Uppsala University


Parallel and Distributed Programming

1TD070

# Project - report

*Authors:*
Csongor Horváth

May 15, 2023

# The problem setting

The problem setting was to create a parallel implementation of the well known matrix multiplication algorithm. Previously we studied the possibilities of serial optimization of the problem in our High Performance Computing course.
The problem what we needed to solve has the following form: given two square dense matrices, $A = \{a_{i,j}\}$ and $B = \{b_{i,j}\}$, $A, B \in \mathbb{R},\ 1, 2, \ldots, n$. Let $C = AB$. It is known, $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}, \quad i, j = 1, 2, , n$.
Now in this given problem setting the task was to design a matrix-multiplication procedure for performing this task in a distributed memory parallel computer environment. Then the implementation of the procedure was made in C using MPI. Finally the task also included the design and performance of a performance evaluation of the final implementation.

It was also given that both matrix should be distributed, therefore there is a need for constant communication between the different processes. The goal in my implementation was the following. If given $p$ processes, then each of them should store and use the $\frac{1}{p}$th part of the $A$ and $B$ matrices in each computation time. I also wanted an implementation, where the local $C$ matrix partitions are disjoint. This wasn't required, but it seems to be a nice motivation.
In the implementation the measured time not include the reading and distribution of data which is done by only one *root* process. It is also not include the final communication which is the gathering the final $C$ values in the root process and writing it out to file. Note that it would be also possible and better in storage usage to read the starting data from file in each process and only store the locally necessary values in each process. Also it would be possible to do the writing without the gathering of the data with the usage of some communication. These options wasn't included in the project as it is out of the project scope.

Now a brief introduction about my project. I did two different implementation, one using a simple combinations of $1D$-type partitioning and another using the $2D$-type partitioning implementing Canon's algorithm. In both case I tried to make the serial code as fast as I could using several steps of optimisation as I will later discuss it. The advantage and disadvantages of both implementation will be discussed in the following sections.

# Partitioning strategies

I choose to make two different implementation of the problem using two different partitioning of the matrices.

For the $1D$-type partitioning I used row-wise partitioning in $A$ and $C$ and column-wise partitioning in $B$. Note that for the possibility if partitioning if we use $p$ processes and the original matrix size of $A$ and $B$ is $n \times n$, then $m = \frac{n}{p}$ and it is assumed that $p$ is divisor of $n$. So the given row-wise partitions are $A^{(s)}$ and $C^{(s)}$ with $n \times m$ size of local block in processor $P_s,\ s = 0, 1, \ldots, p-1$. Now if I use the same notation as in the description so the data distribution imposes additional structure in each block-column $C_i^{(s)},\ i = 0, 1, \ldots, p-1$, each of size $m \times m$. Now there is a given structure in the column-wise partition of $B$, so $B^{(j)}$ is $m \times n$ size local block for $j = 0, 1, \ldots, p-1$.

Now when in a given processor $A^{(s)}$ and $B^{(j)}$ is stored, then we can calculate the values of $C_j^{(s)}$ using $(C_j^{(s)})_{k,l} = \sum_{i=0}^{n} (A^{(s)})_{k,i} \cdot (B^{(j)})_{i,l}$, where $(M)_{i,j}$ is denote the $M$ matrix $i$th row, $j$th column.
So for all the calculation the following is necessary: at some time there should be $A^{(s)}, C^{(s)}$ and $B^{(j)}$ for $s, j = 0, 1, \ldots p-1$. This can be done by storing $A^{(s)}, C^{(s)}$ in process $P_s$ and start with storing $B^{(s)}$ here. Then calculate $C_s^{(s)}$ and for $p-1$ iteration move the local $B^s$ to the next process. So in the next step in $P_s$ there will be the $B^{(s-1)\%p}$ matrix. And we can calculate $C_{(s-1)\%p}^{(s)}$. Here $a\%b$ denote $a$ modulo $b$. We continue this for the remaining iterations. Not including the start and final communication this method communicates $(p-1) \cdot p \cdot m \cdot n = (p-1)n^2$ numbers (double).

For the $2D$-type partitioning I used Cannon's algorithm. Note that for my implementation a necessary thing is that $p$ is a square number and that $n$ is dividable by $\sqrt{p}$. So $m = \frac{n}{\sqrt{p}}$ and we will denote the processes as $P_{i,j},\ i, j = 0, \ldots \sqrt{p} - 1$. Now in $P_{i,j}$ we will calculate a $m \times m$ part of $C$, which is denoted by $C^{(i,j)}$ and if we split $C$ by $m$ rows and columns, then $C^{(i,j)}$ is the in the $i$th row and $j$th columns of the matrix generated by the $m \times m$ sub matrices.
As $A$ and $B$ are square matrix so the easiest way to split $A$ and $B$ the same way as $C$. So there are $A^{(i,j)}$ and $B^{(i,j)}$ sub matrices for $i, j = 0, 1 \ldots \sqrt{p} - 1$. For calculating $C^{(i,j)}$, we need the following pairs $A^{(i,k)}, B^{(k,j)}$ for $k =$

$0, 1, \ldots \sqrt{p} - 1$. As it is given in the Cannon's algorithms there is a given starting distribution using which in each process has the correct pairs occurs if we iterate as follow: after adding the correct calculations result to $C^{(i,j)}$ we move the local $A$ left and $B$ up in the $2D$ cyclic grid. The details of this can be found in my code or under Cannon's algorithm in the internet.
Now the used number of number communication in this case is the following: $2(\sqrt{p} - 1) \cdot p \cdot m \cdot m = 2(\sqrt{p} - 1)n^2$.

Reasoning for the algorithms. I wanted to make two implementation to be able to compare them. The Cannon's algorithm was chosen due to it's efficiency, but it has several problem, like the fact that the number of processors should be a square number so it is hard to make scaling experiment for it as the resources are limited. The other was chosen due to it's simplicity and easy implementation purposes and the lack of extra criteria.

# Implementation

As previously described I made 2 separate implementation and I tried to optimize both version. The two implementation has a similar framework. The difference is only at the measured place and the matrix distribution beforehand the other codes are the same, so it can be discussed together.

There are a read and write function which is called only from the root process and the read fills up the $A$ and $B$ matrices and the write is saving to file the final $C$ matrix. Additionally there is a print matrix function for testing and also a fill matrix which can generate random $A$ and $B$ to start with with fixed size if the `FROM_FILE` variable is set to false. In the main I read the arguments and declare the variables first. After it from the root I read (or generate) the matrices and share the $n$ value and distribute the matrices to their staring process.
Then I start the timer and for the necessary number of times I calculate a local $C$ partition and send the data to the next process. As this is the measured part I tried to speed up is with several trick. First I made sure the order of the loops in the calculation of $C$ values are optimal. Note that it is the most important part as accidentally I first used a non optimal ordering for the 1D-type slicing case and after correcting it I saw a more than 10 times speed up. Another trick is that for the indexes I can calculate the values in the outer loops. This is more important in the 1D case, where originally there is a an expression with modulo from the most outer loop, and by moving it's calculation outside I reached a further 25% speed up. Finally the smallest and most controversial thing in my mind is the speed up from changing the communication. In my original implementation I used `MPI_Sendrecv_replace` to move the data. This can only be called after the calculations are done. I tried to speed up this by using non blocking send and receive calls before calculation and adding wait after the calculation.
This way in the background the communication can happen while the processes works on the calculations and not loosing time with it. Note that running these in the background can also use resources and for this implementation a storage area shall be allocated in the process to store the received data while the original data is used to calculate the $C$ values. Since I don't know how the `MPI_Sendrecv_replace` is implemented it is possible that with the usage of the extra buffer allocation I use more memory this way. On the other hand I will include the time measurements with this version of com-

munications as well, and as we will see we doesn't gain much performance this way. So I will run the strong scaling test for 4 binary file: `matmul, matmul2, canon, canon2`. Here the version with 2 in the end are the one using the above described background send version. For reference I use my original implementation using row-wise and column-wise splitting of the matrices with the name `matmul`. And my implementation of Canon's algorithm is in the binary called `canon`.
After the timed section I collect the run times same as in the previous project and I also collect the final $C$ matrix in the root process and write run time to console from there and write the result to file.

Now about the distribution of data. I used a self defined vector type and as scatter would continue from the end of the previous vector type and for me I need to start from the first non sent element, so other then the row communication I use Send and Receive function to distribute the starting data. This way playing with the indexes I can send the data to the correct starting process even in the Cannon's algorithm case. As in the other implementation $C$ stored row-wise it can be collect with the Gather function. But in the Cannon's algorithm case there also needed a more complicated implementation with send and receive using the self defined vector type.

This is a brief description of the implementation and how they will bee referenced later. My submission includes the original 1D based implementation with the usage of `MPI_Sendrecv_replace` communication, but the other codes can be found here in the Appendix.

# Performance experiment

## Strong scaling

For the strong scaling experiment I made timings with all four version. I used the *input3600.txt* input file for time measurements and changed the number of cores used. The run times can be seen in Table 1. As it can be seen the Canon's algorithm is slightly faster for the cases using less core then what we have in resources in the running of the timings. Also it can be seen that the change in communication result in a very slight improvement in the performance.
On the other hand all 4 version has a very similar performance even if the difference can be clearly seen in the run time, so my submitted version (`matmul`) is not significantly worse then the other version created for testing references.

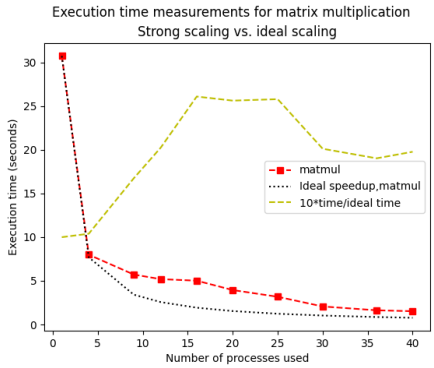
Also note that in the application of this codes the distribution and collection of data is also important, so I made time measurements with that included and even though the Cannon's algorithm has a bit more complicated structure for this the run times compared to each other didn't change. So thees run times are relevant for comparison considering real application. And thus my conclusion is that these are eqvivalently good for real application.

| | $-n$ 1 | $-n$ 4 | $-n$ 9 | $-n$ 12 | $-n$ 16 | $-n$ 20 | $-n$ 25 | $-n$ 30 | $-n$ 36 | $-n$ 40 |
|---|---|---|---|---|---|---|---|---|---|---|
| matmul | 30.7539 | 7.9928 | 5.7216 | 5.1917 | 5.0179 | 3.9425 | 3.1745 | 2.0632 | 1.6259 | 1.5210 |
| matmul2 | 30.7312 | 7.8869 | 5.6193 | 5.0797 | 4.9034 | 3.8564 | 3.1053 | 2.1630 | 1.5027 | 1.4058 |
| canon | 29.8793 | 8.1596 | 5.3015 | | 4.7498 | | 2.6882 | | 2.0104 | |
| canon2 | 29.8679 | 8.0281 | 5.1428 | | 4.6168 | | 2.7121 | | 1.7681 | |

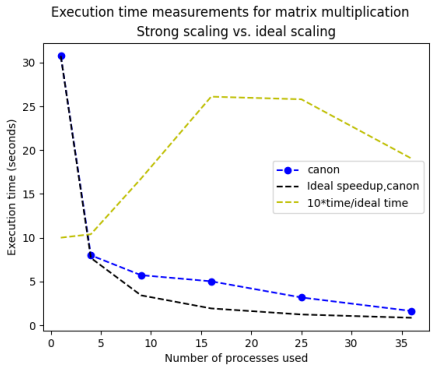
Table 1: Time measurement for strong scaling (*input3600.txt*) in 2 node 32 core

Now I want to graphically show the scaling result. As the 3 run times are very similar I will make plot only using the `matmul` and `canon` results and show them in 2 different plot.
[1]~~As in Figure~~ 1 ~~we can see the measured scaling and the ideal speed up are~~

[1]Updating the old report's conclusion for scaling




(a) matmul run times




(b) canon run times

Figure 1: Weak scaling

~~close to each other. The ideal speed up is the run time on 1 core divided by the used number of processor. In the begining using 4 core we get a perfect speed up, then the difference is larger from the ideal speed up in the next some cases and then using more core and getting closer to the limit of resources the measured time is getting very close once again to the ideal speed up. As in the plots we can see the ratio between the measured time and the ideal speed up goes up to 2.5 and then start getting smaller. As we reached the and of the resources it starts getting worse again as in Figure~~ 1 ~~we can see it.~~

~~Overall considering the task I would say even if the speed up is far from perfect it is still a reasonably good speed up for the given task.~~

We can see in Figure 1 the strong scaling of the code. As we can see in both case between 4 and 16 core usage the ratio between the ideal speedup and the measured run time are getting significantly slower, but later this ratio doesn't continue to increase. To investigate the reasons behind this I run map analysis on the `matmul` code using 4 and 16 core on the 3600 input file, to see what change. In the case of 4 process, the communication takes up around 7% of the overall CPU usage and around 93% comes from the calculation line, and 70% of this is spent on memory access and only the rest for the real calculation. This means that the most crucial part of the code is the memory access and the communication is not too important for small number of process. Now for 16 process I got the followings: communication 21%, the

line of calculation is 78% and 74% of this comes from communication.
From this using the knowledge that between 4 and 16 process the ratio with the ideal speed up grow to twice the size we can say that both the calculation process use significantly more resources as the ideal speed up would predict and also the communication is using significantly more resource. So the conclusion is that the most significant part is the slow memory access and it is partly resulting the imperfect scaling, and also the significant grow after a certain amount of process reached in the communication contribute to the imperfect scaling.

## Weak scaling experiment

| #iter / size | $-n$ 1 / 3600 | $-n$ 4 / 5716 | $-n$ 9 / 7488 | $-n$ 16 / 9072 | $-n$ 25 / 10525 |
|---|---|---|---|---|---|
| matmul | 30.7606 | 33.5240 | 47.4283 | 75.4612 | 76.1026 |
| canon | 29.7804 | 31.9980 | 48.4836 | 77.3664 | 77.3933 |

Table 2: Time measurement for weak scaling in 2 node 32 core

For the weak scaling I used the different size of input data. The weak scaling experiment was run for both the original `matmul` and the `canon` binary file. I used the files in the given directory. There the input sizes were given in a way that for weak scaling I should use the square numbers. So the weak scaling can be done for both implementation. The run times can be seen in Table 2.
The pair of processor number input file size can be found also in this table. Surprisingly here for larger number of processes the cannon is slower.

Now I plot the weak scaling wiht ideal speed up in the Figure 2 only using the measurements for `matmul`.
As in the figure we see the weak scaling for 4 process is almost perfect. Then as the number of process grow the scaling is getting worse. But the ratio between the ideal speed up and the measured time is very similar as it was in the previous cases. And the reason behind, while the scaling is getting worse for a while is the same as for the strong scaling.

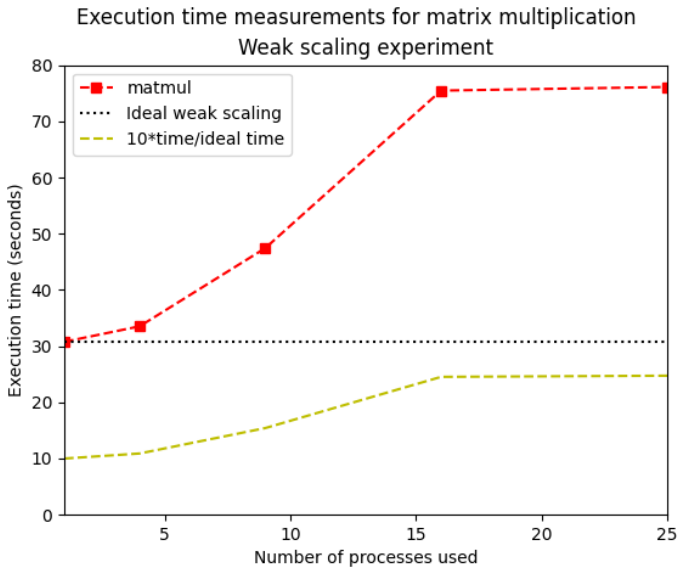



Figure 2: Weak scaling for matmul

# Summary

So the conclusion looking at the scaling and the ratio between the measured and ideal run time are seemingly very similar for both weak and strong scaling. So in the beginning the scaling is good and then the ratio grows up to 2.5 and finally it is falling back to around being around 2. This means that for small cases ($n = 4$) the scaling is very good and for larger number of cores it is getting worse for a while, but after the ratio stops getting worse and becomes mostly stable.

Overall the scaling results significant speed up in all cases and for application in large scale matrix multiplication it is very useful. For reference I made a time measurement using the largest file with $n = 18000$ using a single core and using 30 core (the limit of resource is 2 node = 32 core), so the largest possible number of cores. Here I get run times: 313.8131s for 30 core and 4937.2002s for 1 core. So there is a clear advantage of the parallel implementation.

So I would recommend the usage of parallelization in the case of an applica-

tion using large scale matrix multiplication or when large number of middle size matrix multiplication is necessary. In this cases the multiple cores can make a large overall impact in the final run time.

# Appendix

## matmul

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define root 0
#define FROM_FILE 1

int read_input(const char *file_name, double **matrix1, double **matrix2) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "r"))) {
                perror("Couldn't open input file");
                return -1;
        }
        int n_val;
        if (EOF == fscanf(file, "%d", &n_val)) {
                perror("Couldn't read element count from input file");
                return -1;
        }
        if (NULL == (*matrix1 = malloc(n_val * n_val * sizeof(double)))) {
                perror("Couldn't allocate memory for input");
                return -1;
        }
        if (NULL == (*matrix2 = malloc(n_val * n_val * sizeof(double)))) {
                perror("Couldn't allocate memory for input");
                return -1;
        }
        for (int i=0; i<n_val*n_val; i++) {
                if (EOF == fscanf(file, "%lf", &((*matrix1)[i]))) {
                        perror("Couldn't read elements from input file");
                        return -1;
                }
        }
        for (int i=0; i<n_val*n_val; i++) {
                if (EOF == fscanf(file, "%lf", &((*matrix2)[i]))) {
                        perror("Couldn't read elements from input file");
                        return -1;
                }
        }
        if (0 != fclose(file)){
                perror("Warning: couldn't close input file");
        }
        return n_val;
}

int write_output(char *file_name, const double *matrix, int n_val) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "w"))) {
                perror("Couldn't open output file");
                return -1;
        }
        for (int i = 0; i < n_val*n_val; i++) {
                if (0 > fprintf(file, "%.6f ", matrix[i])) {
                        perror("Couldn't write to output file");
                }
        }
        if (0 != fclose(file)) {
                perror("Warning: couldn't close output file");
        }
        return 0;
}

void print_local(double* matrix, int n, int k, int rank){
    sleep(rank);
    printf("From process: %d\n", rank);
    for (size_t i = 0; i < k; i++)
    {
        for (size_t j = 0; j < n; j++)
        {
```

```
            printf("%lf ",matrix[i*n+j]);
        }
        printf("\n");

    }
}
void fill_matrix(double* matrix, int length){
    srand(clock());
    for (size_t i = 0; i < length; i++)
    {
        matrix[i]=(rand() % 10000)/100000.0;
    }
}

int main(int argc, char **argv){
    // Arguments
    if (3 != argc) {
                printf("Usage: ./matmul input_file output_file\n");
                return 1;
        }
        char *input_name = argv[1];
        char *output_name = argv[2];

        // define variables
    int left, right;
    int size, rank;
    int n, m;
    int id0, id1, id2;
    double s_time, loc_time, glob_time;
    double* A;
    double* B;
    double* C;
    double* A_rows;
    double* B_cols;
    double* C_rows;

    MPI_Datatype col_type;

    //MPI init
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    right = rank + 1;
        if (right == size) right = 0;

        left = rank - 1;
        if (left == -1) left = size-1;

    if(rank==root){
    // Read input file
        #if FROM_FILE
    if (0 > (n = read_input(input_name, &A, &B))) {
                return 2;
        }
    #else
    n=1000;
    A=(double*)malloc(n*n*sizeof(double));
    B=(double*)malloc(n*n*sizeof(double));
    fill_matrix(A, n*n);
    fill_matrix(B, n*n);
    #endif
    }

        //Broadcast n & init varibles
    MPI_Bcast(&n, 1, MPI_INT, root, MPI_COMM_WORLD);

    m=n/size;
    A_rows=(double*)malloc(m*n*sizeof(double));
    B_cols=(double*)calloc(m*n, sizeof(double));
    C_rows=(double* )calloc(m*n, sizeof(double));


        // Distributing the input data by row & col
    MPI_Type_vector(n, m, n, MPI_DOUBLE, &col_type);
    MPI_Type_commit(&col_type);
```

```
    MPI_Scatter(A, m*n, MPI_DOUBLE, A_rows, m*n, MPI_DOUBLE, root, MPI_COMM_WORLD);

    if(rank==root){
        for (size_t i = 1; i < size; i++)
        {
            MPI_Send(&B[i*m], 1, col_type, i, i, MPI_COMM_WORLD);
        }
        // using sendrecv to avoid deadlock when sending data to itself
        MPI_Sendrecv(B, 1, col_type, root, root, B_cols, m*n,  MPI_DOUBLE, root, root,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else{
        MPI_Recv(B_cols, m*n, MPI_DOUBLE, root, rank, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
            ;
    }

    // As A,B no more needed we can free the memory for them in the root
    if(rank==root){
        free(A);
        free(B);
    }


s_time = MPI_Wtime();

// Calculate C vals
for (size_t l = 0; l < size; l++)
{
    id0=(rank+l)%size*m;
    for (size_t i = 0; i < m; i++)
    {
        id1=i*n;
        for (size_t j = 0; j < n; j++)
        {
            id2=j*m;
            for(size_t k = 0; k < m; k++)
            {
                C_rows[k+id0+id1]+=A_rows[j+id1]*B_cols[id2+k];
            }
        }
    }
    // Change col blocks
    if(l!=l-1)
    MPI_Sendrecv_replace(B_cols, m*n, MPI_DOUBLE, left, 0, right, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
}

    // Gather timer data and C
    loc_time = MPI_Wtime() - s_time;

    MPI_Reduce(&loc_time, &glob_time, 1, MPI_DOUBLE, MPI_MAX, root, MPI_COMM_WORLD);
    if(rank==root)
    C=(double*)malloc(n*n*sizeof(double));
    MPI_Gather(C_rows, n*m, MPI_DOUBLE, C, n*m, MPI_DOUBLE, root, MPI_COMM_WORLD);

    if(rank==root){
        printf("%.4lf\n",glob_time);
        write_output(output_name, C, n);
        free(C);
    }

    free(A_rows);
    free(B_cols);
    free(C_rows);
    MPI_Finalize();
}
```

# Makefile

```
##############################################################################
# Makefile for assignment 2, Parallel and Distributed Computing 2023.
##############################################################################

CC = mpicc
CFLAGS = -std=c99 -g -O3
LIBS = -lm

BIN = matmul

all: $(BIN)

stencil: matmul.c
	$(CC) $(CFLAGS) -o $@ $< $(LIBS)

clean:
	$(RM) $(BIN)
```

# canon

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <math.h>
#include <mpi.h>
#define root 0
#define FROM_FILE 1

int read_input(const char *file_name, double **matrix1, double **matrix2) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "r"))) {
                perror("Couldn't open input file");
                return -1;
        }
        int n_val;
        if (EOF == fscanf(file, "%d", &n_val)) {
                perror("Couldn't read element count from input file");
                return -1;
        }
        if (NULL == (*matrix1 = malloc(n_val * n_val * sizeof(double)))) {
                perror("Couldn't allocate memory for input");
                return -1;
        }
        if (NULL == (*matrix2 = malloc(n_val * n_val * sizeof(double)))) {
                perror("Couldn't allocate memory for input");
                return -1;
        }
        for (int i=0; i<n_val*n_val; i++) {
                if (EOF == fscanf(file, "%lf", &((*matrix1)[i]))) {
                        perror("Couldn't read elements from input file");
                        return -1;
                }
        }
        for (int i=0; i<n_val*n_val; i++) {
                if (EOF == fscanf(file, "%lf", &((*matrix2)[i]))) {
                        perror("Couldn't read elements from input file");
                        return -1;
                }
        }
        if (0 != fclose(file)){
                perror("Warning: couldn't close input file");
        }
        return n_val;
}
int write_output(char *file_name, const double *matrix, int n_val) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "w"))) {
                perror("Couldn't open output file");
                return -1;
        }
        for (int i = 0; i < n_val*n_val; i++) {
                if (0 > fprintf(file, "%.6f ", matrix[i])) {
                        perror("Couldn't write to output file");
                }
        }
        if (0 != fclose(file)) {
                perror("Warning: couldn't close output file");
        }
        return 0;
}
void print_local(double* matrix, int n, int k, int rank){
    sleep(rank);
    printf("From process: %d\n", rank);
    for (size_t i = 0; i < k; i++)
    {
        for (size_t j = 0; j < n; j++)
        {
            printf("%lf ",matrix[i*n+j]);
        }
        printf("\n");

    }
}
```

```
void fill_matrix(double* matrix, int length){
    srand(clock());
    for (size_t i = 0; i < length; i++)
    {
        matrix[i]=(rand() % 10000)/100000.0;
    }
}

int main(int argc, char *argv[])
{
     // Arguments
    if (3 != argc) {
                printf("Usage: ./matmul input_file output_file\n");
                return 1;
        }
        char *input_name = argv[1];
        char *output_name = argv[2];

        // define variable
    int size, rank;
    int n, N_loc;
    int id0, id1;
    double s_time, loc_time, glob_time;
    double* A;
    double* B;
    double* C;
    double* A_loc;
    double* B_loc;
    double* C_loc;

    int source, up, right, down, left;
    int ndims, dims[2], periods[2], reorder;

    MPI_Datatype col_type;
    MPI_Comm comm2D;
    MPI_Status status;
    //MPI init
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank==root){
    // Read input file
        #if FROM_FILE
    if (0 > (n = read_input(input_name, &A, &B))) {
                return 2;
        }
    #else
    n=1000;
    A=(double*)malloc(n*n*sizeof(double));
    B=(double*)malloc(n*n*sizeof(double));

    fill_matrix(A, n*n);
    fill_matrix(B, n*n);
    #endif
    }


        //Broadcast n & init varibles
    MPI_Bcast(&n, 1, MPI_INT, root, MPI_COMM_WORLD);
    dims[0] = sqrt(size); dims[1] = sqrt(size);
    if(rank==root && (dims[0]*dims[1]!=size || n%dims[0]!=0)){
        printf("Number of process should be a square number and it should divide n!!!\n"
            );
    }
    periods[0]=1; periods[1]=1;
    reorder=0, ndims=2;
    MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,periods,reorder,&comm2D);
    MPI_Cart_shift(comm2D,1,1,&left,&right);
    MPI_Cart_shift(comm2D,0,1,&up,&down);
    N_loc=n/dims[0];
    A_loc=(double*)malloc(N_loc*N_loc*sizeof(double));
    B_loc=(double*)malloc(N_loc*N_loc* sizeof(double));
    C_loc=(double* )calloc(N_loc*N_loc, sizeof(double));
```

```
    // Distributing the input data for Cannon's alg starting distribution
MPI_Type_vector(N_loc, N_loc, n, MPI_DOUBLE, &col_type);
MPI_Type_commit(&col_type);

if(rank==root){
    for (int i = 0; i < dims[0]; i++){
        for (int j = 0; j < dims[0]; j++)
        {
            if(i!=0 || j!=0){
                MPI_Send(&A[i*(N_loc*n)+(((j+i)%dims[0])*N_loc)], 1, col_type, i*
                    dims[0]+j, i*dims[0]+j, MPI_COMM_WORLD);
                MPI_Send(&B[((j+i)%dims[0])*(N_loc*n)+j*N_loc], 1, col_type, i*dims
                    [0]+j, i*dims[0]+j, MPI_COMM_WORLD);
            }
        }
    }
    // using sendrecv to avoid deadlock when sending data to itself
    MPI_Sendrecv(A, 1, col_type, root, root, A_loc, N_loc*N_loc, MPI_DOUBLE, root,
        root, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(B, 1, col_type, root, root, B_loc, N_loc*N_loc, MPI_DOUBLE, root,
        root, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else{
    MPI_Recv(A_loc, N_loc*N_loc, MPI_DOUBLE, root, rank, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    MPI_Recv(B_loc, N_loc*N_loc, MPI_DOUBLE, root, rank, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
}

// As A,B no more needed we can free the memory for them in the root
if(rank==root){
    free(A);
    free(B);
}


s_time = MPI_Wtime();

for(int shift=0;shift<dims[0];shift++) {
// Matrix multiplication
for(int i=0;i<N_loc;i++){
    id0=i*N_loc;
    for(int k=0;k<N_loc;k++){
        id1=k*N_loc;
        for(int j=0;j<N_loc;j++)
            C_loc[id0+j]+=A_loc[id0+k]*B_loc[id1+j];
    }
}

if(shift==dims[0]-1) break;
// Communication
MPI_Sendrecv_replace(A_loc, N_loc*N_loc, MPI_DOUBLE, left, 0, right, 0, comm2D,
    MPI_STATUS_IGNORE);
MPI_Sendrecv_replace(B_loc, N_loc*N_loc, MPI_DOUBLE, up, 0, down, 0, comm2D,
    MPI_STATUS_IGNORE);
}

// Gather timer data and C
loc_time = MPI_Wtime() - s_time;
MPI_Reduce(&loc_time, &glob_time, 1, MPI_DOUBLE, MPI_MAX, root, MPI_COMM_WORLD);



if(rank==root){
    C=(double*)malloc(n*n*sizeof(double));
    for (size_t i = 0; i < dims[0]; i++){
        for (size_t j = 0; j < dims[0]; j++)
        {
            if(i!=0 || j!=0){
                MPI_Recv(&C[i*(N_loc*n)+(j*N_loc)], 1, col_type, i*dims[0]+j, i*dims
                    [0]+j, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
        }
    }
    MPI_Sendrecv(C_loc, N_loc*N_loc, MPI_DOUBLE, root, root, C, 1, col_type, root,
        root, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

```
    else{
        MPI_Send(C_loc, N_loc*N_loc, MPI_DOUBLE, root, rank, MPI_COMM_WORLD);
    }

 if(rank==root){
        printf("%.4lf\n",glob_time);
        write_output(output_name, C, n);
        free(C);
    }

    free(A_loc);
    free(B_loc);
    free(C_loc);
    MPI_Comm_free( &comm2D );
    MPI_Finalize();
    return 0;
}
```

# matmul2

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define root 0
#define FROM_FILE 1

int read_input(const char *file_name, double **matrix1, double **matrix2) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "r"))) {
                perror("Couldn't open input file");
                return -1;
        }
        int n_val;
        if (EOF == fscanf(file, "%d", &n_val)) {
                perror("Couldn't read element count from input file");
                return -1;
        }
        if (NULL == (*matrix1 = malloc(n_val * n_val * sizeof(double)))) {
                perror("Couldn't allocate memory for input");
                return -1;
        }
        if (NULL == (*matrix2 = malloc(n_val * n_val * sizeof(double)))) {
                perror("Couldn't allocate memory for input");
                return -1;
        }
        for (int i=0; i<n_val*n_val; i++) {
                if (EOF == fscanf(file, "%lf", &((*matrix1)[i]))) {
                        perror("Couldn't read elements from input file");
                        return -1;
                }
        }
        for (int i=0; i<n_val*n_val; i++) {
                if (EOF == fscanf(file, "%lf", &((*matrix2)[i]))) {
                        perror("Couldn't read elements from input file");
                        return -1;
                }
        }
        if (0 != fclose(file)){
                perror("Warning: couldn't close input file");
        }
        return n_val;
}

int write_output(char *file_name, const double *matrix, int n_val) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "w"))) {
                perror("Couldn't open output file");
                return -1;
        }
        for (int i = 0; i < n_val*n_val; i++) {
                if (0 > fprintf(file, "%.6f ", matrix[i])) {
                        perror("Couldn't write to output file");
```

```
                }
        }
        if (0 != fclose(file)) {
                perror("Warning: couldn't close output file");
        }
        return 0;
}

void print_local(double* matrix, int n, int k, int rank){
    sleep(rank);
    printf("From process: %d\n", rank);
    for (size_t i = 0; i < k; i++)
    {
        for (size_t j = 0; j < n; j++)
        {
            printf("%lf ",matrix[i*n+j]);
        }
        printf("\n");

    }
}
void fill_matrix(double* matrix, int length){
    srand(clock());
    for (size_t i = 0; i < length; i++)
    {
        matrix[i]=(rand() % 10000)/100000.0;
    }
}

int main(int argc, char **argv){
    // Arguments
    if (3 != argc) {
                printf("Usage: ./matmul input_file output_file\n");
                return 1;
        }
        char *input_name = argv[1];
        char *output_name = argv[2];

        // define variables
    int left, right;
    int size, rank;
    int n, m;
    int id0, id1, id2;
    double s_time, loc_time, glob_time;
    double* A;
    double* B;
    double* buffer;
    double* temp;
    double* C;
    double* A_rows;
    double* B_cols;
    double* C_rows;

    MPI_Datatype col_type;
    MPI_Request r_send;
    MPI_Request r_recv;

    //MPI init
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    right = rank + 1;
        if (right == size) right = 0;

        left = rank - 1;
        if (left == -1) left = size-1;

    if(rank==root){
    // Read input file
        #if FROM_FILE
    if (0 > (n = read_input(input_name, &A, &B))) {
                return 2;
        }
    #else
    n=1000;
```

```
    A=(double*)malloc(n*n*sizeof(double));
    B=(double*)malloc(n*n*sizeof(double));
    fill_matrix(A, n*n);
    fill_matrix(B, n*n);
    #endif
    }

        //Broadcast n & init varibles
    MPI_Bcast(&n, 1, MPI_INT, root, MPI_COMM_WORLD);

    m=n/size;
    A_rows=(double*)malloc(m*n*sizeof(double));
    B_cols=(double*)calloc(m*n, sizeof(double));
    buffer=(double*)calloc(m*n, sizeof(double));
    C_rows=(double* )calloc(m*n, sizeof(double));


        // Distributing the input data by row & col
    MPI_Type_vector(n, m, n, MPI_DOUBLE, &col_type);
    MPI_Type_commit(&col_type);

    MPI_Scatter(A, m*n, MPI_DOUBLE, A_rows, m*n, MPI_DOUBLE, root, MPI_COMM_WORLD);

    if(rank==root){
        for (size_t i = 1; i < size; i++)
        {
            MPI_Send(&B[i*m], 1, col_type, i, i, MPI_COMM_WORLD);
        }
        // using sendrecv to avoid deadlock when sending data to itself
        MPI_Sendrecv(B, 1, col_type, root, root, B_cols, m*n,  MPI_DOUBLE, root, root,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else{
        MPI_Recv(B_cols, m*n, MPI_DOUBLE, root, rank, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
            ;
    }

    // As A,B no more needed we can free the memory for them in the root
    if(rank==root){
        free(A);
        free(B);
    }


s_time = MPI_Wtime();

// Calculate C vals
for (size_t l = 0; l < size; l++)
{
    if(l!=l-1){
    MPI_Isend(B_cols, m*n, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, &r_send);
    MPI_Irecv(buffer, m*n, MPI_DOUBLE,right, 0, MPI_COMM_WORLD, &r_recv);
    }
    id0=(rank+l)%size*m;
    for (size_t i = 0; i < m; i++)
    {
        id1=i*n;
        for (size_t j = 0; j < n; j++)
        {
            id2=j*m;
            for(size_t k = 0; k < m; k++)
            {
                C_rows[k+id0+id1]+=A_rows[j+id1]*B_cols[id2+k];
            }
        }
    }
    // Change col blocks
    if(l==l-1) break;
    MPI_Wait(&r_send, MPI_STATUS_IGNORE);
    MPI_Wait(&r_recv, MPI_STATUS_IGNORE);
    temp=B_cols;
    B_cols=buffer;
    buffer=temp;
}

    // Gather timer data and C
    loc_time = MPI_Wtime() - s_time;
```

```
    MPI_Reduce(&loc_time, &glob_time, 1, MPI_DOUBLE, MPI_MAX, root, MPI_COMM_WORLD);
    if(rank==root)
    C=(double*)malloc(n*n*sizeof(double));
    MPI_Gather(C_rows, n*m, MPI_DOUBLE, C, n*m, MPI_DOUBLE, root, MPI_COMM_WORLD);

    if(rank==root){
        printf("%.4lf\n",glob_time);
        write_output(output_name, C, n);
        free(C);
    }

    free(A_rows);
    free(B_cols);
    free(C_rows);
    MPI_Finalize();
}
```

# canon2

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <math.h>
#include <mpi.h>
#define root 0
#define FROM_FILE 1

int read_input(const char *file_name, double **matrix1, double **matrix2) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "r"))) {
                perror("Couldn't open input file");
                return -1;
        }
        int n_val;
        if (EOF == fscanf(file, "%d", &n_val)) {
                perror("Couldn't read element count from input file");
                return -1;
        }
        if (NULL == (*matrix1 = malloc(n_val * n_val * sizeof(double)))) {
                perror("Couldn't allocate memory for input");
                return -1;
        }
        if (NULL == (*matrix2 = malloc(n_val * n_val * sizeof(double)))) {
                perror("Couldn't allocate memory for input");
                return -1;
        }
        for (int i=0; i<n_val*n_val; i++) {
                if (EOF == fscanf(file, "%lf", &((*matrix1)[i]))) {
                        perror("Couldn't read elements from input file");
                        return -1;
                }
        }
        for (int i=0; i<n_val*n_val; i++) {
                if (EOF == fscanf(file, "%lf", &((*matrix2)[i]))) {
                        perror("Couldn't read elements from input file");
                        return -1;
                }
        }
        if (0 != fclose(file)){
                perror("Warning: couldn't close input file");
        }
        return n_val;
}
int write_output(char *file_name, const double *matrix, int n_val) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "w"))) {
                perror("Couldn't open output file");
                return -1;
        }
        for (int i = 0; i < n_val*n_val; i++) {
                if (0 > fprintf(file, "%.6f ", matrix[i])) {
                        perror("Couldn't write to output file");
                }
        }
        if (0 != fclose(file)) {
                perror("Warning: couldn't close output file");
        }
        return 0;
}
void print_local(double* matrix, int n, int k, int rank){
    sleep(rank);
    printf("From process: %d\n", rank);
    for (size_t i = 0; i < k; i++)
    {
        for (size_t j = 0; j < n; j++)
        {
            printf("%lf ",matrix[i*n+j]);
        }
        printf("\n");

    }
}
```

```
void fill_matrix(double* matrix, int length){
    srand(clock());
    for (size_t i = 0; i < length; i++)
    {
        matrix[i]=(rand() % 10000)/100000.0;
    }
}

int main(int argc, char *argv[])
{
     // Arguments
    if (3 != argc) {
                printf("Usage: ./matmul input_file output_file\n");
                return 1;
        }
        char *input_name = argv[1];
        char *output_name = argv[2];

        // define variable
    int size, rank;
    int n, N_loc;
    int id0, id1;
    double s_time, loc_time, glob_time;
    double* A;
    double* buffer_A;
    double* temp_A;
    double* B;
    double* buffer_B;
    double* temp_B;
    double* C;
    double* A_loc;
    double* B_loc;
    double* C_loc;

    int source, up, right, down, left;
    int ndims, dims[2], periods[2], reorder;

    MPI_Request r_send_A;
    MPI_Request r_recv_A;
    MPI_Request r_send_B;
    MPI_Request r_recv_B;
    MPI_Datatype col_type;
    MPI_Comm comm2D;
    MPI_Status status;
    //MPI init
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank==root){
    // Read input file
        #if FROM_FILE
    if (0 > (n = read_input(input_name, &A, &B))) {
                return 2;
        }
    #else
    n=1000
    A=(double*)malloc(n*n*sizeof(double));
    B=(double*)malloc(n*n*sizeof(double));

    fill_matrix(A, n*n);
    fill_matrix(B, n*n);
    #endif
    }


        //Broadcast n & init varibles
    MPI_Bcast(&n, 1, MPI_INT, root, MPI_COMM_WORLD);
    dims[0] = sqrt(size); dims[1] = sqrt(size);
    if(rank==root && (dims[0]*dims[1]!=size || n%dims[0]!=0)){
        printf("Number of process should be a square number and it should divide n!!!\n"
            );
    }
    periods[0]=1; periods[1]=1;
    reorder=0, ndims=2;
    MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,periods,reorder,&comm2D);
```

```
MPI_Cart_shift(comm2D,1,1,&left,&right);
MPI_Cart_shift(comm2D,0,1,&up,&down);
N_loc=n/dims[0];
A_loc=(double*)malloc(N_loc*N_loc*sizeof(double));
B_loc=(double*)malloc(N_loc*N_loc* sizeof(double));
buffer_A=(double*)malloc(N_loc*N_loc*sizeof(double));
buffer_B=(double*)malloc(N_loc*N_loc* sizeof(double));
C_loc=(double* )calloc(N_loc*N_loc, sizeof(double));


    // Distributing the input data for Cannon's alg starting distribution
MPI_Type_vector(N_loc, N_loc, n, MPI_DOUBLE, &col_type);
MPI_Type_commit(&col_type);

if(rank==root){
    for (int i = 0; i < dims[0]; i++){
        for (int j = 0; j < dims[0]; j++)
        {
            if(i!=0 || j!=0){
                MPI_Send(&A[i*(N_loc*n)+(((j+i)%dims[0])*N_loc)], 1, col_type, i*
                    dims[0]+j, i*dims[0]+j, MPI_COMM_WORLD);
                MPI_Send(&B[((j+i)%dims[0])*(N_loc*n)+j*N_loc], 1, col_type, i*dims
                    [0]+j, i*dims[0]+j, MPI_COMM_WORLD);
            }
        }
    }
    // using sendrecv to avoid deadlock when sending data to itself
    MPI_Sendrecv(A, 1, col_type, root, root, A_loc, N_loc*N_loc, MPI_DOUBLE, root,
        root, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(B, 1, col_type, root, root, B_loc, N_loc*N_loc, MPI_DOUBLE, root,
        root, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else{
    MPI_Recv(A_loc, N_loc*N_loc, MPI_DOUBLE, root, rank, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    MPI_Recv(B_loc, N_loc*N_loc, MPI_DOUBLE, root, rank, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
}

// As A,B no more needed we can free the memory for them in the root
if(rank==root){
    free(A);
    free(B);
}



s_time = MPI_Wtime();

for(int shift=0;shift<dims[0];shift++) {
if(shift!=dims[0]-1){
MPI_Isend(A_loc, N_loc*N_loc, MPI_DOUBLE, left, 0, comm2D, &r_send_A);
MPI_Irecv(buffer_A, N_loc*N_loc, MPI_DOUBLE,right, 0, comm2D, &r_recv_A);
MPI_Isend(B_loc, N_loc*N_loc, MPI_DOUBLE, up, 0, comm2D, &r_send_B);
MPI_Irecv(buffer_B, N_loc*N_loc, MPI_DOUBLE,down, 0, comm2D, &r_recv_B);
}
// Matrix multiplication
for(int i=0;i<N_loc;i++){
    id0=i*N_loc;
    for(int k=0;k<N_loc;k++){
        id1=k*N_loc;
        for(int j=0;j<N_loc;j++)
            C_loc[id0+j]+=A_loc[id0+k]*B_loc[id1+j];
    }
}

if(shift==dims[0]-1) break;
// Communication
MPI_Wait(&r_send_A, MPI_STATUS_IGNORE);
MPI_Wait(&r_send_B, MPI_STATUS_IGNORE);
MPI_Wait(&r_recv_A, MPI_STATUS_IGNORE);
MPI_Wait(&r_recv_B, MPI_STATUS_IGNORE);
temp_A=A_loc;
A_loc=buffer_A;
buffer_A=temp_A;
temp_B=B_loc;
B_loc=buffer_B;
```

```
    buffer_B=temp_B;
    }

   // Gather timer data and C
    loc_time = MPI_Wtime() - s_time;
    MPI_Reduce(&loc_time, &glob_time, 1, MPI_DOUBLE, MPI_MAX, root, MPI_COMM_WORLD);



    if(rank==root){
        C=(double*)malloc(n*n*sizeof(double));
        for (size_t i = 0; i < dims[0]; i++){
            for (size_t j = 0; j < dims[0]; j++)
            {
                if(i!=0 || j!=0){
                    MPI_Recv(&C[i*(N_loc*n)+(j*N_loc)], 1, col_type, i*dims[0]+j, i*dims
                        [0]+j, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                }
            }
        }
        MPI_Sendrecv(C_loc, N_loc*N_loc, MPI_DOUBLE, root, root, C, 1, col_type, root,
             root, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else{
        MPI_Send(C_loc, N_loc*N_loc, MPI_DOUBLE, root, rank, MPI_COMM_WORLD);
    }

 if(rank==root){
        printf("%.4lf\n",glob_time);
        write_output(output_name, C, n);
        free(C);
    }

    free(A_loc);
    free(B_loc);
    free(C_loc);
    MPI_Comm_free( &comm2D );
    MPI_Finalize();
    return 0;
}
```