UPPSALA
UNIVERSITET

Assignment 1

Parallel and distributed programming 5 credits 1TD070 62007

Group members:

Ibrohim Hamoud ibrohimmn1994@gmail.com

Uppsala

March 19, 2022

1 Introduction and Problem description

This work deals with parallelizing an algorithm for one-dimensional-5th order stencil application. The application is commonly used as a numerical method in finding a solution for partial differential equations. The stencil computation is a class of algorithms that update elements in a multidimensional grid based on neighboring values using a fixed pattern, and it functions as a kernel in digital image processing. The stencil can be thought of as a tensor of wights (coefficients) that sweeps through a larger tensor of the same dimension. Thus, updating the tensor's elements by the weighted sum of the neighbors.

The parallelization task involves dividing a one-dimensional tensor among multiple processors and letting each processor perform stencil computation on its share of the tensor. The computation is applied repeatedly in a parallel for a certain number of iterations. This algorithm parallelization is expected to speed up the computation for an increasing number of processors as the workload per processor is subsequently decreasing.

2 Algorithm parallelization

The problem is organized around a central data structure that is a one-dimensional tensor for the problem at hand. This input vector can be decomposed into smaller segments (chunks) that can be updated concurrently, after distributing the chunks among the processors. The computation breaks down into three components. First, exchange boundary data, where a 5th order stencil computation means that each chunk requires accessing the first and last two elements in the adjacent chunk that resides in the neighboring processor. Second, update the interiors or each chunk. Third, update the boundary regions.

The communication of boundary data is achieved by extending every chunk with an addition of two phantom elements(ghost cells) at the start and end of the chunk. Thus, at the start of each iteration, an MPI- communication [1] is performed where these ghost cells store the ghost cells of the adjacent chunks. In such a manner, both the second and third computational components are merged together, and the stencil will not run off at the end of the chunk.

The parallelized algorithm is shown in Algorithm 1. The algorithm makes use of mpi-vectors to facilitate communication between the processors. Distributing input data and collecting output-data from and to root processor, respectively, is achieved with MPI gather/scatter functions. In each computational stencil iteration, the ghost cells are communicated asynchronously with Irecv/Isend. Thus, avoiding deadlock that might occur when using Send/Recv and employing large number of processors.

```
Algorithm 1:
          <<  Parallelized algorithm  >>
          ------------------------------
 1- Root processor extract the array form the file

    *communicate the number of elements
        MPI_Bast(number of element)

    *define: len = number of elment/number of processors

2- define mpi vectors that are used to communicate vector's chunks
        MPI_Type_vector(len, 1, 1, MPI_DOUBLE, &block)

    * Distribute vector's chunks of length len from root processor to-
      all other processors using the mpi-vector "block"
        MPI_Scatter( )
```

```
3- Start Loop s = 0:N
    * Communicate the end ghost cells between adjacent processors using
        MPI_Irecv()
        MPI_Isend()

    * Apply stencil on the inner field.

    * MPI_Waitall() to ensure the communication is completed.

    * Apply stencil on the boundary

    * swap <output, input> and repeat

end loop

4- Collect the chunks into a global output
    MPI_Gather()
```

3 Performance

First, the strong scalability was checked for an input vector of 10^8 elements with 4-time steps. Keeping the number of the input size constant, the number of processes gradually increased from 1 to 32, and the respective execution time was recorded. Here, the time measurement is set to be the timely execution of the slowest processor. Thereafter the speedup ads a function of the number of processors is plotted (Figure 1), with the data results shown in the table 1. Here, the speedup is time on 1 process divided by time on p processes.

For the weak scalability, the size of the problem was varied by increasing the size of the input vector, and with equal proportion, the number of processors was increased. Plot 2 shows the execution time as a function of the scale of the problem, and a table of data results is attached to it.

4 Results and discussion

The experimental results in the table 1 exhibit strong scaling with the number of MPI processes used to run the simulation. Here, the problem size is fixed to an array of size 10^8 for 4 iterations and the number of nodes increases. The experiment is conducted on Snowy cluster at Uppmax [2]. The simulation requires for each iteration two copy steps of the ghost cells from and to the buffers that are used for the data exchange. However, the overhead of copying is negligible. These ghost cells are not used during computation of the inner domain. Therefore, the inner field is updated asynchronously with respect to the exchange of ghost cells whereas The boundary of the domain computed after communicating the ghost cells. This overlap in communication and computation implementation helps in improving the time performance of the simulation as the some local work is done while the communication is proceeding in the background.

Plot 2 shows a linear increase in time execution with the size of the problem. Here, the array size is increased by two along with the number of processes, starting from array of size 10^6 up to 16×10^6. Although the number of resources are scaled up proportionally to problem size, the experimental results show a linear increase in time execution with the size of the problem. This proves that if a problem only requires a small amount of resources, it is not beneficial to use a large amount of resources to carry out the computation.

Table 1: Time measurements for different number of processors for an input vector of size 10^8 elements and 4 iterations.

Number of processors	Time
1	1.105886
2	0.570209
4	0.279804
8	0.180967
16	0.139088
32	0.070764
64	0.036651

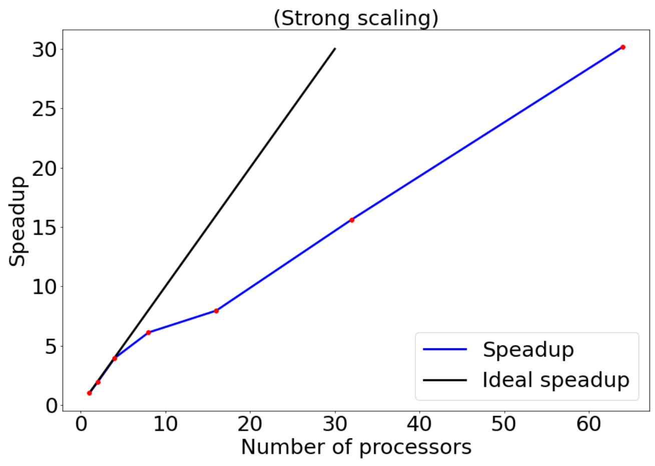

Figure 1: Plot of the speedup as a function of the number of the processors in blue, and the ideal speedup for the line in black.

Table 2: Measurements of the execution time for different problem size, with the number of processes scaled proportionally. Each simulation in the table is run for 100 iterations.

Number of element	Number of processors	Time
10^6	1	0.157955
2×10^6	2	0.182670
4×10^6	4	0.263052
8×10^6	8	0.300766
16×10^6	16	0.530214

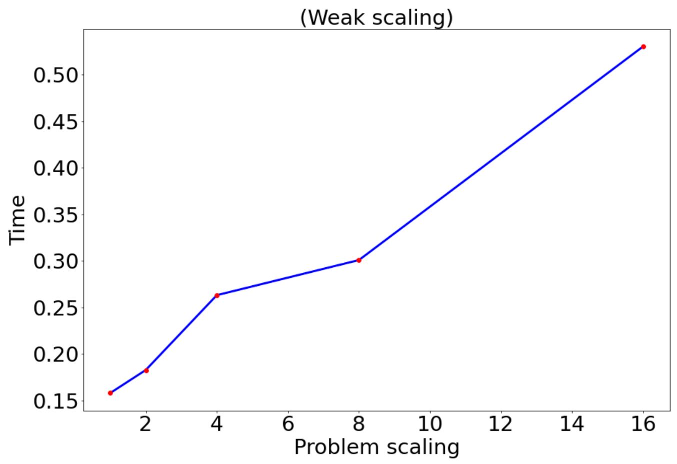

Figure 2: Plot of the execution time as a function of the problem scale.

References

[1] *Open MPI-documentation.* URL: https://www.open-mpi.org/doc/ (cit. on p. 1).

[2] *Snowy User Guide.* URL: https://www.uppmax.uu.se/support/user-guides/snowy-user-guide/ (cit. on p. 2).