UPPSALA UNIVERSITY

COURSE NAME: PARALLEL AND DISTRIBUTED PROGRAMMING

COURSE CODE: 1TD070

One-dimensional stenscil application

Author: Alhassan JAWAD

Supervisor: Roman Iakymchuk

Date: 13-04-2024

Assignment 2

1 Problem formulation[1]

In this assignment, we are going to apply a one-dimensional stencil on an array of elements representing function values $f(x)$ for a finite set of N values $x_0, x_1, ..., x_{N-1}$ residing on the interval $0 \leq x < 2 \times \pi$ for which each value $x_i = i \times h$ where $h = \frac{2\pi}{N}$. Applying the stencil on an element v_0 representing the function value $f(x_i)$ simply means computing the sum:

$$\frac{1}{12h} \times v_{-2} - \frac{8}{12h} \times v_{-1} + 0 \times v_0 + \frac{8}{12h} \times v_{+1} - \frac{1}{12h} \times v_{+2} \tag{1.0.1}$$

where v_{-j} is the element j steps to the left of v_0 (that is, $f(x_{i-j})$). Likewise, v_{+j} is the element j steps to the right of v_0 (that is, $f(x_{i+j})$).[2] This sum approximates the first derivative $f(x)$ for $x = x_i$.

My task is to parallelize a short program that applies this stencil on a set of values.

1.1 Provided files

1.1 Provided files

The following are files were provided with the assignment description:

- A .tar compressed file containing:
  - The serial stencil code to be parallelized (***stencil.h*** & ***stencil.c***)
  - A makefile to build the program
  - A shell-script to be used be as a preliminary check before submission

1.1.1 The Serial Code

Summarizing important information about the serial code ***stencil.h***:

- First argument: Input file path/name
- Second argument: Output file path/name
- Third argument: Integer specifying how many times the stencil will be applied
- Input file must contain $N + 1$ numbers
  - First number is N (number of function values)
  - Subsequent numbers are the actual function values.
- Output file will contain stencil application (the results)

The program reads data from an input file (using the function $read\_input$), Performs a stencil operation on the data, and writes the final result to an output file (using the function $write\_output$). It measures execution time without accounting for input/output operations (File I/O) and treats edges as neighbors (periodic boundaries).

[1] Taken directly from the Assignment PDF

[2] For this case, we assume that the middlemost stencil weight is 0, which is not always the case!

- There is no need to parallelize $read\_input$ or $write\_output$.
- Due to usage of *periodic boundaries*, the last element in the array is considered as left neighbor while the first element is considered right neighbor.

1.1.2 Requirements for the parallelized code

- I/O of the program should be handled by process unit 0
  - It should evenly distribute the values among all processors (before the first stencil application starts)
  - It should collect the result when the last stencil application is completed
- Assume that the number of values $N + 1$ is divisible by the number of processors => $(N + 1)$ modulo $p = 0$ for p = Number of processor units
- Each processor should apply stencil application on its received values and then store the results in a different new array => each processor applies stencil on its values specified number of times, each new application applied on last computed individual result
- Assume that the number of function values per process is larger than the stencil width
- Note that each process will need data that is stored on its neighbors nodes (to be able to apply the stencil)!
- Each process have their own time measurement to apply the stencil, but only the largest value among all processors should be printed.
  - Include application of the stencil
  - Include inter-process communication used on each step
  - Don't include time for executing I/O operations
  - Don't include time for initial distribution and final collection of data

Your parallelization is supposed to speed up the execution of the program, but you must not change its functionality compared to the original program!

2 Parallelization

This section will give a brief description about the changes and modifications I did to transform the serial code *stencil.c* into a parallelized code that I called ***stencil_mpi.c***.

After starting the code in the same way by including *stencil.h* and the appropriate packages needed, I declared the initial variables needed in the same way as done in the serial code. But here, I declared a variable *num_values* that is responsible for the number of values in the input file, afterwards I declared a *data_chunk* variable that is responsible for assigning the number values to be processed by each process.

In the next section responsible for initializing the MPI Setup, the following listing shows what I did:

```
int rank, size;
MPI_Request request[4];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

At the stage of reading the input file, I didn't change how the *read_input* file is called as there is no need to parallelize the I/O operations. However, I used dynamic memory allocation to allocate memory space for the output in advance as it may balance memory efficiency and flexibility to different input sizes without the need to modify the code later on. The following listing shows what I did and where in the code:

```
if (0 == rank){
    if (0 > (num_values = read_input(input_name, &input)))
        ...
    // Dynamic memory allocation
    if (NULL == (output = malloc(num_values * sizeof(double)))){
        perror("Error: Couldn't allocate memory for output");
        return 2;
    }
}
```

After initializing the MPI setup and reading the file, I used ***MPI_Bcast*** to broadcast the total number of values from the root processor to all other processes, I also declared some constants and values corresponding to the (1.0.1) formula shown at page 2. The corresponding memories are located for the input and output and I used ***MPI_Scatter*** to scatter the input values randomly among all processors.

```
MPI_Bcast(&num_values, 1, MPI_INT, 0, MPI_COMM_WORLD);
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)};
data_chunk = num_values/size;
double *local_input = (double *)malloc((data_chunk+EXTENT*2) * sizeof(double));
double *local_output = (double *)malloc((data_chunk+EXTENT*2) * sizeof(double));
MPI_Scatter(input, data_chunk, MPI_DOUBLE, local_input+EXTENT, data_chunk,
    MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Following the above, I started a timer and defined the left/right neighbors of a process. I also made some checks that guarantees what to do in case a neighbor doesn't exist meaning that we go out of bounds (periodic boundaries).

```
    double start_time = MPI_Wtime();
    int left = rank - 1;
    if (left < 0) {left = size - 1;}
    int right = rank + 1;
    if (right >= size) {right = 0;}
```

I managed the passing of data between the processes by using the non-blocked version of send and receives in MPI, this means that I used ***MPI_iSend*** to send data while ***MPI_Recv*** received the data after a ***MPI_Wait*** is executed. The reason for using MPI_Wait is to avoid *deadlock* situations.

```
for (int s = 0; s < num_steps; s++){
    MPI_Isend(local_input+EXTENT, EXTENT, MPI_DOUBLE, left, 0, MPI_COMM_WORLD,
        &request[0]);
    MPI_Isend(local_input+data_chunk, EXTENT, MPI_DOUBLE, right, 1,
        MPI_COMM_WORLD, &request[2]);
    MPI_Recv(local_input+data_chunk+EXTENT, EXTENT, MPI_DOUBLE, right, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(local_input, EXTENT, MPI_DOUBLE, left, 1, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

    // Apply stencil
        ...
        ...
    MPI_Wait(&request[1], MPI_STATUS_IGNORE);
        ...
        ...
    MPI_Wait(&request[3], MPI_STATUS_IGNORE);
        ...
        ...
    // Swap input and output
        ...
        ...
}
```

Lastly, I stopped the timer, printed just the max time by using the ***MPI_Reduce()*** function, gathered the outputs from the different processors using ***MPI_Gather*** and finally deallocated/freed the memory locations saved for the outputs/inputs.

```
    ...
    MPI_Reduce(&time_taken, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Gather(local_output+EXTENT, data_chunk, MPI_DOUBLE, output, data_chunk,
        MPI_DOUBLE, 0, MPI_COMM_WORLD);
    if (rank == 0){
        ...
        ...
        #ifdef PRODUCE_OUTPUT_FILE
        if (0 != write_output(output_name, output, num_values)){
            return 2;
        }
        #endif
        free(input);
        free(output);
    }
    free(local_input);
    free(local_output);
    MPI_Finalize();
    return 0;
}
```

I didn't modify anything in the I/O operations as the time is measured without taking into account I/O operations.

The kind of communication I used was a combination of ***Non-blocking Sends and Receives*** and ***Synchronization with MPI_Wait***. Regarding the complete code, it will be attached with this report under *Appendix*.

3 Performance

Performance experiments and execution times are measured on the *snowy* cluster (Linux Virtual Machine) and the specifications of the server are noted in the following table:

Part	**Specifications**
CPU	Intel Xeon E5-2630 v4
Memory	128GB / 256GB / 1TB
Operating System	CentOS 7
Server Name	rackham.uppmax.uu.se

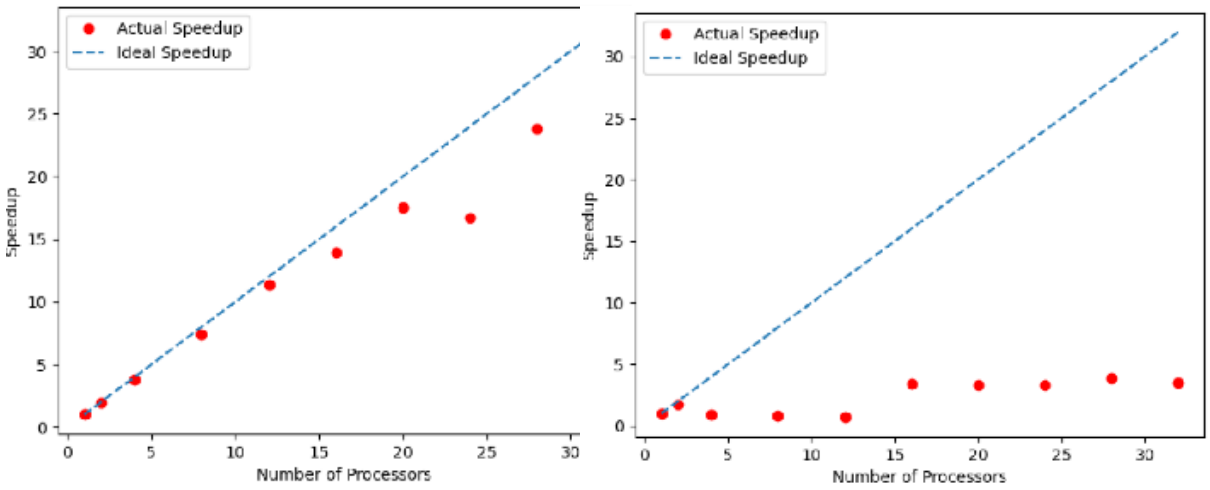
Table 3.0.1: Server Specifications used for performance experiments on the stencil application

According to the assignment instructions, I am supposed to do both a strong scaling and weak scaling performance experiments on the parallelized stencil application code.

Before doing this measurements and performance experiments, I ran the code on a given input_file with 96 number_of_values and 1/4 number_of_steps and made sure that the output_file contained the correct results according to reference files that were provided for us on the rackham.uppmax.u.se server. I had some segmentation fault errors in my code that made the job submission crash giving core.163... files created but I solved the errors by rewriting the code.

3.1 Strong Scalability

In the instructions, it was stated that for strong scalability, it is enough to experiment with one file size. So I experimented on the codes performance using the given input_file *input1000000.txt* with num_values set at 2 meaning that the problem size is 2×10^6. The following 2 figures show MATLAB generated plots for the **speedup values** vs **number of processors**:



(a) Execution time for processor unit 0 based on number of processors used

(b) Maximum execution time for the computation part of the code

Figure 3.1.1: Speedup vs Number of Processors

The following table shows the execution time for different number of processors and their speedup:

# of Processors	Speedup	Time(s)
1	1	1.037897
2	1.77	0.587432
4	0.90	1.161231
8	0.77	1.343215
12	0.69	1.515424
16	3.42	0.302311
20	3.31	0.313565
24	3.37	0.308031
28	3.87	0.2681340
32	3.52	0.2932812

Table 3.1.2: Execution times for a varying number of processors

3.2 Weak Scalability

For the weak scalability experiments, I could either vary the input size or the number of stencil applications. Thinking that maybe inputs with more number of applications will take more time, I decided to instead keep using the *input1000000.txt* file and instead vary the size of the problem by changing the *num_values* variable. The following 2 figures show the weak scalability speedup and efficiency, respectively.

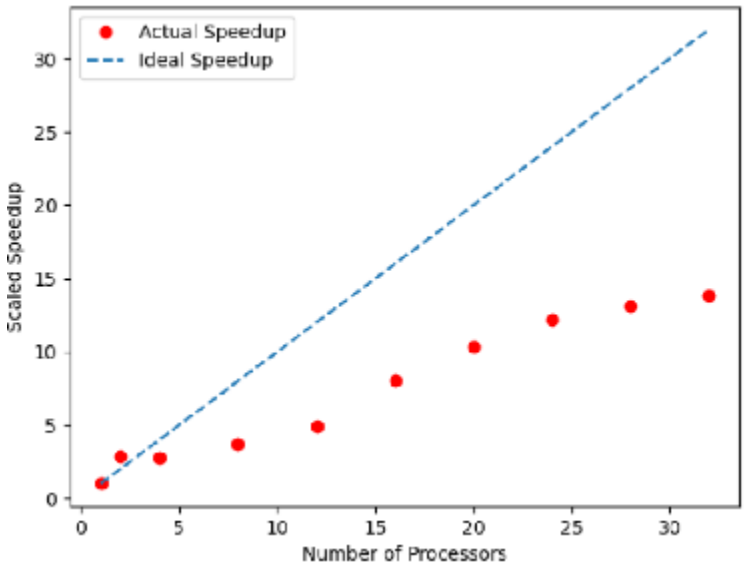


Figure 3.2.2: Weak Scalability speedup vs number of processors

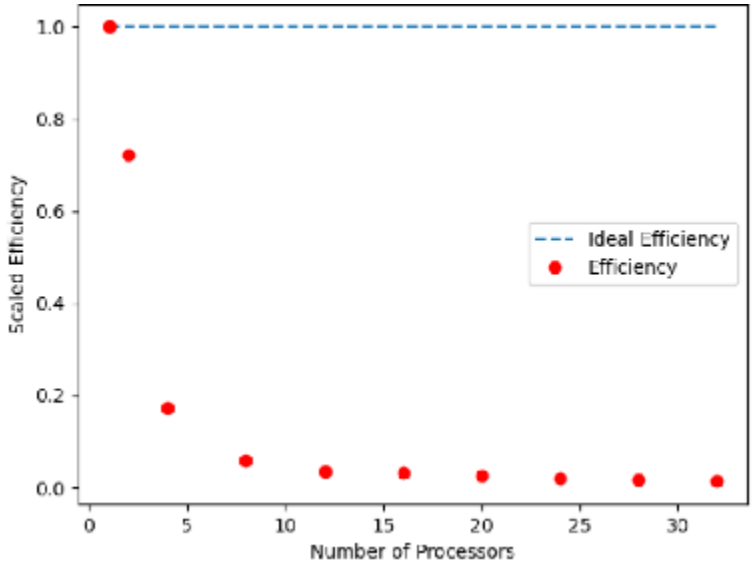


Figure 3.2.3: Efficiency of the code for weak scalability. The efficiency is calculated as $\frac{\text{The Actual Speedup}}{\text{The Ideal Speedup}}$

Beside the above figures, I have a table that shows the time and speedup gotten for varying number of processors/size of problem:

# of Processors	Size of Problem ($x10^6$	Speedup	Time(s)
1	1	1.00	0.776858
2	2	2.89	0.538182
4	4	2.76	1.126565
8	8	3.72	1.672383
12	12	4.87	1.915068
16	16	8.03	1.548688
20	20	10.39	1.495444
24	24	12.19	1.529725
28	28	13.15	1.654181
32	32	13.83	1.796925

4 Discussion & Conclusion

Strong Scalability

Regarding the strong scaling performance, the experiments showed results that aren't very good where the speedup doesn't increment proportionally with respect to the number of processors. The explanation could be that the communication overhead going on in the program might turn into a bottleneck as the number of processes.

Worth noting is that Amdahl's law can be to make sense of some part of this issue.

$$Speedup = \frac{1}{s + \frac{p}{N}}$$

where s is the extent of execution time spent on the sequential part of the code, p is the extent of execution time spent on the parallel part of the code and N is the number of processors.

On the other hand, the left subplot of figure *3.1.1* shows us the execution time (speedup vs number of processors) for processor unit 0 which proves the performance improvements when increasing the number of processors.

Honestly speaking, I thought that these results would have surfaced if I used blocking communication functions like *MPI_Send/MPI_Rec* or *MPI_Sendrecv*,as it felt that blocking communication have a higher chance of getting the results I got. This I thought was a little weird but didn't really find an explanation for why it happened, so to find the reason for this phenomena, further work should be done.

Weak Scalability

The results of the weak scalability, for the most part, live up to my expectations and assumptions. The program's performance and speedup increases with the increase of the number of processors used, but this performance increase is nowhere near the ideal scaled speedup. Here Gustafson's law can be introduced to make sense of this peculiarity.

$$\text{Scaled Speedup} = s + p \times N$$

where N is the number of processes, p is the proportion of execution time spent on the parallel part of the code, and s is the proportion of execution time spent on the serial part pf the code.

With Gustafson's law the scaled speedup increases straightly concerning the number of processors (with an incline smaller than one), and there is no cutoff (upper limit) for the scaled speedup. The plot of weak scalability efficiency also shows how efficiency of the program decreases when using more processors.

Appendix

Parallelized code *stencil_mpi.c*

```
/*
@desc
 This script is the parallelized version of the serial code stencil.c
 script that applies a 5-point stencil application.
@author
 Alhassan Jawad
@since
 13-04-2024
@version
 1.1.5
*/
#include "stencil.h"

int main(int argc, char **argv) {
//---------------------------------------------------------------------
// Check if the number of arguments is correct
   if (4 != argc) {
      printf("Usage: stencil input_file output_file number_of_steps\n");
      return 1;
   }
//---------------------------------------------------------------------
// Declare initial variables
   char *input_name = argv[1];
   char *output_name = argv[2];
   int num_steps = atoi(argv[3]);
   int num_values; // Total number of values
    int data_chunk; // Values for each processor
//---------------------------------------------------------------------
// Initilize MPI setup
    int rank, size;
   MPI_Request request[4];
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Read the input file
   double *input = 0;
   double *output = 0;

   if (rank == 0){
      if (0 > (num_values = read_input(input_name, &input))) {
         return 2;
      }
        // Dynamic memory allocation
      if (NULL == (output = malloc(num_values * sizeof(double)))) {
         perror("Couldn't allocate enough memory for output");
         return 2;
      }
   }
//---------------------------------------------------------------------
```

```
// Stencil values
   MPI_Bcast(&num_values, 1, MPI_INT, 0, MPI_COMM_WORLD);
   double h = 2.0*PI/num_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)};

   data_chunk = num_values / size;
   double *local_input = (double *)malloc((data_chunk + 2*EXTENT) * sizeof(double));
   double *local_output = (double *)malloc((data_chunk + 2*EXTENT) *
       sizeof(double));
   MPI_Scatter(input, data_chunk, MPI_DOUBLE, local_input+EXTENT, data_chunk,
       MPI_DOUBLE, 0, MPI_COMM_WORLD);
//---------------------------------------------------------------------
// Start timer & define neighbors
   double start_time = MPI_Wtime();
   int left = rank - 1;
   if (left < 0) {left = size - 1;}
   int right = rank + 1;
   if (right > size -1) {right = 0;}
//---------------------------------------------------------------------
```

```
// Repeatedly apply stencil
   for (int s = 0; s < num_steps; s++) {
        // Send data to left/right neighbor
        // Receive data from right/left neighbor
      MPI_Isend(local_input+EXTENT, EXTENT, MPI_DOUBLE, left, 000, MPI_COMM_WORLD,
          &request[0]);
      MPI_Irecv(local_input+data_chunk+EXTENT, EXTENT, MPI_DOUBLE, right, 000,
          MPI_COMM_WORLD, &request[1]);
      MPI_Isend(local_input+data_chunk, EXTENT, MPI_DOUBLE, right, 111,
          MPI_COMM_WORLD, &request[2]);
      MPI_Irecv(local_input, EXTENT, MPI_DOUBLE, left, 111, MPI_COMM_WORLD,
          &request[3]);

        // Apply stencil
      for (int i=2*EXTENT; i<data_chunk; 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] = result;
      }
        // Wait for communication to complete
      MPI_Wait(&request[1], MPI_STATUS_IGNORE);
      for (int i = data_chunk; i < data_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] = result;
      }

      MPI_Wait(&request[3], MPI_STATUS_IGNORE);
      for (int i = EXTENT; i < 2*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] = result;
      }
        // Swap input and output, locally of course
      if (s < num_steps-1) {
         double *tmp = local_input;
         local_input = local_output;
         local_output = tmp;
      }
   }
//---------------------------------------------------------------------
```

```
// Stop timer & end MPI
   double time_taken = MPI_Wtime() - start_time;
   double max_time;
   MPI_Reduce(&time_taken, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

   MPI_Gather(local_output+EXTENT, data_chunk, MPI_DOUBLE, output, data_chunk,
       MPI_DOUBLE, 0, MPI_COMM_WORLD);

   if (rank==0){
      printf("%f\n", max_time);
        // Write the output to the output file
      #ifdef PRODUCE_OUTPUT_FILE
      if (0 != write_output(output_name, output, num_values)) {
         return 2;
      }
      #endif
      free(input);
      free(output);
   }

   free(local_input);
   free(local_output);
   MPI_Finalize();
   return 0;
}

//---------------------------------------------------------------------
```

```c
// Changed nothing below this line
int read_input(const char *file_name, double **values) {
   FILE *file;
   if (NULL == (file = fopen(file_name, "r"))) {
      perror("Couldn't open input file");
      return -1;
   }
   int num_values;
   if (EOF == fscanf(file, "%d", &num_values)) {
      perror("Couldn't read element count from input file");
      return -1;
   }
   if (NULL == (*values = malloc(num_values * sizeof(double)))) {
      perror("Couldn't allocate memory for input");
      return -1;
   }
   for (int i=0; i<num_values; i++) {
      if (EOF == fscanf(file, "%lf", &((*values)[i]))) {
         perror("Couldn't read elements from input file");
         return -1;
      }
   }
   if (0 != fclose(file)){
      perror("Warning: couldn't close input file");
   }
   return num_values;
}


int write_output(char *file_name, const double *output, int num_values) {
   FILE *file;
   if (NULL == (file = fopen(file_name, "w"))) {
      perror("Couldn't open output file");
      return -1;
   }
   for (int i = 0; i < num_values; i++) {
      if (0 > fprintf(file, "%.4f ", output[i])) {
         perror("Couldn't write to output file");
      }
   }
   if (0 > fprintf(file, "\n")) {
      perror("Couldn't write to output file");
   }
   if (0 != fclose(file)) {
      perror("Warning: couldn't close output file");
   }
   return 0;
}
```