UPPSALA
UNIVERSITET

Parallell and Distributed Programming
Assignment 1

Authors:

Aikaterini Manousidou

Moa Li Lekander

Uppsala

April 20, 2021

Contents

1 Introduction **1**
1.1 Theory 1

2 Parallelization **2**

3 Verification of Correctness **3**

4 Performance Experiments **4**

5 Discussion **7**

6 Appendix **9**
6.1 Serial Code 9
6.2 Parallelized code 11
6.3 Performance Verification and Evaluation Code 14

1 Introduction

A commonly used tool within image processing and computer simulations, is the operation of stencil applications. In this assignment a serial code is provided that initiates a one dimensional array of function values $f(x_i) = sin(\frac{2\pi i}{N})$, where N is the total number of function values in the array and i is the index of the value. Then, a stencil is applied on every element in the array. The stencil in this assignment is defined so that the two adjacent values on either side of the element in question are added together to obtain the new value, according to equation (1). In (1), $f(x_i)$ represents the element the stencil is applied to and $f(x_{i-j})$ are its adjacent elements to the left and $f(x_{i+j})$ the adjacent elements to the right. This stencil is used to approximate the first derivative of the function values in the array.

$$\frac{1}{12h} \cdot f(x_{i-2}) - \frac{8}{12h} \cdot f(x_{i-1}) + 0 \cdot f(x_i) + \frac{8}{12h} \cdot f(x_{i+1}) - \frac{1}{12h} \cdot f(x_{i+2}) \tag{1}$$

The aim of this assignment is to parallelize the given serial code using the *Message Passing Interface* (MPI). The number of function values and the number of times the stencil is to be applied are provided as input to the program. The boundary values of the array are periodic. Additionally, the implementation of the parallellized code assumes that the number of function values are divisible by the specified number of processes and that the number of values each process works on is larger than the width of the stencil. After the parallelized implementation is made its functionality will be verified and its performance evaluated by examining both the strong and weak scaling of the program.

1.1 Theory

A parallelized program can be evaluated with two different types of scalings, a strong and a weak scaling. A strong scaling analysis implies that the size of the program is kept constant while increasing the number of processes. A tool for the evaluation of the performance of a parallelized program is calculating the strong scaling speed-up, as seen in equation (2), where N is the size of the problem, p is the number of processes, S is the speed-up and T_{serial} and $T_{parallel}$ are the timings for the serial and parallelized codes, respectively. Ideally, the speed-up $S(N, p)$ should be equal to p. In other words, it should be linear and equal to the number of processes used.[1]

$$S(N, p) = \frac{T_{serial}(N)}{T_{parallel}(N, p)} \tag{2}$$

On the other hand, a weak scaling analysis indicates that the problem size increases relative to the number of processes, so that the workload for each process is kept constant. The speed-up for this type of scaling can be seen in equation (3).[2][3]

$$S(N, p) = \frac{T_{serial}(N)}{T_{parallel}(N, p)} \times p \tag{3}$$

Another tool that is widely used to evaluate parallel programs is the parallel efficiency, which can also be perceived as the speed-up per process and can be found in equation (4).

$$E(N, p) = \frac{S(N, p)}{p} = \frac{T_{serial}(N)}{p \times T_{parallel}(N, p)} \tag{4}$$

As a result, a program that retains constant efficiency while the problem size remains the same is called strongly scalable, although a program that retains constant efficiency while the problem size increases is called weakly scalable.[1]

2 Parallelization

The computations were parallelized by equally dividing the array values among the processes. Firstly, the process with rank zero fills the input array with function values, thereafter, the array is divided among the processes by the `MPI_Scatter` function. This function was used because it provides an efficient way of splitting the array into parts and sending each of the them to different processes. In every process, the given part of the array is stored in a local input array which has a total of two extra elements (initially empty) on each end. This was implemented for the purpose of handling the edges of the local arrays correctly.

A for-loop is used to repeatedly apply the stencil to the arrays. In the beginning of every loop iteration, the four extra elements of the local input arrays are filled with values by using the non-blocking send and receive functions `MPI_Isend` and `MPI_Irecv` together with `MPI_Wait`. These are used so that the function values needed to apply the stencil to the edge elements of the local array are obtained. Hence, every process communicates with the processes that store the adjacent parts of the array as it sends and receives the edge values. When all necessary function values are provided, the stencil application occurs and the new values are stored in a local output vector. Thereafter, a call to the function `MPI_Barrier` is made. This barrier function is used for the purpose of synchronizing the processes, in order to make sure that all processes have finished computing their new values for the given step before they all move on to the next iteration and apply the stencil again. Therefore, when all processes have called the barrier function the program will continue and the local input vector will be filled with the newly computed values stored in the local output vector before moving on to the next loop iteration.

pWhen the stencil has been applied the given number of steps, a call to the function `MPI_Gather` is made by the process with rank zero, in order to store all the final values in an output array. The maximum stencil application time for the processes is computed using the function `MPI_Reduce`, and finally all allocated memory is being freed before the program terminates. Both the serial and the parallelized programs can be found in the Apendix sections 6.1 and 6.2, respectively.

3 Verification of Correctness

In order to make sure that the functionality of the serial program was kept as the parallelized version was implemented, a verification of correctness was made. This verification was made by comparing the values in the produced output arrays obtained from the serial and parallelized programs, for different input values. Different number of processes were also used in the tests in order to verify that the program was working for different number of processes. Figure 1 presents the result of these measurements, where it can be observed that the computed output values are the same for both programs. In the two top plots in Figure 1 the output strongly resembles a negative sine curve, which is the expected result since the 10th derivative of a sine curve is a negative sine curve. In the same way, the output in the bottom plot in Figure 1 resembles a cosine curve, which corresponds to the 5th derivative of a sine curve. From these results, it could be concluded that the functionality of the serial program was preserved in the parallelized version.

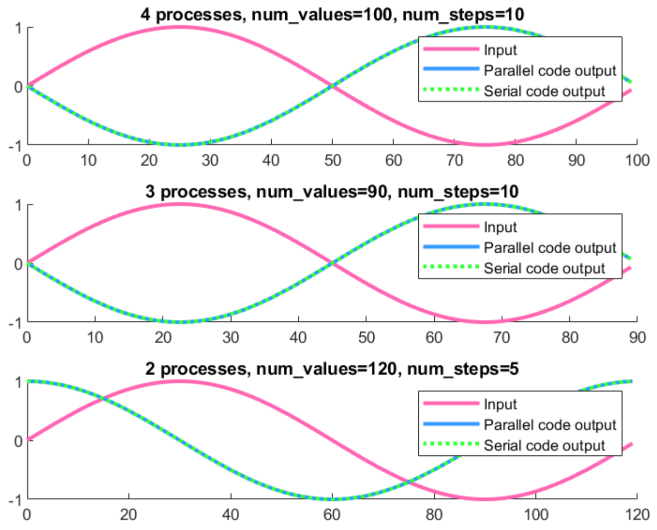

Figure 1: Input function values together with output function values for both the serial and the parallelized programs, for different input parameters and number of processes used for the parallelized code

4 Performance Experiments

In order to evaluate the performance of the parallelized code, a series of experiments was carried out. These experiments were performed on one of the scientific Linux hosts, `gullviva`. This specific computer acquires an AMD Opteron(TM) Processor 6274 with 2 sockets and 8 cores per socket, resulting in a total of 16 cores. This implies that the best and highest performance should be achieved when using 16 processes. Thereafter, the compiler Open MPI with version 1.8.1 was used to compile both the serial and parallelized programs.

First, a strong scaling analysis was performed. This implicated measuring the stencil application time for a problem of a fixed size but for an increasing number of processes. The number of function values that was chosen for this experiment was 1.6×10^7 in order to maximize the amount of processes this number was divisible with. The number of steps was set to 100 and the amounts of processes tested for the parallelized code were 2, 4, 5, 8, 10, and 16. After obtaining the stencil application times, the speed-up was calculated with the help of equation (2) and Figure 2 was generated, illustrating the relations between the number of processes, stencil application time and speed-up. Figure 2 also portrays the ideal values of the stencil applications timings and the ideal speed-up. The numeric values of the measured timings can be found in Table 1.

Table 1: Table over the number of processes, the stencil application timings acquired during the strong scaling analysis and the calculated speed-up. All values have been rounded to five significant digits.

Processes	Stencil application time (s)	Speed-up
1	76.0257	1.0000
2	73.3039	1.0371
4	32.0735	2.3704
5	37.9536	2.0031
8	23.4678	3.2396
10	25.7712	2.9500
16	19.0992	3.9806

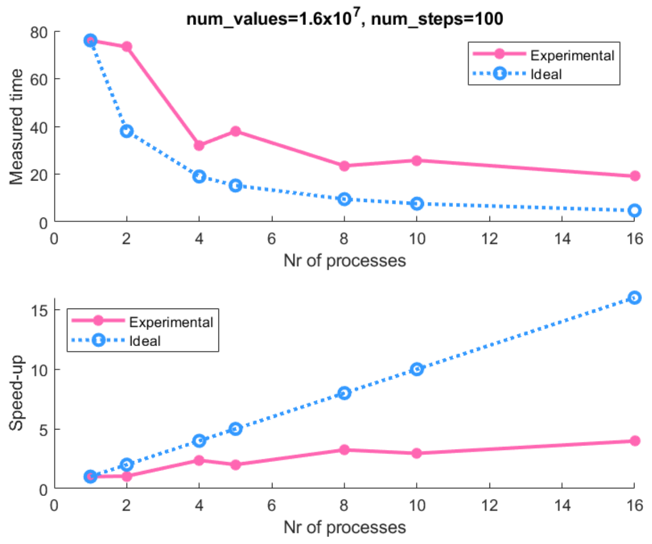

Figure 2: Plot over the strong scaling analysis, illustrating the relations between the number of processes and the measured timings and the number of processes and the speed-up, respectively.

Then, a weak scaling analysis was performed. This implied measuring the stencil application time for a number of values of $1000000 \times p$, where p depicts the number of processes. The number of steps was, once again, set to 100 and the numbers of processes tested for the parallelized code were increased from 2 to 16, with increments of 1. After measuring the stencil application timings, the speed-up was determined with the help of equation (3). Figure 3 was thereupon produced, depicting the relations between the number of processes, stencil application time and speed-up. In Figure 3 the ideal values of the stencil applications timings and the the speed-up are also represented. The numeric values of the measured timings can be found in Table 2.

Table 2: Table over the number of processes, the stencil application timings acquired during the weak scaling analysis and the calculated speed-up. All values have been rounded to five significant digits.

Processes	Problem size	Stencil application time (s)	Speed-up
1	1×10^6	0.4465	1.0000
2	2×10^6	0.7506	1.1898
3	3×10^6	0.7562	1.7716
4	4×10^6	0.7712	2.3160
5	5×10^6	1.0157	2.1983
6	6×10^6	1.0515	2.5480
7	7×10^6	1.0625	2.9419
8	8×10^6	1.0943	3.2645
9	9×10^6	1.3855	2.9007
10	10×10^6	1.4634	3.0515
11	11×10^6	1.4390	3.4135
12	12×10^6	1.4444	3.7099
13	13×10^6	1.7913	3.2407
14	14×10^6	1.8479	3.3830
15	15×10^6	1.8560	3.6089
16	16×10^6	2.3554	3.0333

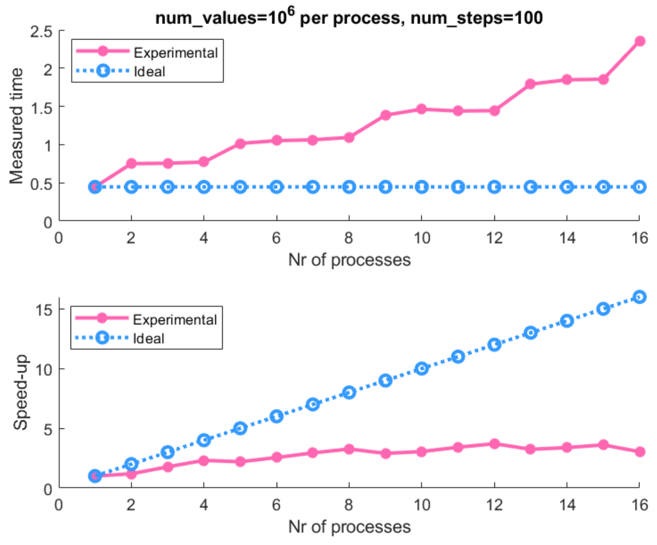

Figure 3: Plot over the weak scaling analysis, illustrating the relations between the number of processes and the measured time and the number of processes and the speed-up, respectively.

Finally, Figure 4 was produced by calculating the efficiency with the help of equation (4). The code for the verification and evaluation of the parallelized program can be found in the Appendix section 6.3.

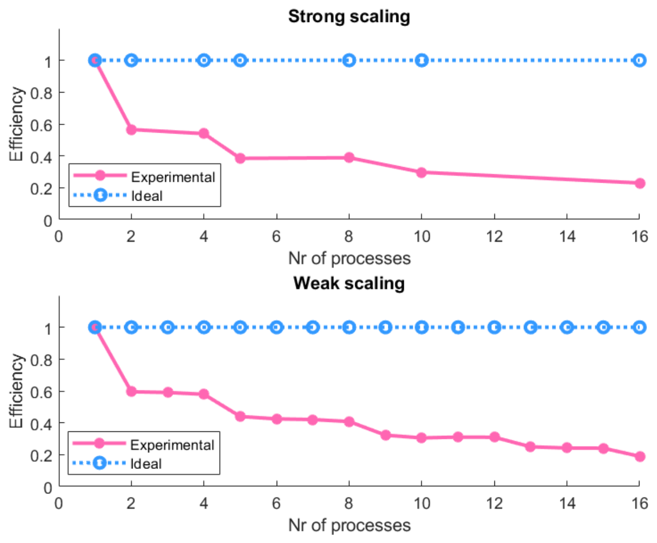

Figure 4: Plot over the efficiencies of the strong and weak scaling analysis.

5 Discussion

From the results presented in Figure 2, it can be concluded that the measured timings seem to follow the trend line of the ideal case. However, there is a difference in the time for the measured values as they are slightly larger than the ideal values, which thereby results in a smaller speedup. This implies that the parallelized program does not appear to be strongly scalable.

When analyzing the weak scaling presented in Figure 3 it can be noted that the measured time increases as the amount of processes involved increase. The curve seems to have a somewhat linear appearance, and as a result the speedup seems to be constant throughout the different number of processes. By comparing to the ideal case, the measured curves seem to have different slopes. This implicates that the parallelized code does not seems to be weakly scalable.

From Figure 4 it can be concluded that the final and parallelized program does not appear to follow the ideal cases, on the contrary it seems to decrease in both types of scalings. This, once again, confirms the conclusions drawn from Figures 2 and 3 that the program is neither strongly scalable nor weakly scalable. On the other hand, it can be noted that the program's best performance occurs when the maximum amount of processes is being used, which is as expected.

References

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

[2] F. Wermelinger, Computational Science and Engineering Laboratory, "Strong and Weak Scaling."

[3] Research Computing, Office of Information Technology, University of Colorado Boulder, "Scaling Analysis."

6 Appendix

6.1 Serial Code

```c
#define PI 3.14159265358979323846
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv) {
    if (3 != argc) {
        printf("Usage: stencil num_values num_steps\n");
        return 0;
    }
  int num_values = atoi(argv[1]);
    int num_steps = atoi(argv[2]);
    int rank, size;

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

    // Generate values for stencil operation
    double *input=(double*)malloc(num_values*sizeof(double));
  double h = 2.0*PI/num_values;
  for (int i=0; i<num_values; i++) input[i]=sin(h*i);

    // Stencil values
    const int STENCIL_WIDTH = 5;
    const int EXTENT = STENCIL_WIDTH/2;
    const double STENCIL[] = {1.0/(12*h), -8.0/(12*h), 0.0, 8.0/(12*h), -1.0/(12*h)};

    // Start timer
    double start = MPI_Wtime();

  // Allocate data for result
  double *output =(double*)malloc(num_values*sizeof(double));

    // Print input values
    /*if(rank == 0){
        printf("input:\n");
        for(int i=0; i<num_values; i++){
            printf("%f ", input[i]);
        }
        printf("\n");
    }*/

    // Repeatedly apply stencil
    for (int s=0; s<num_steps; s++) {
    // Apply stencil on left boundary with periodic cond
        for (int i=0; i<EXTENT; i++) {
            double result = 0;
            for (int j=0; j<STENCIL_WIDTH; j++) {
                int index = (i - EXTENT + j + num_values) % num_values;
                result += STENCIL[j] * input[index];
            }
            output[i] = result;
        }

    // Apply stencil on inner points
        for (int i=EXTENT; i<num_values-EXTENT; i++) {
            double result = 0;
            for (int j=0; j<STENCIL_WIDTH; j++) {
                int index = i - EXTENT + j;
                result += STENCIL[j] * input[index];
            }
            output[i] = result;
        }

    // Apply stencil on right boundary with periodic cond
        for (int i=num_values-EXTENT; i<num_values; i++) {
            double result = 0;
            for (int j=0; j<STENCIL_WIDTH; j++) {
                int index = (i - EXTENT + j) % num_values;
                result += STENCIL[j] * input[index];
            }
            output[i] = result;
        }

        // Swap input and output
        if (s < num_steps-1) {
            double *tmp = input;
            input = output;
            output = tmp;
        }
    }

    // Stop timer
    double my_execution_time = MPI_Wtime() - start;
  printf("%f\n", my_execution_time);

    //Print output vector
    /*if(rank == 0){
        printf("\noutput:\n");
        for(int i=0; i<num_values; i++){
            printf("%f ", output[i]);
        }
        printf("\n");
    }*/

    // Write output to file
  /*FILE *file=fopen("output_old.txt","w");
  for (int i = 0; i < num_values; i++)
    fprintf(file, "%f \n", output[i]);
  fclose(file);
  */

    // Clean up
  free(input);
    free(output);
  MPI_Finalize();
    return 0;
}
```

6.2 Parallelized code

```c
#define PI 3.14159265358979323846
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv) {
    if (3 != argc) {
        printf("Usage: stencil num_values num_steps\n");
        return 0;
    }
  int num_values = atoi(argv[1]);
    int num_steps = atoi(argv[2]);
    int rank, size;
    double execution_time, total_start_time, start_time, max_time, total_time;

    MPI_Status status;
    MPI_Request request;

  MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size); /* Get the number of processors */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* Get my number                */
    int chunk = num_values/size; // number of values for every process

    // Stencil values
    const int STENCIL_WIDTH = 5;
    const int EXTENT = STENCIL_WIDTH/2;
    double h = 2.0*PI/num_values;
    const double STENCIL[] = {1.0/(12*h), -8.0/(12*h), 0.0, 8.0/(12*h), -1.0/(12*h)};

    // Check assumptions
    if(num_values%size != 0) {
        if(rank==0) printf("ERROR: num_values must be divisible by number of processes!\n");
        exit(0);
    } else if(chunk < STENCIL_WIDTH){
        if(rank==0) printf("ERROR: Too many processors for given num_values!\n");
        exit(1);
    }

    // Allocate data for input and output
    double *input=(double*)malloc(num_values*sizeof(double));
    double *output =(double*)malloc(num_values*sizeof(double));
    double *local_input =(double *)malloc((chunk+EXTENT*2)*sizeof(double));
    double *local_output =(double *)malloc(chunk*sizeof(double));

    if(rank==0){
        // Generate values for stencil operation
    for (int i=0; i<num_values; i++) input[i]=sin(h*i);

        // Print input values
        /*printf("input:\n");
        for(int i=0; i<num_values; i++){
            printf("%f ", input[i]);
        }
        printf("\n");*/

        // Write input values to file
        /*
    FILE *file1=fopen("input.txt","w");
    for (int i = 0; i < num_values; i++)
        fprintf(file1, "%f \n", input[i]);
    fclose(file1);*/

        total_start_time = MPI_Wtime();
    }

    MPI_Scatter(&input[0], chunk, MPI_DOUBLE, &local_input[EXTENT], chunk, MPI_DOUBLE, 0, ...
        MPI_COMM_WORLD);

    // Start timer
    start_time = MPI_Wtime();

    // Repeatedly apply stencil
    for (int s=0; s<num_steps; s++) {
        if(rank==0){
            MPI_Isend(&local_input[chunk+EXTENT-2], 2, MPI_DOUBLE, 1, 300+1, ...
                MPI_COMM_WORLD, &request);
            MPI_Isend(&local_input[2], 2, MPI_DOUBLE, size-1, 100+(size-1), ...
                MPI_COMM_WORLD, &request);

            MPI_Irecv(&local_input[chunk+EXTENT], 2, MPI_DOUBLE, 1, 100+rank, ...
                MPI_COMM_WORLD, &request);
            MPI_Wait(&request, &status);

            MPI_Irecv(&local_input[0], 2, MPI_DOUBLE, size-1, 300+0, MPI_COMM_WORLD, ...
                &request);
            MPI_Wait(&request, &status);
        } else if(rank==size-1){
            MPI_Isend(&local_input[chunk+EXTENT-2], 2, MPI_DOUBLE, 0, 300+0, ...
                MPI_COMM_WORLD, &request);
            MPI_Isend(&local_input[2], 2, MPI_DOUBLE, rank-1, 100+(rank-1), ...
                MPI_COMM_WORLD, &request);

            MPI_Irecv(&local_input[chunk+EXTENT], 2, MPI_DOUBLE, 0, 100+rank, ...
                MPI_COMM_WORLD, &request);
            MPI_Wait(&request, &status);
            MPI_Irecv(&local_input[0], 2, MPI_DOUBLE, rank-1, 300+rank, MPI_COMM_WORLD, ...
                &request);
            MPI_Wait(&request, &status);
        } else {
            MPI_Isend(&local_input[2], 2, MPI_DOUBLE, rank-1, 100+(rank-1), ...
                MPI_COMM_WORLD, &request);
            MPI_Isend(&local_input[chunk+EXTENT-2], 2, MPI_DOUBLE, rank+1, 300+(rank+1), ...
                MPI_COMM_WORLD, &request);

            MPI_Irecv(&local_input[0], 2, MPI_DOUBLE, rank-1, 300+rank, MPI_COMM_WORLD, ...
                &request);
            MPI_Wait(&request, &status);
            MPI_Irecv(&local_input[chunk+EXTENT], 2, MPI_DOUBLE, rank+1, 100+rank, ...
                MPI_COMM_WORLD, &request);
            MPI_Wait(&request, &status);
        }

    // Apply stencil
        for (int i=EXTENT; i<chunk+EXTENT; i++) {
            double result = 0;
            for (int j=0; j<STENCIL_WIDTH; j++) {
                int index = i - EXTENT + j;
                result += STENCIL[j] * local_input[index];
            }
            local_output[i-EXTENT] = result;
        }

        MPI_Barrier(MPI_COMM_WORLD); // synchronize processes

        // Swap input and output
        if (s < num_steps-1) {
            for(int i=0; i<chunk; i++){
                local_input[i+EXTENT] = local_output[i];
            }
        } else {
            double execution_time = MPI_Wtime()-start_time; // stop timer
            MPI_Gather(&local_output[0], chunk, MPI_DOUBLE, &output[0], chunk, MPI_DOUBLE, ...
                0, MPI_COMM_WORLD);
            if (rank == 0) total_time = MPI_Wtime()-total_start_time; // stop timer
        }
    }

    // Print output values
    /*if(rank==0){
        printf("output:\n");
        for(int i=0; i<num_values; i++){
            printf("%f ", output[i]);
        }
        printf("\n");
    }*/

    execution_time = MPI_Wtime()-start_time; // stop timer

    // Find max time among processors
    MPI_Reduce(&execution_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    // Display timing results
    if(rank==0){
        printf("Maximum stencil application time: %f\n", max_time);
        printf("Total time: %f\n", total_time);
    }

    /*
    // Write to file
  FILE *file=fopen("output.txt","w");
  for (int i = 0; i < num_values; i++)
    fprintf(file, "%f \n", output[i]);
  fclose(file);
  */

    // Clean up
  free(input);
    free(output);
    free(local_input);
    free(local_output);
  MPI_Finalize();
    return 0;
}
```

6.3 Performance Verification and Evaluation Code

```
close all;
clear all;

%------------- PDP ASSIGNMENT 1 -------------

%------------- Evaluation of output -------------

%Importing data from text-file and putting it in an array
input100= importdata ('input100.txt');
output100= importdata ('output2_100.txt');
output_old100= importdata ('output_old100.txt');
time100 = 0:1:99;

input90= importdata ('input90.txt');
output90= importdata ('output2_90.txt');
output_old90= importdata ('output_old90.txt');
time90 = 0:1:89;

input120= importdata ('input120.txt');
output120= importdata ('output2_120.txt');
output_old120= importdata ('output_old120.txt');
time120 = 0:1:119;

figure(1)
subplot(3,1,1);
hold on
plot(time100,input100, 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(time100,output100, 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
plot(time100,output_old100, ':', 'Color', [51/255 255/255 51/255], 'LineWidth', 2);
legend('Input', 'Parallel code output', 'Serial code output');
title('4 processes, num\_values=100, num\_steps=10');
hold off
subplot(3,1,2);
hold on
plot(time90,input90, 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(time90,output90, 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
plot(time90,output_old90, ':', 'Color', [51/255 255/255 51/255], 'LineWidth', 2);
legend('Input', 'Parallel code output', 'Serial code output');
title('3 processes, num\_values=90, num\_steps=10');
hold off
subplot(3,1,3);
hold on
plot(time120,input120, 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(time120,output120, 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
plot(time120,output_old120, ':', 'Color', [51/255 255/255 51/255], 'LineWidth', 2);
legend('Input', 'Parallel code output', 'Serial code output');
title('2 processes, num\_values=120, num\_steps=5');
hold off

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

cores1 = [1, 2, 4, 5, 8, 10, 16];

% Measured times

% num_values = 16 000 000
% num_steps = 100

% T1 = 6.795839; % 1 core
% time1 = zeros(1,length(cores1));
% time1(1) = T1;
```

```matlab
% time1(2) = 6.01033; % 2 cores
% time1(3) = 3.149910; % 4 cores
% time1(4) = 3.538132; % 5 cores
% time1(5) = 2.188927; % 8 cores
% time1(6) = 2.288942; % 10 cores
% time1(7) = 1.850582; % 16 cores

% num_values = 160 000 000
% num_steps = 100

T1 = 76.025657; % 1 core
time1 = zeros(1,length(cores1));
time1(1) = T1;
time1(2) = 73.303855; % 2 cores
time1(3) = 32.073502; % 4 cores
time1(4) = 37.953636; % 5 cores
time1(5) = 23.467748; % 8 cores
time1(6) = 25.771217; % 10 cores
time1(7) = 19.099210; % 16 cores

% Calculating speed-up, efficiency and ideal time
speedup1 = zeros(1,length(cores1));
efficiency1 = zeros(1,length(cores1));
t1 = zeros(1,length(cores1));
for i=1:length(cores1)
    speedup1(i) = T1/time1(i);
    efficiency1(i) = speedup1(i)/cores1(i);
    t1(i) = T1/cores1(i);
end

% Generating figures
figure(2)
subplot(2,1,1)
hold on
plot(cores1, time1, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores1, t1, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
legend('Experimental', 'Ideal');
title('num\_values=1.6x10^7, num\_steps=100');
xlabel('Nr of processes');
ylabel('Measured time');
hold off
subplot(2,1,2)
hold on
plot(cores1,speedup1, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores1, cores1, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
legend('Experimental', 'Ideal');
xlabel('Nr of processes');
ylabel('Speed-up');
hold off

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

cores2 = 1:1:16;

% Measured times

% 1000 num_values / process
% num_steps = 100

% T2 = 0.000323; % 1 core
% time2 = zeros(1,length(cores2));
% time2(1) = T2;
% time2(2) = 0.001286; % 2 cores
% time2(3) = 0.001558; % 3 cores
% time2(4) = 0.001879; % 4 cores
% time2(5) = 0.001912; % 5 cores
% time2(6) = 0.002038; % 6 cores
% time2(7) = 0.002130; % 7 cores
% time2(8) = 0.002668; % 8 cores
% time2(9) = 0.002908; % 9 cores
% time2(10) = 0.002942; % 10 cores
% time2(11) = 0.002956; % 11 cores
% time2(12) = 0.003156; % 12 cores
% time2(13) = 0.002898; % 13 cores
% time2(14) = 0.003546; % 14 cores
% time2(15) = 0.003381; % 15 cores
% time2(16) = 0.003897; % 16 cores

% 1 000 000 num_values / process
% num_steps = 100

T2 = 0.446543;
time2 = zeros(1,length(cores2));
time2(1) = T2;
time2(2) = 0.750622; % 2 cores
time2(3) = 0.756170; % 3 cores
time2(4) = 0.771220; % 4 cores
time2(5) = 1.015651; % 5 cores
time2(6) = 1.051516; % 6 cores
time2(7) = 1.062520; % 7 cores
time2(8) = 1.094302; % 8 cores
time2(9) = 1.385474; % 9 cores
time2(10) = 1.463371; % 10 cores
time2(11) = 1.438969; % 11 cores
time2(12) = 1.444379; % 12 cores
time2(13) = 1.791291; % 13 cores
time2(14) = 1.847930; % 14 cores
time2(15) = 1.856020; % 15 cores
time2(16) = 2.355385; % 16 cores

% Calculating speed-up, efficiency and ideal time
speedup2 = zeros(1,length(cores2));
efficiency2 = zeros(1,length(cores2));
t2 = T2*ones(1,length(cores2));
for i=1:length(cores2)
    speedup2(i) = T2/time2(i)*cores2(i);
    efficiency2(i) = speedup2(i)/cores2(i);
end

% Generating figures
figure(3)
subplot(2,1,1)
hold on
plot(cores2, time2, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores2, t2, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
legend('Experimental', 'Ideal');
title('num\_values=10^6 per process, num\_steps=100');
xlabel('Nr of processes');
ylabel('Measured time');
hold off
subplot(2,1,2)
hold on
plot(cores2,speedup2, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores2, cores2, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
legend('Experimental', 'Ideal');
xlabel('Nr of processes');
ylabel('Speed-up');
hold off

% Calculating ideal efficiency
```

```
e1 = ones(1,length(cores1));
e2 = ones(1,length(cores2));

figure(4)
subplot(2,1,1)
hold on
plot(cores1, efficiency1, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores1, e1, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
ylim([0 1.2])
legend('Experimental', 'Ideal');
title('Strong scaling');
xlabel('Nr of processes');
ylabel('Efficiency');
hold off
subplot(2,1,2)
hold on
plot(cores2,efficiency2, '-*', 'Color', [255/255 102/255 178/255], 'LineWidth', 2);
plot(cores2, e2, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
ylim([0 1.2])
legend('Experimental', 'Ideal');
title('Weak scaling');
xlabel('Nr of processes');
ylabel('Efficiency');
hold off
```