT, master, row+1, MPI_COMM_WORLD);
141     }
142 }
143 MPI_Finalize();
144 }

```

A.2 Original vs. Sequential Version

```
0 #include <stdlib.h>
1 #include <stdio.h>
2 #include <mpi.h>
3 #include <assert.h>
4 // #include <isp.h>
5
6 #define MAX_ROWS 4//1024
7 #define MAX_COLS 8//2048
8 #define MAX_BROWS MAX_COLS
9 #define MAX_BCOLS 4//1024
10 #define MAX_CROWS MAX_ROWS
11 #define MAX_CCOLS MAX_BCOLS
12 #define MAX_A_ENTRIES (MAX_ROWS * MAX_COLS)
13 #define MAX_B_ENTRIES (MAX_BROWS * MAX_BCOLS)
14 #define MAX_C_ENTRIES (MAX_ROWS * MAX_BCOLS)
15
16 int main(int argc, char *argv[]) {
17     float *a, *b, *c, **d, **e, **f;
18     float *buffer, *ans;
19     int myid, master, numprocs, ierr;
20     int i, j, k, numsent, sender;
21     int anstype, row, arows, acols, brows, bcols, crows, ccols;
22     MPI_Status status;
23     char namefile[10];
24
25     master = 0;
26     arows = MAX_ROWS;
27     acols = MAX_COLS;
28     brows = MAX_BROWS;
29     bcols = MAX_BCOLS;
30     crows = arows;
31     ccols = bcols;
32
33     {
34         /* Sequential Version */
35
36         d = (float **) malloc(MAX_ROWS * sizeof(float));
37         for(i = 0; i < MAX_ROWS; i++)
38             d[i] = (float *) malloc(MAX_COLS * sizeof(float));
39
40         e = (float **) malloc(MAX_BROWS * sizeof(float));
41         for(i = 0; i < MAX_BROWS; i++)
42             e[i] = (float *) malloc(MAX_BCOLS * sizeof(float));
43
44         f = (float **) malloc(MAX_ROWS * sizeof(float));
45         for(i = 0; i < MAX_ROWS; i++)
46             f[i] = (float *) malloc(MAX_BCOLS * sizeof(float));
47
48         for (i = 0; i < MAX_ROWS; i++)
49             for (j = 0; j < MAX_COLS; j++)
50                 d[i][j] = (float) (i*10 + j);
51
52         for (i = 0; i < MAX_BROWS; i++)
53             for (j = 0; j < MAX_BCOLS; j++)
54                 e[i][j] = (float) (i*10 + j);
55
56         for (i = 0; i < arows; i++) {
57             for (j = 0; j < bcols; j++)
58                 for (k = 0, f[i][j] = 0.0; k < brows; k++)
59                     f[i][j] += d[i][k]*e[k][j];
60         }
61     }
62
63     /* Parallel Version */
64     {
65         MPI_Init(&argc, &argv);
66         MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
67         MPI_Comm_rank(MPI_COMM_WORLD, &myid);
68
69         a = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
70         b = (float *) malloc(MAX_B_ENTRIES * sizeof(float));
71         c = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
72         buffer = (float *) malloc(MAX_COLS * sizeof(float));
73         ans = (float *) malloc(MAX_COLS * sizeof(float));
```

```

74
75     if (myid == master) {
76
77         /* read a, b */
78
79         for (i = 0; i < MAX_AROWS; i++)
80     for (j = 0; j < MAX_ACOLS; j++)
81     a[i*MAX_ACOLS + j] = (float) (i*10 + j);
82
83         for (i = 0; i < MAX_BROWS; i++)
84     for (j = 0; j < MAX_BCOLS; j++)
85     b[i*MAX_BCOLS + j] = (float) (i*10 + j);
86
87         /* finished reading */
88     numsent = 0;
89
90     MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
91     for (i = 0; i < numprocs-1; i++) {
92     for (j = 0; j < acols; j++) {
93     buffer[j] = a[i*acols+j];
94     }
95
96     MPI_Send(buffer, acols, MPI_FLOAT, i+1, i+1, MPI_COMM_WORLD);
97     numsent++;
98     }
99     for (i = 0; i < crows; i++) {
100 MPI_Recv(ans, ccols, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
101 MPI_COMM_WORLD, &status);
102
103 /* Writes results */
104 sender = status.MPI_SOURCE;
105
106 anstype = status.MPI_TAG - 1;
107 for (j = 0; j < ccols; j++) {
108     c[anstype*ccols+j] = ans[j];
109 }
110 if (numsent < arows) {
111     for (j = 0; j < acols; j++) {
112     buffer[j] = a[numsent*acols+j];
113     }
114     MPI_Send(buffer, acols, MPI_FLOAT, sender, numsent+1, MPI_COMM_WORLD);
115     numsent++;
116 }
117 else {
118     MPI_Send(buffer, 1, MPI_FLOAT, sender, 0, MPI_COMM_WORLD);
119 }
120 }
121
122 /*
123 printf("\nParallel Result:\n");
124 for (i = 0; i < MAX_CROWS; i++) {
125     for (j = 0; j < MAX_CCOLS; j++) {
126     printf("%f\t", c[i*MAX_CCOLS + j]);
127     }
128     printf("\n");
129 }
130
131 printf("\nSequential Result:\n");
132 for (i = 0; i < MAX_CROWS; i++) {
133     for (j = 0; j < MAX_CCOLS; j++) {
134     printf("%f\t", f[i][j]);
135     }
136     printf("\n");
137 }
138 */
139
140 /* Assertions */
141 for (i = 0; i < MAX_CROWS; i++) {
142     for (j = 0; j < MAX_CCOLS; j++) {
143     assert(c[i * MAX_CCOLS + j] == f[i][j]);
144     }
145 }
146
147 }
148 else {
149     MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
150     while (1) {
151     MPI_Recv(buffer, acols, MPI_FLOAT, master, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

```

```

152     if (status.MPI_TAG == 0) break;
153     row = status.MPI_TAG - 1;
154     for (i = 0; i < bcols; i++) {
155         ans[i] = 0.0;
156         for (j = 0; j < acols; j++) {
157             ans[i] += buffer[j]*b[j*bcols+i];
158         }
159     }
160     MPI_Send(ans, bcols, MPI_FLOAT, master, row+1, MPI_COMM_WORLD);
161 }
162 }
163 MPI_Finalize();
164 }
165 }

```

A.3 Original Non-Blocking Version

```

0  #include <stdio.h>
1  #include <stdlib.h>
2  #include <mpi.h>
3  // #include <isp.h>
4
5  #define MAX_ROWS 1024
6  #define MAX_COLS 2048
7  #define MAX_BROWS MAX_COLS
8  #define MAX_BCOLS 1024
9  #define MAX_CROWS MAX_ROWS
10 #define MAX_CCOLS MAX_BCOLS
11 #define MAX_A_ENTRIES (MAX_ROWS * MAX_COLS)
12 #define MAX_B_ENTRIES (MAX_BROWS * MAX_BCOLS)
13 #define MAX_C_ENTRIES (MAX_ROWS * MAX_BCOLS)
14
15 int main(int argc, char *argv[]) {
16
17     float *a, *b, *c;
18     float *buffer, *ans;
19
20     int myid, master, numprocs, ierr;
21     int i, j, numsent, sender;
22     int anstype, row, arows, acols, brows, bcols, crows, ccols;
23     MPI_Status status;
24     MPI_Request request;
25     FILE *times;
26     double t_start, t_end, start, end;
27     char namefile[10];
28
29     MPI_Init(&argc, &argv);
30     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
31     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
32
33     master = 0;
34     arows = MAX_ROWS;
35     acols = MAX_COLS;
36     brows = MAX_BROWS;
37     bcols = MAX_BCOLS;
38     crows = arows;
39     ccols = bcols;
40
41     a = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
42     b = (float *) malloc(MAX_B_ENTRIES * sizeof(float));
43     c = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
44     buffer = (float *) malloc(MAX_COLS * sizeof(float));
45     ans = (float *) malloc(MAX_COLS * sizeof(float));
46
47     if (myid == master) {
48
49         /* create the times file */
50         times = fopen("iSend.times.log", "a");
51
52         if (times == NULL) {
53             printf("Error Open File!");
54             return(0);
55         }
56

```

```

57  /* read a, b */
58
59  for (i = 0; i < MAX_AROWS; i++)
60      for (j = 0; j < MAX_ACOLS; j++)
61          a[i*MAX_ACOLS + j] = (float) (i*10 + j);
62
63  for (i = 0; i < MAX_BROWS; i++)
64      for (j = 0; j < MAX_BCOLS; j++)
65          b[i*MAX_BCOLS + j] = (float) (i*10 + j);
66
67  /* finished reading */
68  numsent = 0;
69
70  /* Total Start */
71  t_start = MPI_Wtime();
72
73  MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
74  for (i = 0; i < numprocs-1; i++) {
75      for (j = 0; j < acols; j++) {
76          buffer[j] = a[i*acols+j];
77      }
78
79      MPI_Isend(buffer, acols, MPI_FLOAT, i+1, i+1, MPI_COMM_WORLD,&request);
80      numsent++;
81  }
82  for (i = 0; i < crows; i++) {
83      MPI_Wait(&request,&status);
84      MPI_Recv(ans, ccols, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
85              MPI_COMM_WORLD, &status);
86
87      sender = status.MPI_SOURCE;
88      anstype = status.MPI_TAG - 1;
89      for (j = 0; j < ccols; j++) {
90          c[anstype*ccols+j] = ans[j];
91      }
92      if (numsent < arows) {
93          for (j = 0; j < acols; j++) {
94              buffer[j] = a[numsent*acols+j];
95          }
96          MPI_Isend(buffer, acols, MPI_FLOAT, sender, numsent+1, MPI_COMM_WORLD,&request);
97          numsent++;
98      }
99      else {
100         MPI_Isend(buffer, 1, MPI_FLOAT, sender, 0, MPI_COMM_WORLD,&request);
101     }
102 }
103
104
105 /* Total End */
106 t_end = MPI_Wtime();
107
108
109 fprintf(times, "\n%d %lg", numprocs, t_end - t_start);
110
111 fclose(times);
112 /*
113 for (i = 0; i < MAX_CROWS; i++) {
114     for (j = 0; j < MAX_CCOLS; j++) {
115         printf("%f\t", c[i*MAX_CCOLS + j]);
116         printf("\n");
117     }
118 */
119 printf("\n\nDone! %e TIME: %g", c[MAX_C_ENTRIES - 1], t_end - t_start);
120 printf("\n\n");
121 }
122 else {
123     MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
124     while (1) {
125         MPI_Recv(buffer, acols, MPI_FLOAT, master, MPI_ANY_TAG,
126                 MPI_COMM_WORLD, &status);
127         if (status.MPI_TAG == 0) break;
128         row = status.MPI_TAG - 1;
129         for (i = 0; i < bcols; i++) {
130             ans[i] = 0.0;
131             for (j = 0; j < acols; j++) {
132                 ans[i] += buffer[j]*b[j*bcols+i];
133             }
134         }

```

```

135     MPI_Send(ans, bcols, MPI_FLOAT, master, row+1, MPI_COMM_WORLD);
136     }
137 }
138 MPI_Finalize();
139 }

```

A.4 Deterministic Version

```

0  #include <stdlib.h>
1  #include <stdio.h>
2  #include <mpi.h>
3  // #include <isp.h>
4
5  #define MAX_AROWS 1024
6  #define MAX_ACOLS 2048
7  #define MAX_BROWS MAX_ACOLS
8  #define MAX_BCOLS 1024
9  #define MAX_CROWS MAX_AROWS
10 #define MAX_CCOLS MAX_BCOLS
11 #define MAX_A_ENTRIES (MAX_AROWS * MAX_ACOLS)
12 #define MAX_B_ENTRIES (MAX_BROWS * MAX_BCOLS)
13 #define MAX_C_ENTRIES (MAX_AROWS * MAX_BCOLS)
14
15 int main(int argc, char *argv[]) {
16
17     float *a, *b, *c;
18     float *buffer, *ans;
19
20     int myid, master, numprocs, ierr;
21     int i, j, numsent, sender;
22     int anstype, row, arows, acols, brows, bcols, crows, ccols;
23     MPI_Status status;
24     FILE *times;
25     double t_start, t_end, start, end;
26     char namefile[10];
27     int *process;
28
29     MPI_Init(&argc, &argv);
30     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
31     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
32
33     master = 0;
34     arows = MAX_AROWS;
35     acols = MAX_ACOLS;
36     brows = MAX_BROWS;
37     bcols = MAX_BCOLS;
38     crows = arows;
39     ccols = bcols;
40
41     a = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
42     b = (float *) malloc(MAX_B_ENTRIES * sizeof(float));
43     c = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
44     buffer = (float *) malloc(MAX_ACOLS * sizeof(float));
45     ans = (float *) malloc(MAX_ACOLS * sizeof(float));
46
47     if (myid == master) {
48
49         /* create the times file */
50         times = fopen("deterministic_times.log", "a");
51
52         if (times == NULL) {
53             printf("Error Open File!");
54             return(0);
55         }
56
57         /* match crow - process */
58         process = (int *) malloc(crows * sizeof(int));
59
60         /* read a, b */
61
62         for (i = 0; i < MAX_AROWS; i++)
63             for (j = 0; j < MAX_ACOLS; j++)
64                 a[i*MAX_ACOLS + j] = (float) (i*10 + j);
65

```

```

66     for (i = 0; i < MAX_BROWS; i++)
67         for (j = 0; j < MAX_BCOLS; j++)
68     b[i*MAX_BCOLS + j] = (float) (i*10 + j);
69
70     /* finished reading */
71     numsent = 0;
72
73     /* Total Start */
74     t_start = MPI_Wtime();
75
76     MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
77     for (i = 0; i < numprocs-1; i++) {
78         for (j = 0; j < acols; j++) {
79             buffer[j] = a[i*acols+j];
80         }
81
82         MPI_Send(buffer, acols, MPI_FLOAT, i+1, i+1, MPI_COMM_WORLD);
83         numsent++;
84         process[numsent-1] = i+1;
85     }
86     for (i = 0; i < crows; i++) {
87         MPI_Recv(ans, ccols, MPI_FLOAT, process[i], MPI_ANY_TAG,
88             MPI_COMM_WORLD, &status);
89
90         /* Writes results */
91         sender = status.MPI_SOURCE;
92
93         anstype = status.MPI_TAG - 1;
94         for (j = 0; j < ccols; j++) {
95             c[anstype*ccols+j] = ans[j];
96         }
97         if (numsent < arows) {
98             for (j = 0; j < acols; j++) {
99                 buffer[j] = a[numsent*acols+j];
100             }
101             MPI_Send(buffer, acols, MPI_FLOAT, sender, numsent+1, MPI_COMM_WORLD);
102             numsent++;
103             process[numsent-1] = sender;
104         }
105         else {
106             MPI_Send(buffer, 1, MPI_FLOAT, sender, 0, MPI_COMM_WORLD);
107         }
108     }
109
110     /* Total End */
111     t_end = MPI_Wtime();
112
113     fprintf(times, "\n%d %lg", numprocs, t_end - t_start);
114
115     fclose(times);
116     /*
117     for (i = 0; i < MAX_CROWS; i++) {
118         for (j = 0; j < MAX_CCOLS; j++) {
119             printf("%f\t", c[i*MAX_CCOLS + j]);
120             printf("\n");
121         }
122     */
123     printf("\n\nDone! %e TIME: %g", c[MAX_C_ENTRIES - 1], t_end - t_start);
124     printf("\n\n");
125
126 }
127 else {
128     MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
129     while (1) {
130         MPI_Recv(buffer, acols, MPI_FLOAT, master, MPI_ANY_TAG,
131             MPI_COMM_WORLD, &status);
132         if (status.MPI_TAG == 0) break;
133         row = status.MPI_TAG - 1;
134         for (i = 0; i < bcols; i++) {
135             ans[i] = 0.0;
136             for (j = 0; j < acols; j++) {
137                 ans[i] += buffer[j]*b[j*bcols+i];
138             }
139         }
140         MPI_Send(ans, bcols, MPI_FLOAT, master, row+1, MPI_COMM_WORLD);
141     }
142 }
143 MPI_Finalize();

```

A.5 Deterministic Non-Blocking Version

```

0  #include <stdlib.h>
1  #include <stdio.h>
2  // #include <mpi.h>
3  #include <isp.h>
4
5  #define MAXAROWS 1024
6  #define MAXACOLS 2048
7  #define MAXBROWS MAXACOLS
8  #define MAXBCOLS 1024
9  #define MAXCROWS MAXAROWS
10 #define MAXCCOLS MAXBCOLS
11 #define MAX_A_ENTRIES (MAXAROWS * MAXACOLS)
12 #define MAX_B_ENTRIES (MAXBROWS * MAXBCOLS)
13 #define MAX_C_ENTRIES (MAXAROWS * MAXBCOLS)
14
15 int main(int argc, char *argv[]) {
16
17     float *a, *b, *c;
18     float *buffer, *ans;
19
20
21     int myid, master, numprocs, ierr;
22     int i, j, numsent, sender;
23     int anstype, row, arows, acols, brows, bcols, crows, ccols;
24     MPI_Status status;
25     MPI_Request request;
26     FILE *times;
27     double t_start, t_end, start, end;
28     char namefile[10];
29     int *process;
30
31     MPI_Init(&argc, &argv);
32     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
33     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
34
35     master = 0;
36     arows = MAX_AROWS;
37     acols = MAX_ACOLS;
38     brows = MAX_BROWS;
39     bcols = MAX_BCOLS;
40     crows = arows;
41     ccols = bcols;
42
43     a = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
44     b = (float *) malloc(MAX_B_ENTRIES * sizeof(float));
45     c = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
46     buffer = (float *) malloc(MAX_ACOLS * sizeof(float));
47     ans = (float *) malloc(MAX_ACOLS * sizeof(float));
48
49     if (myid == master) {
50
51         /* create the times file */
52         times = fopen("iSendD.times.log", "a");
53
54         if (times == NULL) {
55             printf("Error Open File!");
56             return(0);
57         }
58
59         /* match crow - process */
60         process = (int *) malloc(crows * sizeof(int));
61
62         /* read a, b */
63
64         for (i = 0; i < MAX_AROWS; i++)
65             for (j = 0; j < MAX_ACOLS; j++)
66                 a[i*MAX_ACOLS + j] = (float) (i*10 + j);
67
68         for (i = 0; i < MAX_BROWS; i++)
69             for (j = 0; j < MAX_BCOLS; j++)

```

```

70     b[i*MAX_BCOLS + j] = (float) (i*10 + j);
71
72     /* finished reading */
73     numsent = 0;
74
75     /* Total Start */
76     t_start = MPI_Wtime();
77
78     MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
79     for (i = 0; i < numprocs-1; i++) {
80         for (j = 0; j < acols; j++) {
81             buffer[j] = a[i*acols+j];
82         }
83
84         MPI_Isend(buffer, acols, MPI_FLOAT, i+1, i+1, MPI_COMM_WORLD,&request);
85         numsent++;
86         process[numsent-1] = i+1;
87     }
88     for (i = 0; i < crows; i++) {
89         MPI_Wait(&request,&status);
90         MPI_Recv(ans, ccols, MPI_FLOAT, process[i], MPI_ANY_TAG,
91             MPI_COMM_WORLD, &status);
92
93         /* Writes results */
94         sender = status.MPI_SOURCE;
95
96         anstype = status.MPI_TAG - 1;
97         for (j = 0; j < ccols; j++) {
98             c[anstype*ccols+j] = ans[j];
99         }
100        if (numsent < arows) {
101            for (j = 0; j < acols; j++) {
102                buffer[j] = a[numsent*acols+j];
103            }
104            MPI_Isend(buffer, acols, MPI_FLOAT, sender, numsent+1, MPI_COMM_WORLD,&request);
105            numsent++;
106            process[numsent-1] = sender;
107        }
108        else {
109            MPI_Isend(buffer, 1, MPI_FLOAT, sender, 0, MPI_COMM_WORLD,&request);
110        }
111    }
112
113    /* Total End */
114    t_end = MPI_Wtime();
115
116    fprintf(times, "\n%d %lg", numprocs, t_end - t_start);
117
118    fclose(times);
119    /*
120    for (i = 0; i < MAX_CROWS; i++) {
121        for (j = 0; j < MAX_CCOLS; j++) {
122            printf("%f\t", c[i*MAX_CCOLS + j]);
123            printf("\n");
124        }
125    */
126    printf("\n\nDone! %e TIME: %g", c[MAX_C_ENTRIES - 1], t_end - t_start);
127    printf("\n\n");
128
129 }
130 else {
131     MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
132     while (1) {
133         MPI_Recv(buffer, acols, MPI_FLOAT, master, MPI_ANY_TAG,
134             MPI_COMM_WORLD, &status);
135         if (status.MPI_TAG == 0) break;
136         row = status.MPI_TAG - 1;
137         for (i = 0; i < bcols; i++) {
138             ans[i] = 0.0;
139             for (j = 0; j < acols; j++) {
140                 ans[i] += buffer[j]*b[j*bcols+i];
141             }
142         }
143         MPI_Send(ans, bcols, MPI_FLOAT, master, row+1, MPI_COMM_WORLD);
144     }
145 }
146 MPI_Finalize();
147 }

```

A.6 More Rows per Process Version

```
0 #include <stdlib.h>
1 #include <stdio.h>
2 // #include <mpi.h>
3 #include <isp.h>
4
5 #define MAX_ROWS 1024
6 #define MAX_COLS 2048
7 #define MAX_BROWS MAX_COLS
8 #define MAX_BCOLS 1024
9 #define MAX_CROWS MAX_ROWS
10 #define MAX_CCOLS MAX_BCOLS
11 #define MAX_A_ENTRIES (MAX_ROWS * MAX_COLS)
12 #define MAX_B_ENTRIES (MAX_BROWS * MAX_BCOLS)
13 #define MAX_C_ENTRIES (MAX_ROWS * MAX_BCOLS)
14
15 int main(int argc, char *argv[]) {
16
17     float *a, *b, *c;
18     float *buffer, *ans;
19
20     int myid, master, numprocs, ierr;
21     int i, j, numsent, sender;
22     int anstype, row, arows, acols, brows, bcols, crows, ccols;
23     MPI_Status status;
24     FILE *times;
25     double t_start, t_end, start, end;
26     char namefile[10];
27     int nrows, elements, k, prows;
28
29     MPI_Init(&argc, &argv);
30     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
31     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
32
33     master = 0;
34     arows = MAX_ROWS;
35     acols = MAX_COLS;
36     brows = MAX_BROWS;
37     bcols = MAX_BCOLS;
38     crows = arows;
39     ccols = bcols;
40
41     a = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
42     b = (float *) malloc(MAX_B_ENTRIES * sizeof(float));
43     c = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
44     buffer = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
45     ans = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
46
47     nrows = crows / (numprocs - 1);
48     elements = acols * nrows;
49
50     if (myid == master) {
51
52         /* create the times file */
53         times = fopen("morerowsprocs-times.log", "a");
54
55         if (times == NULL) {
56             printf("Error Open File!");
57             return(0);
58         }
59
60         /* read a, b */
61
62         for (i = 0; i < MAX_ROWS; i++) {
63             // printf("\n");
64             for (j = 0; j < MAX_COLS; j++) {
65                 a[i*MAX_COLS + j] = (float) (i*10 + j);
66                 // printf(" %g ", a[i*MAX_COLS + j]);
67             }
68         }
69
70         printf("\n");
71
72         for (i = 0; i < MAX_BROWS; i++) {
73             // printf("\n");
```

```

74     for (j = 0; j < MAX_BCOLS; j++) {
75     b[i*MAX_BCOLS + j] = (float) (i*10 + j);
76     //printf(" %g  ",b[i*MAX_BCOLS + j]);
77     }
78     }
79
80     // finished reading
81     numsent = 0;
82
83     // Total Start
84     t_start = MPI_Wtime();
85
86     // printf("\n\n Elementi in broadcast %d",brows * bcols);
87     MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
88
89     for (i = 0; i < numprocs - 1; i++) {
90         for (j = 0; j < elements; j++) {
91             buffer[j] = a[(acols * nrows * i) + j];
92         }
93         // printf("\nProcesso: %d # elementi: %d",i+1,elements);
94         MPI_Send(buffer, elements, MPI_FLOAT, i+1, i+1, MPI_COMM_WORLD);
95     }
96
97     // Master compute its part of matrix
98     prows = nrows * (numprocs - 1);
99     row = arows - prows;
100    if(row > 0) {
101        for(k = 0; k < row; k++) {
102            for(i = 0; i < bcols; i++) {
103                c[(prows * ccols) + (ccols * k) + i] = 0.0;
104                for(j = 0; j < acols; j++) {
105                    c[(prows * ccols) + (ccols * k) + i] += a[(prows * acols) + (k * acols) + j] * b[(j * bcols)←
+ i];
106                }
107            }
108        }
109    }
110
111    for (i = 0; i < numprocs - 1; i++) {
112        MPI_Recv(ans, bcols * nrows, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
113
114        sender = status.MPI_SOURCE;
115        for(k = 0; k < nrows; k++) {
116            for(j = 0; j < ccols; j++) {
117                c[(nrows * ccols * (sender - 1)) + (ccols * k) + j] = ans[(ccols * k) + j];
118            }
119        }
120    }
121
122    // Total End
123    t_end = MPI_Wtime();
124
125    fprintf(times, "\n%d %lg", numprocs, t_end - t_start);
126
127    fclose(times);
128
129    // printf("\n\n");
130    for (i = 0; i < MAX_CROWS; i++) {
131        // printf("\n\n");
132        for (j = 0; j < MAX_CCOLS; j++) {
133            // printf("%g\t", c[(i * MAX_CCOLS) + j]);
134        }
135    }
136    //printf("\n\n");
137    printf("\n\nDone! %e Time %g\n\n", c[MAX_C_ENTRIES - 1], t_end - t_start);
138 }
139 else {
140     MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
141     MPI_Recv(buffer, elements, MPI_FLOAT, master, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
142
143     for(k = 0; k < nrows; k++) {
144         for(i = 0; i < bcols; i++) {
145             ans[i + (k * bcols)] = 0.0;
146             for(j = 0; j < acols; j++) {
147                 ans[i + (k * bcols)] += buffer[j + (k * acols)] * b[j * bcols + i];
148             }
149         }
150     }

```

```

151     MPI_Send(ans, bcols * nrows, MPI_FLOAT, master, status.MPI_TAG, MPI_COMM_WORLD);
152 }
153 MPI_Finalize();
154 }

```

A.7 Scatter-Gather Version

```

0  #include <stdlib.h>
1  #include <stdio.h>
2  // #include <mpi.h>
3  #include <isp.h>
4
5  #define MAX_AROWS 1024
6  #define MAX_ACOLS 2048
7  #define MAX_BROWS MAX_ACOLS
8  #define MAX_BCOLS 1024
9  #define MAX_CROWS MAX_AROWS
10 #define MAX_CCOLS MAX_BCOLS
11 #define MAX_A_ENTRIES (MAX_AROWS * MAX_ACOLS)
12 #define MAX_B_ENTRIES (MAX_BROWS * MAX_BCOLS)
13 #define MAX_C_ENTRIES (MAX_AROWS * MAX_BCOLS)
14
15 int main(int argc, char *argv[]) {
16
17     float *a, *b, *c;
18     float *buffer, *ans;
19
20     int myid, master, numprocs, ierr;
21     int i, j, numsent, sender;
22     int anstype, row, arows, acols, brows, bcols, crows, ccols;
23     MPI_Status status;
24     FILE *times;
25     double t_start, t_end, start, end;
26     char namefile[10];
27     int nrows, elements, k, prows;
28
29     MPI_Init(&argc, &argv);
30     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
31     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
32
33     master = 0;
34     arows = MAX_AROWS;
35     acols = MAX_ACOLS;
36     brows = MAX_BROWS;
37     bcols = MAX_BCOLS;
38     crows = arows;
39     ccols = bcols;
40
41     if((arows % numprocs) != 0) {
42         if(myid == master) {
43             printf("\nThe number of processors must be a submultiple of number of rows!");
44             printf("\nThe number of rows is: %d\n\n", crows);
45         }
46         MPI_Finalize();
47         return(0);
48     }
49
50     a = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
51     b = (float *) malloc(MAX_B_ENTRIES * sizeof(float));
52     c = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
53     buffer = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
54     ans = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
55
56     nrows = crows / (numprocs);
57     elements = acols * nrows;
58
59     if (myid == master) {
60
61         // create the times file
62         times = fopen("scatter-times.log", "a");
63
64         if(times == NULL) {
65             printf("Error Open File!");
66             return(0);

```

```

67     }
68
69     // read a, b
70
71     for (i = 0; i < MAX_ROWS; i++) {
72         //printf("\n");
73         for (j = 0; j < MAX_COLS; j++) {
74             a[i*MAX_COLS + j] = (float) (i*10 + j);
75             //printf(" %g ",a[i*MAX_COLS + j]);
76         }
77     }
78
79     printf("\n");
80
81     for (i = 0; i < MAX_BROWS; i++) {
82         //printf("\n");
83         for (j = 0; j < MAX_BCOLS; j++) {
84             b[i*MAX_BCOLS + j] = (float) (i*10 + j);
85             //printf(" %g ",b[i*MAX_BCOLS + j]);
86         }
87     }
88
89     // finished reading
90     numsent = 0;
91
92     // Total Start
93     t_start = MPI_Wtime();
94 }
95 // printf("\n\n Elementi in broadcast %d",brows * bcols);
96 MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
97
98 MPI_Scatter(a, elements, MPI_FLOAT,a, elements, MPI_FLOAT,0, MPI_COMM_WORLD);
99
100 for(k = 0; k < nrows; k++) {
101     for(i = 0; i < bcols; i++) {
102         c[i + (k * bcols)] = 0.0;
103         for(j = 0; j < acols; j++) {
104             c[i + (k * bcols)] += a[j + (k * acols)] * b[j * bcols + i];
105             //printf("\nC %f",c[i + (k * bcols)]);
106         }
107     }
108 }
109
110 MPI_Gather(c, nrows * ccols, MPI_FLOAT, c, nrows * ccols, MPI_FLOAT,0, MPI_COMM_WORLD);
111
112 if(myid == 0) {
113     t_end = MPI_Wtime();
114
115     fprintf(times, "\n%d %lg", numprocs, t_end - t_start);
116
117     fclose(times);
118
119     // Print results
120     printf("\n\nDone! %e TIME: %g", c[MAX_C_ENTRIES - 1], t_end - t_start);
121     printf("\n\n");
122     /*     printf("\n\n");
123         for(i = 0; i < MAXCROWS; i++) {
124             printf("\n");
125             for(j = 0; j < MAXCCOLS; j++)
126                 printf(" %f\t",c[i * MAXCCOLS + j]);
127             */
128 }
129
130 MPI_Finalize();
131 }

```

A.8 One-Sided Communications Version

```

0 #include <stdlib.h>
1 #include <stdio.h>
2 #include <mpi.h>
3 // #include <isp.h>
4
5 #define MAX_ROWS 1024

```

```

6 #define MAX_COLS 2048
7 #define MAX_ROWS MAX_COLS
8 #define MAX_COLS 1024
9 #define MAX_ROWS MAX_COLS
10 #define MAX_COLS MAX_COLS
11 #define MAX_A_ENTRIES (MAX_ROWS * MAX_COLS)
12 #define MAX_B_ENTRIES (MAX_ROWS * MAX_COLS)
13 #define MAX_C_ENTRIES (MAX_ROWS * MAX_COLS)
14
15 int main(int argc, char *argv[]) {
16
17     float *a, *b, *c;
18     float *buffer, *ans;
19
20     int myid, master, numprocs, ierr;
21     int i, j, numsent, sender;
22     int anstype, row, arows, acols, brows, bcols, crows, ccols;
23     MPI_Status status;
24     FILE *times;
25     double t_start, t_end, start, end;
26     char namefile[10];
27     int *process;
28     int nrows, elements, k, prows;
29     MPI_Win winA, winB, winC;
30     MPI_Group comm_group, group;
31
32     MPI_Init(&argc, &argv);
33     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
34     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
35
36     master = 0;
37     arows = MAX_ROWS;
38     acols = MAX_COLS;
39     brows = MAX_ROWS;
40     bcols = MAX_COLS;
41     crows = arows;
42     ccols = bcols;
43
44     a = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
45     b = (float *) malloc(MAX_B_ENTRIES * sizeof(float));
46     c = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
47     buffer = (float *) malloc(MAX_A_ENTRIES * sizeof(float));
48     ans = (float *) malloc(MAX_C_ENTRIES * sizeof(float));
49
50     nrows = crows / (numprocs - 1);
51     elements = acols * nrows;
52
53     if (myid == master) {
54
55         /* create the times file */
56         times = fopen("one-sided.times.log", "a");
57
58         if (times == NULL) {
59             printf("Error Open File!");
60             return(0);
61         }
62
63         /* read a, b */
64
65         for (i = 0; i < MAX_ROWS; i++) {
66             //printf("\n");
67             for (j = 0; j < MAX_COLS; j++) {
68                 a[i*MAX_COLS + j] = (float) (i*10 + j);
69                 //printf(" %g ", a[i*MAX_COLS + j]);
70             }
71         }
72
73         printf("\n");
74
75         for (i = 0; i < MAX_ROWS; i++) {
76             //printf("\n");
77             for (j = 0; j < MAX_COLS; j++) {
78                 b[i*MAX_COLS + j] = (float) (i*10 + j);
79                 //printf(" %g ", b[i*MAX_COLS + j]);
80             }
81         }
82

```

```

83 MPI_Win_create(a,MAX_A_ENTRIES * sizeof(float),sizeof(float),MPI_INFO_NULL,MPI_COMM_WORLD,&winA)←
84 MPI_Win_create(b,MAX_B_ENTRIES * sizeof(float),sizeof(float),MPI_INFO_NULL,MPI_COMM_WORLD,&winB)←
85 MPI_Win_create(c,MAX_C_ENTRIES * sizeof(float),sizeof(float),MPI_INFO_NULL,MPI_COMM_WORLD,&winC)←
86
87 MPI_Win_fence(0,winA);
88 MPI_Win_fence(0,winB);
89 MPI_Win_fence(0,winC);
90
91 // Total Start
92 t_start = MPI_Wtime();
93
94 // Master compute its part of matrix
95 prows = nrows * (numprocs - 1);
96 row = arows - prows;
97 if(row > 0) {
98     for(k = 0; k < row; k++) {
99         for(i = 0; i < bcols; i++) {
100             c[(prows * ccols) + (ccols * k) + i] = 0.0;
101             for(j = 0; j < acols; j++) {
102                 c[(prows * ccols) + (ccols * k) + i] += a[(prows * acols) + (k * acols) + j] * b[(j * bcols)←
103             }
104         }
105     }
106 }
107
108 MPI_Win_fence(0,winA);
109 MPI_Win_fence(0,winB);
110 MPI_Win_fence(0,winC);
111
112 // Total End
113 t_end = MPI_Wtime();
114
115 fprintf(times,"\\n%d %lg",numprocs,t_end - t_start);
116
117 fclose(times);
118
119 /*
120 printf("\\nProcess %d # Processes %d\\n",myid,numprocs);
121 printf("\\n\\n");
122 for (i = 0; i < MAXCROWS; i++) {
123     printf("\\n");
124     for (j = 0; j < MAXCCOLS; j++) {
125         printf("%g\\t", c[(i * MAXCCOLS) + j]);
126     }
127 }
128 */
129 printf("\\n\\n");
130 printf("\\n\\nDone! %e Time %g\\n\\n",c[MAX_C_ENTRIES - 1],t_end - t_start);
131
132 }
133 else {
134
135 MPI_Win_create(a,MAX_A_ENTRIES * sizeof(float),sizeof(float),MPI_INFO_NULL,MPI_COMM_WORLD,&winA)←
136 MPI_Win_create(b,MAX_B_ENTRIES * sizeof(float),sizeof(float),MPI_INFO_NULL,MPI_COMM_WORLD,&winB)←
137 MPI_Win_create(c,MAX_C_ENTRIES * sizeof(float),sizeof(float),MPI_INFO_NULL,MPI_COMM_WORLD,&winC)←
138 MPI_Win_fence(0,winA);
139 MPI_Win_fence(0,winB);
140 MPI_Win_fence(0,winC);
141
142 MPI_Get(a, nrows * acols, MPI_FLOAT, 0, (myid - 1) * acols * nrows, nrows * acols, MPI_FLOAT, ←
winA);
143 MPI_Get(b, MAX_B_ENTRIES, MPI_FLOAT, 0, 0, MAX_B_ENTRIES, MPI_FLOAT, winB);
144
145 MPI_Win_fence(0,winA);
146 MPI_Win_fence(0,winB);
147
148 for(k = 0; k < nrows; k++) {
149     for(i = 0; i < bcols; i++) {
150         c[i + (k * bcols)] = 0.0;
151         for(j = 0; j < acols; j++) {
152             c[i + (k * bcols)] += a[j + (k * acols)] * b[j * bcols + i];

```

```

153     }
154   }
155 }
156 /*
157 printf("Processo %d # Processes\n\n",myid;numprocs);
158 printf("\n\n");
159 for (i = 0; i < MAXCROWS; i++) {
160   printf("\n");
161   for (j = 0; j < MAXCCOLS; j++) {
162     printf("%g\t", c[(i * MAXCCOLS) + j]);
163   }
164 }
165 */
166
167 MPI_Put(c, ccols * nrows, MPI_FLOAT, 0, (myid - 1) * ccols * nrows, ccols * nrows, MPI_FLOAT, &
winC);
168
169 MPI_Win_fence(0,winC);
170 }
171 MPI_Finalize();
172 }

```