UPPSALA
UNIVERSITET

Parallel and Distributed Programming

Assignment 1
One-dimensional Stencil Application

Group 16
Jinglin Gao, Ruihang Cao, Aoping Lyu

May 14, 2023

1 Introduction

In this assignment, a one-dimensinal stencil is applied on an array of elements representing function values $f(x)$ for a finite set of N values $x_0, x_1, x_2, ..., x_{N-1}$, residing on the interval $0 <= x < 2 \cdot \pi$. Here, each value $f(x_i)$ simply means computing the sum;

$$\frac{1}{12h} \cdot v_{-2} - \frac{8}{12h} \cdot v_{-1} + 0 \cdot v_0 + \frac{8}{12h} \cdot v_{+1} - \frac{1}{12h} \cdot v_{+2}$$

where v_{-j} is the element j steps to the left of v_0 (that is, $f(x_{i-k})$. Likewise, v_{+j} is the element j steps to the right v_0 ($f(x_{i+j})$). This sum approximates the first derivative $f'(x)$ for $x = x_i$. The given codes of 1D stencil problem will be parallized using MPI in this Assginment.

2 Parallelization

As the requirements, the I/O of the program should be handled by the rank 0 process; and this process should distribute the values evenly among all processes before the first stencil application and collect the result when the last application is done. After the initial distribution, each process should apply the stencil on its received values and store the result in another array. Thereafter, each process repeatedly applies the stencil on the result of its last application until the specified number of applications are made.

First, the number of values, and the chunk are introduced. `number_values` represents the total number of values need to be handled, and `chunk` represents the size of the data chunk that each process will work on. The data is divided into equal-sized chunks to distribute the workload among the available processes. Then, the MPI is initialised, which is shown in Listing 1.

Listing 1: MPI setup

```
1  int num_values, chunk;
2
3  int rank, size;
4  MPI_Status status;
5  MPI_Init(&argc, &argv); /* Initialize MPI */
6    MPI_Comm_size(MPI_COMM_WORLD, &size); /* Get the number of processes */
7    MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* Get my number         */
```

In the stage of reading file, the rank 0 process is used. The memory for output data is also allocated at the beginning in advance using dynamic memory allocation. The codes are shown in Listing 2.

Listing 2: Read input file

```
1  double *input = 0;
2  double *output = 0;
3  if (rank==0){
4    // Read input file
5    if (0 > (num_values = read_input(input_name, &input))) {
6      return 2;
7    }
8    // Allocate data for result
9    if (NULL == (output = malloc(num_values * sizeof(double)))) {
10     perror("Couldn't allocate memory for output");
11     return 2;
12   }
13 }
```

Then, `MPI_Bcast` is used to broadcast the `number_values` to all processes. The corresponding memories are allocated for local input and output. Note here, The input data is scattered among the processes using `MPI_Scatter`, which is the inverse operation of `MPI_Gather`, it sends split data chunk to each process. In the final stage, the results from each process will be gathered back to the root process using `MPI_Gather`.

Listing 3: Stencil values

```
// Stencil values
MPI_Bcast(&num_values, 1, MPI_INT, 0, MPI_COMM_WORLD);

      /* ...Same with serial version...*/

chunk = num_values / size;
double *local_input = (double *)malloc((chunk + 2*EXTENT) * sizeof(double));
double *local_output = (double *)malloc((chunk + 2*EXTENT) * sizeof(double));
MPI_Scatter(input, chunk, MPI_DOUBLE, local_input+EXTENT, chunk, MPI_DOUBLE, 0,
     MPI_COMM_WORLD);
```

Left and right neighbours are defined as the process need to exchange data between neighboring processes then do the stencil calculations.

Listing 4: Define left and right neighbours

```
      // Define left and right neighbors
int left = rank - 1;
if (left < 0){left = size-1;}
int right = rank + 1;
if (right > size -1){right=0;}
```

The data passing among the process and its neighbours is managed by using the combined send and receive function `MPI_Sendrecv`. For this case, the `STENCIL_WIDTH` is 5, and the integer `EXTENT` is 2. The first `MPI_Sendrecv` operation sends the `EXTENT` data from the current process (rank) to the left neighboring process (left) and receives the `EXTENT` data from the right neighboring process (right), and the second `MPI_Sendrecv` operation did it in the opposite way. Buffers also used here for data exchange between processes. After the communication, the computation operations starts, here two `for` loops are used for computation.

Listing 5: Apply stensil

```
// Repeatedly apply stencil
  for (int s=0; s<num_steps; s++) {
    // first pass data to the left then to the right
    MPI_Sendrecv(local_input+EXTENT, EXTENT, MPI_DOUBLE, left, 000, local_input+chunk+EXTENT,
         EXTENT, MPI_DOUBLE, right, 000, MPI_COMM_WORLD, &status);
    MPI_Sendrecv(local_input+chunk, EXTENT, MPI_DOUBLE, right, 111, local_input, EXTENT,
         MPI_DOUBLE, left, 111, MPI_COMM_WORLD, &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] = result;
    }
```

Here, the communication pattern used in this code is point-to-point communication using `MPI_Sendrecv`. `MPI_Sendrecv` combines both send and receive operations in a single function call, allowing for simultaneous sending and receiving of data between neighboring processes. The last stage is to swap the input and output, and then gather the results from each process and write the output, finally free the used memory.

Listing 6: I/O swap and gather results

```
// Swap local_input and local_output
    if (s < num_steps-1) {
      double *tmp = local_input;
      local_input = local_output;
      local_output = tmp;
    }
  }

  // Stop timer
  double my_execution_time = MPI_Wtime() - start;
  double maxtime;
```

```
12  MPI_Reduce( &my_execution_time, &maxtime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
13  MPI_Gather(local_output+EXTENT, chunk, MPI_DOUBLE, output, chunk, MPI_DOUBLE, 0,
        MPI_COMM_WORLD);
14
15  if (rank==0){
16    printf("%f\n", maxtime);
17    #ifdef PRODUCE_OUTPUT_FILE
18    if (0 != write_output(output_name, output, num_values)) {
19      return 2;
20    }
21    #endif
22
23    // Clean up
24    free(input);
25    free(output);
26  }
27
28  free(local_input);
29  free(local_output);
30  MPI_Finalize(); /* Shut down and clean up MPI */
```

3 Performace experiments

Configuration

The execution times are measured with the Linux virtual machine, and its specification is listed in Table 1 below.

Table 1: Server Specifications

Component	**Specification**
CPU	AMD Opteron (Bulldozer) 6282SE, 2.6 GHz, 16-cores, dual socket
Memory	128 GB
Operating System	Ubuntu 22.04
Compiler Version	gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0
Server Name	vitsippa.it.uu.se

Strong Scalability

In the strong scaling performance experiments, two execution times were measured, one is execution time of p0, and the max execution time for the computation part. The experiments are managed with the Linux machine stated above using `mpirun --bind-to none -np X stencil /home/maya/public/PDP_Assignment1/input100000000.txt result.txt 2`. Here the input file is `input100000000.txt` and the `num_values` is 2, so the problem size is 2×10^8. The graph of strong scaling speedup for the root process (p0) is shown in Figure 1.

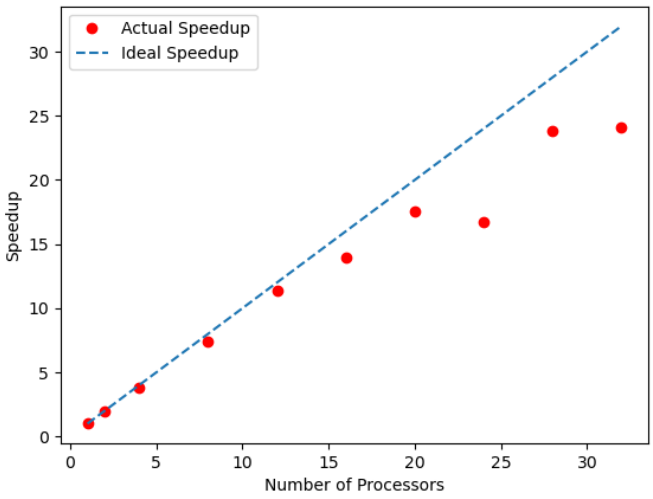

Figure 1: Strong Scaling Speedup for `p0`

The max execution time for the computation part is also measured. The speedup graph and the execution time for different number of processes used are shown in Figure 2 and Table 2.

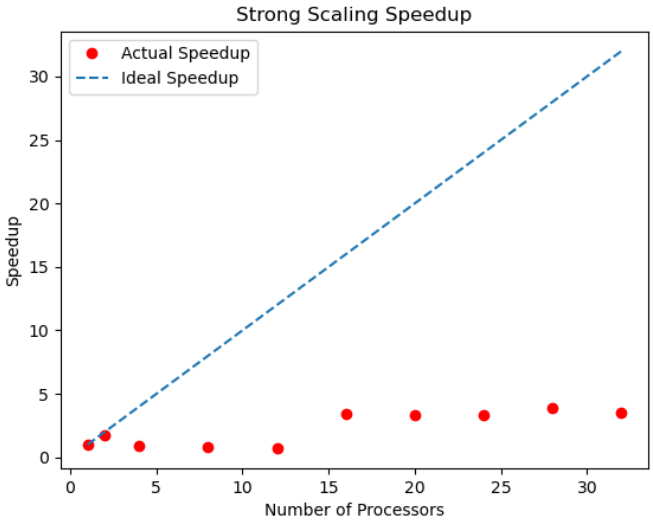

Figure 2: Strong Scaling Speedup

Table 2: Execution time

Number of processes	Time(s)	Speedup
1	1.037985	1
2	0.587072	1.77
4	1.161410	0.90
8	1.345415	0.77
12	1.514724	0.69
16	0.303111	3.42
20	0.313265	3.31
24	0.308031	3.37
28	0.268239	3.87
32	0.294812	3.52

Weak Scalability

In the weak scaling performance experiments, the execution time is measured for corresponding number processes and size of problem. The experiments are managed with the Linux machine stated above using `mpirun --bind-to none -np X stencil /home/maya/public/PDP_Assignment1/input100000000.txt result.txt X`. Here the same input file is used and the number of processes is set with respect of `num_values` (i.e. the size of problem). The scaled speedup graph and the execution time for different number of processes and size of problem are shown in Figure 3 and Table 3. The efficiency is also calculated by $\frac{Actual\_speedup}{Ideal\_speedup}$ and plotted in Figure 4.

Table 3: Execution time

Number of processes	Size of Problem ($\times 10^8$)	Time(s)	Speedup
1	1	0.776858	1.00
2	2	0.538182	2.89
4	4	1.126565	2.76
8	8	1.672383	3.72
12	12	1.915068	4.87
16	16	1.548688	8.03
20	20	1.495444	10.39
24	24	1.529725	12.19
28	28	1.654181	13.15
32	32	1.796925	13.83

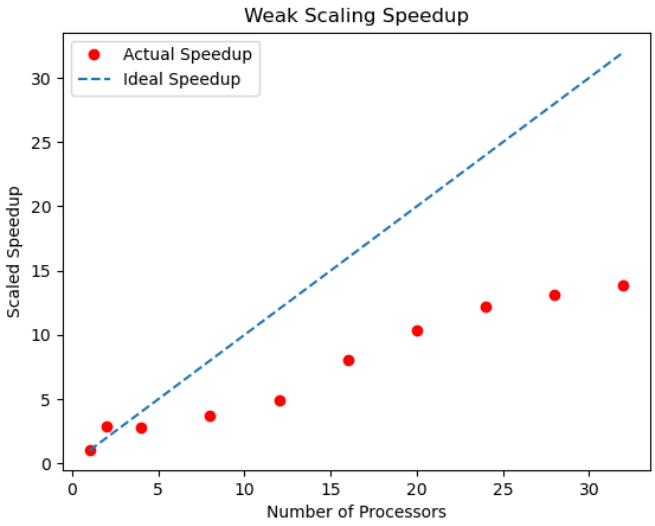

Figure 3: Weak Scaling Speedup

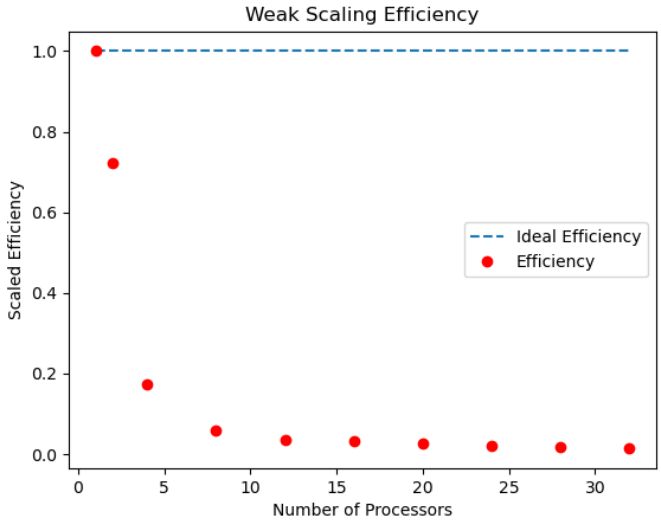

Figure 4: Weak Scaling Efficiency

4 Discussion

The communication is done by using `MPI_Sendrecv` in our solution. The sending function `MPI_Ssend` was tried at the beginning, however, combined communication function has lower efficiency. Consequently, the blocking communication `MPI_Sendrecv` is applied to speedup the communications. This type of communication simplifies the code and reduces the chances of deadlocks, which can occur when using separate send and receive calls.

The results of strong scaling experiments are not ideal, the speedup doesn't increase gradually with respect of the number of processes. The reason could be that the communication overhead of the program may become a bottleneck as the number of processes increases. Also, the Amdahl's law can be introduced to explain this kind of problem.

$$Speedup = 1/(s + p/N)$$

where s is the proportion of execution time spent on the serial part, p is the proportion of execution time spent on the part that can be parallelized, and N is the number of processes. However, the speedup for the root process generally fit the expectations of the strong scaling, which proves the performance improvement when more processes took part in the execution.

The results of weak experiments generally meet the expectations. More processes speedup the program, but the speedup is further away from the ideal scaled speedup. Here Gustafson's law can be introduced to explain this phenomenon.

$$Scaled\ speedup = s + p \times N$$

where s is the proportion of execution time spent on the serial part, p is the proportion of execution time spent on the part that can be parallelized, and N is the number of processes. With Gustafson's law the scaled speedup increases linearly with respect to the number of processes (with a slope smaller than one), and there is no upper limit for the scaled speedup. The graph of weak scaling efficiency also shows the efficiency gradually decreases when using more processes.

Except for the strong and weak scalability evaluation, we also have a evaluation on the execution time for different processes. We notice that the execution time on some specific process always significantly larger than other processes, which has strongly affected the measurement of the max execution time. Theoretically, the workload of each process is distributed rough evenly, it will not be such execution time difference as we measured. To find the reason for this phenomenon, further work should be done afterwards.

5 Corrections

Upon further research of the issue, we found that a process does not necessitate the values from its neighbors to apply the stencil to its 'center' values. More specifically, for the array we have defined, which has a length of `chunk+2EXTENT`, the values at indices from `2EXTENT` to `chunk-1` do not require data from neighbouring processes to complete the stencil computation. Therefore, we employ non-blocking communication methods, `MPI_Isend` and `MPI_Irecv`, to allow for simultaneous communication and computation for these specific indices. Then, we utilize `MPI_Wait` to ensure communication completion before applying the stencil to the 'boundary' values. The corrected codes for communication and computation part are shown below:

Listing 7: Communication and computation

```
// Repeatedly apply stencil
for (int s=0; s<num_steps; s++) {
  // first pass data to the left then to the right
  MPI_Isend(local_input+EXTENT, EXTENT, MPI_DOUBLE, left, 000, MPI_COMM_WORLD, &request[0]);
  MPI_Irecv(local_input+chunk+EXTENT, EXTENT, MPI_DOUBLE, right, 000, MPI_COMM_WORLD, &request
      [1]);
  MPI_Isend(local_input+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<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;
  }

  MPI_Wait(&request[1], MPI_STATUS_IGNORE);
  for (int i=chunk; 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] = 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;
  }
```

6 Reference

Xi, L. (2018). Scalability – Strong and Weak Scaling. PDC Blog. KTH Royal Institute of Technology. https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/ [Accessed on 6 April 2023]