UPPSALA
UNIVERSITET

Parallell and Distributed Programming
Assignment 2

Authors:

Aikaterini Manousidou

Moa Li Lekander

Uppsala

June 17, 2021

Contents

1 Introduction

The matrix-matrix multiplication operation is of use in many areas of science. In a large variety of research fields, large matrices often have to be multiplied which requires heavy computational resources. A way of making these calculations more efficient and reducing the computational time, is to do them in parallel. In this assignment a parallel matrix-matrix multiplication algorithm for square dense matrices is implemented using C and MPI. The performance of the implementation is then evaluated by examining the strong and weak scaling of the program.

1.1 Theory

1.1.1 Cannon's Algorithm

One algorithm for performing parallelized matrix-matrix multiplications on a distributed memory is Cannon's algorithm. Firstly, this algorithm divides the matrices that are to be multiplied into quadratic grids (see matrices A and B in Figure 1), and then distributes the data so that every process stores one of these grids. Every process will compute the corresponding grid in the resulting C matrix (assuming $AB = C$).

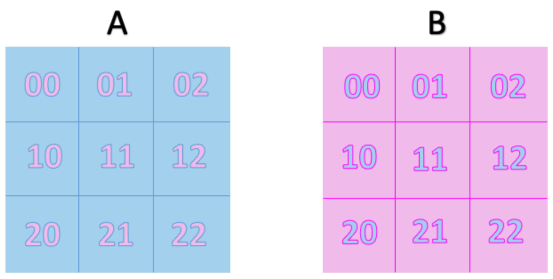

Figure 1: Visualization of matrices A and B partitioned into grids, when 9 processes are used.

In the next step, the local matrices in every process must be shifted in a way so that all processes can begin performing matrix multiplication on their local sub-matrices. This desired distribution is achieved by shifting the sub-matrices `i` cyclic steps to the left in the A matrix and `j` cyclic steps upwards in the B matrix, where `i` is the row of the position of the process and `j` is the column (see Figure 2 for result after shifting).

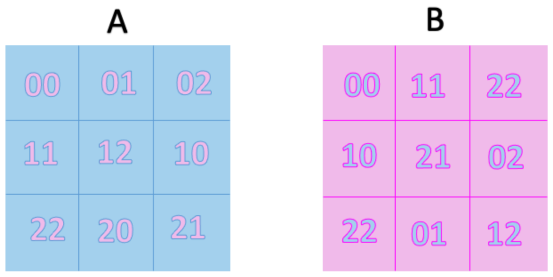

Figure 2: Visualization of matrices A and B partitioned into grids after initial shift, when 9 processes are used.

When the initial distribution has been achieved and the local multiplications has been made, each sub-matrix of A is shifted one cyclic step to the left and one cyclic step up for the sub-matrices in B. The partial results are added together, and this procedure is then repeated until all sub-matrices have been multiplied.

1.1.2 Strong and Weak Scaling

To evaluate the performance of a program, both strong and weak scalability tests can be made. A test of a programs strong scalability means that the problem size is kept constant while the number of processes increase. To analyze the strong scalability the speedup of the program can be studied. The strong scalability speedup can be computed using the measured time for the serial program and the parallelized program, according to equation (1), where N is the size of the problem, p is the number of processes, S is the speedup and $T(N,1)$ and $T(N,p)$ are the timings for one and p processes, respectively. [1]

$$S(N,p) = \frac{T(N,1)}{T(N,p)} \tag{1}$$

Similarly, in a weak scaling analysis both the size of the problem and the number of processes is increased, in a way so that the amount of work provided for every process remains constant. A tool that also can be used to analyze the performance of the program is the parallel efficiency, which is presented in equation (2).

$$E(N,p) = \frac{T(N,1)}{p \cdot T(N,p)} \tag{2}$$

In summary, a program that is strongly scalable retains constant efficiency as the problem size stays the same, and a weakly scalable program retains a constant efficiency as the problem size increases.[1]

2 Implementation

2.1 Partitioning Strategy

Since Cannon's algorithm was chosen in this assignment, it was most suitable to partition the matrices in a 2D block checkerboard strategy, so that the processes would line up with the grid structure that was used in the algorithm (as visualized in Figure 1 and 2). As this partitioning strategy was used it had to be assumed that the size of the matrices was divisible by the square root of the number of processes used, in order to be able to partition the processes in the described grid structure. As this assumption is required for the algorithm to work, many combinations of matrix sizes and number of processes used might have to be discarded which is a disadvantage of the chosen algorithm. Also, the setup of this algorithm implies that relatively many messages have to be sent between the processes which also is a negative aspect. An advantage of this partitioning strategy together with Cannon's algorithm however, is that the required amount of memory is constant and does not depend on the number of processes (as it only depends on the matrix size).

2.2 The Code

The implementation of the program can be found in Appendix section 6.2. In the first part of the code the MPI is initialized and the 2D Cartesian topology is created using the function `MPI_Cart_create`, with the `periods` parameter set to `1` to enable a periodic structure (so that the cyclic shifts will easily be done). The total number of processes, the local rank and its coordinates in the Cartesian topology are stored in the variables `size`, `rank` and `grid_coordinates` respectively.

The program is called with two input arguments that specify the input- and output files that are to be used. The input file provides the size of the matrix together with the elements of both the A and B matrices that are to be multiplied. The process of rank 0 reads the input file and stores the values in the matrices `A` and `B` using the written function `readInput`. Here, the assumptions required for the algorithm to work are also checked and the program prints an error message and terminates if they are not fulfilled. The size of the matrices stored in the variable `n` is sent to all processes using the function `MPI_Bcast`, and thereafter the size of the local matrices that are to be distributed to all the processes is computed on every process and stored in the local variable `chunk`. After this, parts of the `A` and `B` matrices are distributed from the process of rank 0 to all processes using a defined `MPI_Type_vector` with `MPI_Isend`, `MPI_Irecv` and `MPI_Wait` calls. The smaller matrices are stored in the local matrices `localA` and `localB`.

When the matrices have been split and distributed among the processes it is time to begin the start up phase of the algorithm, which is to shift the matrices to obtain the initial distribution that is presented in Figure 2. This is achieved by using the function `MPI_Cart_shift` to get which ranks every process should send and receive their local matrices to and from. That information is then used with calls to `MPI_Isend`, `MPI_Irecv` and `MPI_Wait` together with two local buffer arrays (`buffA` and `buffB`) which have the same size as the local arrays, to perform the actual shifting of matrices. Then, the compute phase of the algorithm begins and a `localC` matrix is allocated on every process. Afterwards, the matrix multiplications are made on the local matrices and shifts are made according to the described process in section 1.1.1. As the matrices in this program are stored in row-major order, the matrix multiplication is made in the "ikj" order for the purpose of making the accessing of the matrix elements as efficient as possible.

After all `localC` matrices have been computed, the results are sent back to the process of rank 0 using `MPI_Isend`, `MPI_Irecv` and `MPI_Wait` calls and the final result is stored in the `C` matrix that was allocated on the process of rank 0. Thereafter, the result is written to the specified output file by rank 0 and all the allocated memory is freed before the program terminates.

3 Numerical Experiments

When evaluating the performance of the program both the weak and strong scalings were analyzed. To do this, a set of experiments were performed on the UPPMAX system Rackham, which enabled a total of 2 nodes or 40 cores to be used in the experiments of this assignment.

The size of the matrices used when performing the strong scaling experiments was 3600, meaning that the number of processes that could be used was 1, serving as the serial code, 4, 9, 16, 25 and 36. The execution time, including the matrix-matrix multiplication and the communications, was measured for every process and the longest of them was considered as the final execution time. It was expected that the implementation would perform better with the increase of processes and perform the best, as in produce the shortest execution time, with the highest number of processes, in this case 36.

For the weak scaling analysis, the size of the matrices had to increase as the number of processes increased, resulting in a constant matrix-size-per-process relation. This implied that the size of the local matrices, `localA`, `localB` and `localC`, previously called `chunk`, had to be kept constant. The `chunk` was set to 1200, following that sizes of 1200, 2400, 3600, 4800, 6000 and 7200 were tested for processes of 1, 4, 9, 16, 25 and 36, respectively. Since these matrix sizes were not provided, the matrices were produced in MATLAB using the code displayed in Appendix section 6.1. In this analysis it was expected that the execution time of implementation would steadily increase due to the increased demand of sending and receiving data between the processes.

4 Results

The strong scaling speedup was calculated with equation (1). The results of the strong scaling analysis can be found in Table 1 and Figure 3.

Table 1: Table over the number of processes, the matrix multiplication timings acquired during the strong scaling analysis and the calculated speedup. All values have been rounded to five significant digits.

Processes	Matrix multiplication time (s)	Speedup
1	30.8563	1.0000
4	10.3525	2.9806
9	4.7014	6.5632
16	3.0197	10.2184
25	1.3381	23.0592
36	0.9504	32.4668

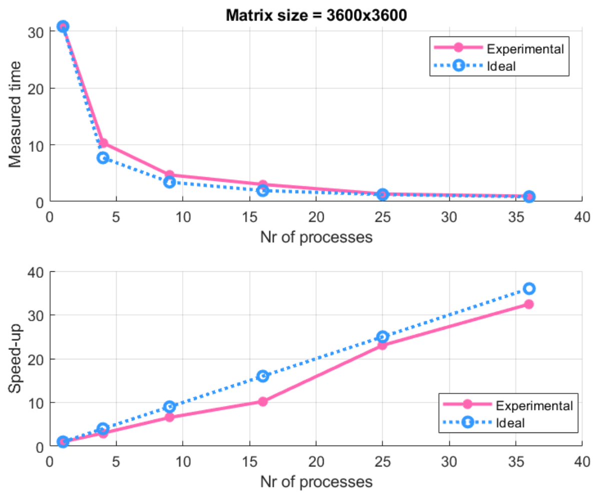

Figure 3: Plot over the strong scaling analysis, illustrating the relations between the number of processes and the measured timings and the number of processes and the speedup, respectively.

The result from the weak scaling experiment is presented in Table 2. Those values are also presented in Figure 4 together with the ideal case.

Table 2: Table over the number of processes and the corresponding matrix multiplication timings acquired during the weak scaling analysis. All values have been rounded to five significant digits.

Processes	Matrix size	Matrix multiplication time (s)
1	1200	0.4858
4	2400	1.7517
9	3600	4.7454
16	4800	8.8938
25	6000	12.2543
36	7200	14.7457

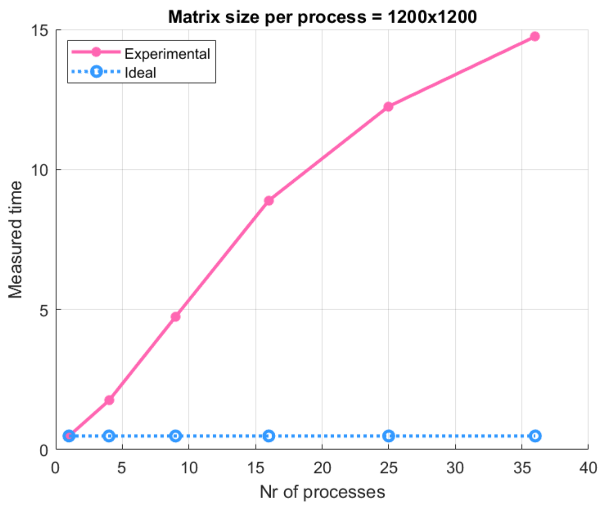

Figure 4: Measured times depending on the number of processes in the weak scaling experiment, together with the ideal case.

Lastly, the efficiency of the strong and weak scalability experiments for this implementation was calculated with equation (2) and can be found in Figure 5.

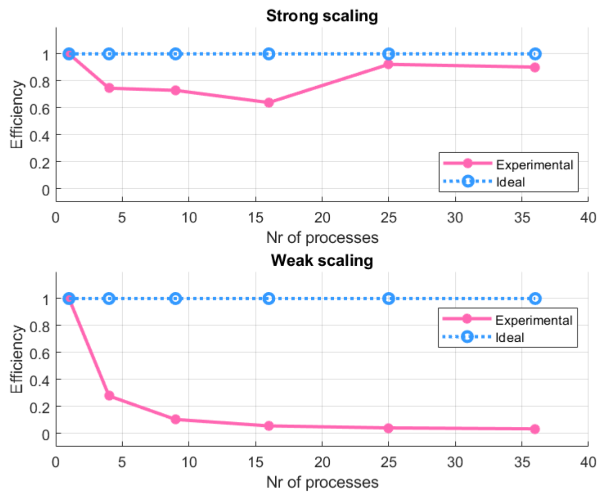

Figure 5: Plot of the efficiencies from the strong and weak scaling analysis.

5 Discussion

From Table 1 and Figure 3, it can be concluded that the measured timings and speedup seem to follow the trend of the ideal case, very closely. Thereby, it is implied that the implementation appears to be strongly scalable.

From Table 2 and Figure 4, it can be determined that the measured timings increase as the amount of processes initiated increases. The relation between the measured time and the processes appears to be linear. With respect to the ideal slope of the measured timings, it is indicated that the implementation does not seem to be weakly scalable. Thus, the hypothesis stated in section 3 seems to be correct.

Lastly, from Figure 5 it can be concluded that the program appear to somewhat follow the trend line of the ideal cases in the strong scaling. However, by observing the efficiency of the weak scaling, the conclusions drawn from Figure 4 and 2 can once again be verified. Thus, the program is strongly scalable but not weakly scalable.

References

[1] P. Pacheco, *An Introduction to Parallel Programming.* International series of monographs on physics, Morgan Kaufmann, 2011.

6 Appendix

6.1 Input Matrices Code

```matlab

%------------- PDP ASSIGNMENT 2 -------------

%----------------- Matrices -----------------

% Matrix sizes
n = [1200 2400 3600 4800 6000 7200];

% Nr of decimals
decimal = 6;

% Creating matrices with random numbers
input1200 = rand(2*n(1),n(1));
input2400 = rand(2*n(2),n(2));
input3600 = rand(2*n(3),n(3));
input4800 = rand(2*n(4),n(4));
input6000 = rand(2*n(5),n(5));
input7200 = rand(2*n(6),n(6));

% Rounding off
for i=1:(2*n)
    for j=1:n
        input1200(i,j)=round(10^decimal*input1200(i,j))/10^decimal;
        input2400(i,j)=round(10^decimal*input2400(i,j))/10^decimal;
        input3600(i,j)=round(10^decimal*input3600(i,j))/10^decimal;
        input4800(i,j)=round(10^decimal*input4800(i,j))/10^decimal;
        input6000(i,j)=round(10^decimal*input6000(i,j))/10^decimal;
        input7200(i,j)=round(10^decimal*input7200(i,j))/10^decimal;
    end
end

% Writing to txt files
writematrix(input1200,'input1200.txt','Delimiter','space');
writematrix(input2400,'input2400.txt','Delimiter','space');
writematrix(input3600,'input3600.txt','Delimiter','space');
writematrix(input4800,'input4800.txt','Delimiter','space');
writematrix(input6000,'input6000.txt','Delimiter','space');
writematrix(input7200,'input7200.txt','Delimiter','space');
```

6.2 Parallelized code

```c
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void readInput(double* A, double* B, int n, FILE *stream_in);
void writeOutput(char* filename, double* C, int n);
void printMatrix(double* M, int n, int row, int col);

int main(int argc, char *argv[]) {
  if(argc != 3){
    printf("Expected input: matmul input_filename output_filename\n");
    exit(0);
  }
  int rank, size, i, j, k, l, chunk, n, grid_size;
  double *A, *B, *C, *localA, *localB, *localC, *buffA, *buffB, *tempA, *tempB;
  MPI_Status status;
  MPI_Request request, request_send, requests[4];

  MPI_Init(&argc, &argv);               // Initialize MPI
  MPI_Comm_size(MPI_COMM_WORLD, &size); // Get the number of processors
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get my number

  // Create a 2D Cartesian topology
  MPI_Comm comm_grid;
  int dims[2], periods[2], reorder, grid_coordinates[2];
  dims[0] = dims[1] = sqrt(size);
  periods[0] = periods[1] = 1;
  reorder = 0;
  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &comm_grid);
  MPI_Cart_coords(comm_grid, rank, 2, grid_coordinates);

  // Let rank 0 read input data
  if(rank == 0){
    char* input_filename = argv[1];

    // Open inputfile for reading
    FILE *stream_in;
    stream_in = fopen(input_filename, "r");
    if(stream_in == NULL){
      printf("Error: Unable to open file: %s\n", input_filename);
      fclose(stream_in);
      exit(0);
    }

    // Read size of matrix
    fscanf(stream_in, "%d ", &n);

    // Check assumption
    if(rank == 0){
      if((int)sqrt(size) * (int)sqrt(size) != size || n % (int)sqrt(size) != 0) {
        printf("ERROR: Matrix cannot be divided into submatrices.\n");
        fclose(stream_in);
        exit(0);
      }
    }

    // Allocate memory for matrices
    A = (double*)malloc(n*n*sizeof(double));
    B = (double*)malloc(n*n*sizeof(double));

    // Read input to matrices
    readInput(A, B, n, stream_in);

    // Print matrices
    /*
    printf("A: \n");
    printMatrix(A, n, n, n);
    printf("\nB: \n");
    printMatrix(B, n, n, n);
    */

  }
  MPI_Barrier(MPI_COMM_WORLD);

  // Start timer
  double starttime = MPI_Wtime();

  // Send n to all processes and compute chunk
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  chunk = n/sqrt(size); // local matrix size

  // Send blocks (of size chunk*chunk) of A and B to processes
  MPI_Datatype newtype;
  int count = chunk, blocklen = chunk, stride = n;
  MPI_Type_vector(count, blocklen, stride, MPI_DOUBLE, &newtype);
  MPI_Type_commit(&newtype);

  // Allocate memory for local matrices
  localA = (double*)malloc(chunk*chunk*sizeof(double));
  localB = (double*)malloc(chunk*chunk*sizeof(double));

  // Send submatrices to processes
  grid_size = sqrt(size);
  if (rank==0) {
    int rank_counter = 0;
    for(i = 0; i < grid_size; i++){
      for(j = 0; j < grid_size; j++){
        MPI_Isend(&A[(i*chunk*n)+(j*chunk)], 1, newtype, rank_counter, 100+rank_counter, ...
            MPI_COMM_WORLD, &request);
        MPI_Isend(&B[(i*chunk*n)+(j*chunk)], 1, newtype, rank_counter, 200+rank_counter, ...
            MPI_COMM_WORLD, &request);
        rank_counter++;
      }
    }
  }
  MPI_Irecv(&localA[0], chunk*chunk, MPI_DOUBLE, 0, 100+rank, MPI_COMM_WORLD, &request);
  MPI_Wait(&request, &status);
  MPI_Irecv(&localB[0], chunk*chunk, MPI_DOUBLE, 0, 200+rank, MPI_COMM_WORLD, &request);
  MPI_Wait(&request, &status);

  // Perform matrix-matrix multiplication: Cannon's algorithm
  // START UP PHASE
  i = grid_coordinates[0];
  j = grid_coordinates[1];
  int up, left, down, right, source, destination;
  if(i != 0){
    MPI_Cart_shift(comm_grid, 1, -i, &source, &left);
    MPI_Cart_shift(comm_grid, 1, i, &source, &right);
  }
  if(j != 0){
    MPI_Cart_shift(comm_grid, 0, -j, &source, &up);
    MPI_Cart_shift(comm_grid, 0, j, &source, &down);
  }

  buffA = (double*)calloc(chunk*chunk,sizeof(double));
  buffB = (double*)calloc(chunk*chunk,sizeof(double));

  if (i != 0) {
    MPI_Isend(&localA[0], chunk*chunk, MPI_DOUBLE, left, 300+left, comm_grid, &request_send);
    MPI_Irecv(&buffA[0], chunk*chunk, MPI_DOUBLE, right, 300+rank, comm_grid, &request);
    MPI_Wait(&request, &status);
    MPI_Wait(&request_send, &status);

    tempA = localA;
    localA = buffA;
    buffA = tempA;
  }
  if (j != 0) {
    MPI_Isend(&localB[0], chunk*chunk, MPI_DOUBLE, up, 400+up, comm_grid, &request_send);
    MPI_Irecv(&buffB[0], chunk*chunk, MPI_DOUBLE, down, 400+rank, comm_grid, &request);
    MPI_Wait(&request, &status);
    MPI_Wait(&request_send, &status);

    tempB = localB;
    localB = buffB;
    buffB = tempB;
  }

  // COMPUTE PHASE
  localC = (double*)calloc(chunk*chunk,sizeof(double));

  MPI_Cart_shift(comm_grid, 1, -1, &source, &left);
  MPI_Cart_shift(comm_grid, 1, 1, &source, &right);
  MPI_Cart_shift(comm_grid, 0, -1, &source, &up);
  MPI_Cart_shift(comm_grid, 0, 1, &source, &down);

  for(l = 0; l < grid_size; l++){
    MPI_Isend(&localA[0], chunk*chunk, MPI_DOUBLE, left, 500+left, comm_grid, &requests[0]);
    MPI_Irecv(&buffA[0], chunk*chunk, MPI_DOUBLE, right, 500+rank, comm_grid, &requests[1]);
    MPI_Isend(&localB[0], chunk*chunk, MPI_DOUBLE, up, 600+up, comm_grid, &requests[2]);
    MPI_Irecv(&buffB[0], chunk*chunk, MPI_DOUBLE, down, 600+rank, comm_grid, &requests[3]);

    // Do matrix-matrix multiplication
    for(int u = 0; u < chunk; u++){ // i
      for(int v = 0; v < chunk; v++){ // k
        for(int s = 0; s < chunk; s++){ // j
          localC[u*chunk+s] += localA[u*chunk+v] * localB[v*chunk+s];
        }
      }
    }
    MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);

    tempA = localA;
    tempB = localB;

    localA = buffA;
    localB = buffB;

    buffA = tempA;
    buffB = tempB;
  }
  MPI_Barrier(MPI_COMM_WORLD);

  // Send localC to rank 0
  MPI_Isend(&localC[0], chunk*chunk, MPI_DOUBLE, 0, 1000+rank, MPI_COMM_WORLD, &request_send);
  if(rank == 0){
    C = (double*)calloc(n*n,sizeof(double));
    int rank_counter = 0;
    for(k = 0; k < sqrt(size); k++){
      for(i = 0; i < sqrt(size); i++){
```

```c
        MPI_Irecv(&C[k*chunk*n+i*chunk], 1, newtype, rank_counter, 1000+rank_counter, ...
            MPI_COMM_WORLD, &request);
        MPI_Wait(&request, &status);

        rank_counter++;
      }
    }
  }
  MPI_Wait(&request_send, &status);

  // Stop timer
  double execution_time = MPI_Wtime()-starttime; // stop timer
  double max_time;
    MPI_Reduce(&execution_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    if(rank==0){
    printf("%lf (size=%d)\n", max_time, size);
    // Print result
    /*
    if(rank == 0){
      printf("\nC: \n");
      printMatrix(C, n, n, n);
    }
    */
    char* output_filename = argv[2];
    writeOutput(output_filename, C, n);
  }

  // Clean up
  if(rank == 0){
    free(A);
    free(B);
    free(C);
  }
  free(buffA);
  free(buffB);
  free(localA);
  free(localB);
  free(localC);
  MPI_Finalize();
  return 0;
}

void readInput(double* A, double* B, int n, FILE *stream_in){
  for(int i = 0; i < n*n; i++){
    fscanf(stream_in, "%lf ", &A[i]);
  }
  for(int i = 0; i < n*n; i++){
    fscanf(stream_in, "%lf ", &B[i]);
  }
  fclose(stream_in);
}

void writeOutput(char* filename, double* C, int n){
  FILE *stream_out;
  stream_out = fopen(filename,"w");
  if(stream_out == NULL){
    printf("Error: unable to open file: %s\n", filename);
    exit(0);
  }
  for(int i = 0; i<n*n; i++){
    fprintf(stream_out, "%-10f", C[i]);
  }
  fclose(stream_out);
}
```

```c

void printMatrix(double* M, int n, int row, int col){
  for(int i = 0; i < row; i++){
    for(int j = 0; j < col; j++){
      printf("%10f ", M[i*n+j]);
    }
    printf("\n");
  }
}
```

6.3 Performance Evaluation Code

```
close all;
clear all;

%------------- PDP ASSIGNMENT 2 -------------

%------------- Strong Scaling -------------

% Matrix size = 3600
cores = [1, 4, 9, 16, 25, 36];

% Measured times

time1 = zeros(1,length(cores));
time1(1) = 30.856336; % 1 core
time1(2) = 10.352450; % 4 cores
time1(3) = 4.701412; % 9 cores
time1(4) = 3.019678; % 16 cores
time1(5) = 1.338134; % 25 cores
time1(6) = 0.950398; % 36 cores

% Calculating speed-up, efficiency and ideal time
speedup1 = zeros(1,length(cores));
efficiency1 = zeros(1,length(cores));
t1 = zeros(1,length(cores));

for i=1:length(cores)
    speedup1(i) = time1(1)/time1(i);
    efficiency1(i) = speedup1(i)/cores(i);
    t1(i) = time1(1)/cores(i);
end

% Generating figures
figure
subplot(2,1,1)
hold on
grid on
plot(cores, time1, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores, t1, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
legend('Experimental', 'Ideal');
title('Matrix size = 3600x3600');
xlabel('Nr of processes');
ylabel('Measured time');
hold off
subplot(2,1,2)
hold on
grid on
plot(cores,speedup1, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores, cores, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
legend('Experimental', 'Ideal');
xlabel('Nr of processes');
ylabel('Speed-up');
hold off

%------------- Weak Scaling -------------

% Measured times
% Sizes = 1200 2400 3600 4800 6000 7200

time2 = zeros(1,length(cores));
time2(1) = 0.485748; % 1 core
time2(2) = 1.751739; % 4 cores
```

```
time2(3) = 4.745370; % 9 cores
time2(4) = 8.893792; % 16 cores
time2(5) = 12.254294; % 25 cores
time2(6) = 14.745703; % 36 cores

% Calculating speed-up, efficiency and ideal time
speedup2 = zeros(1,length(cores));
efficiency2 = zeros(1,length(cores));
t2 = time2(1)*ones(1,length(cores));
for i=1:length(cores)
    speedup2(i) = time2(1)/time2(i)*cores(i);
    efficiency2(i) = speedup2(i)/cores(i);
end

% Generating figures
figure
hold on
grid on
plot(cores, time2, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores, t2, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
legend('Experimental', 'Ideal');
title('Matrix size per process = 1200x1200');
xlabel('Nr of processes');
ylabel('Measured time');
hold off

% Calculating ideal efficiency
e1 = ones(1,length(cores));
e2 = ones(1,length(cores));

figure
subplot(2,1,1)
hold on
grid on
plot(cores, efficiency1, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores, e1, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
ylim([-0.1 1.2])
legend('Experimental', 'Ideal');
title('Strong scaling');
xlabel('Nr of processes');
ylabel('Efficiency');
hold off
subplot(2,1,2)
hold on
grid on
plot(cores,efficiency2, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores, e2, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
ylim([-0.1 1.2])
legend('Experimental', 'Ideal');
title('Weak scaling');
xlabel('Nr of processes');
ylabel('Efficiency');
hold off
```