# ASSIGNMENT 1:
1-D stencil application

Parallel and Distributed programming, 10 hp

Erik Bjerned, Oskar Holmes & Mattias Persson

2023-03-03

UPPSALA
UNIVERSITET

## Contents

## 1 Introduction

## 2 Parallelization

This task can be divided into two parts. The first is initially partitioning the one dimensional data into a partition for each PE. As this is simply done in splitting the array into equal sized segments, `MPI_Scatter` can be used for the initial distribution of data. Ghost rows, or elements are introduced into each segment, this is to handle the edge cases where a processor needs to access the data in another partition. To avoid the need for synchronous communication between the processes ghost elements are used. That is, extra data is loaded for reference but is not modified. Since the stencil application should have edge wrapping, the ring communication pattern is the best alternative and this is implemented with simple `MPI_Send` and `MPI_Recv` commands. This inter-processor communication has, however, to occur at every application of the stencil reducing the parallel performance when applying the stencil many times.

With the data transfer handled, the stencil is applied with almost the same code as given in the assignment. There are some modifications done as the "input" data is no longer a single array, but instead consists of the actual data and four ghost elements, which introduces some `if`-statements. This could be avoided by assembling the ghost elements and the data into a single array, however this would require more memory to be used, which for large arrays could become problematic, and could introduce latency into the calculation.

To find the processor with the longest execution time, the `MPI_Reduce` command is used together with the `MPI_MAX` operator. This gathers all individual execution times from every processor and only returns the largest, which keeps this part nice and simple.

## 3 Method for Performance tests

The performance experiments applied to the code were set up to see the speedup for the main computational part, aswell as to see how much time was used on the transfer of information between the cores. This is also compared to the total time used for the whole program(except the read/write functions).

There are two different of performance tests applied, to test both strong and weak scalability. Strong scalability is tested by increasing the amount of processors while using the largest of the input files, where $N = 10^8$ and only applying the stencil once. The weak scalability is tested by holding the workload for each processor constant, while still increasing the problem size. This is done by increasing input size in increments of $1, 2, 4, 8$ and increasing the amount of processors likewise.

## 4 Performance Test Results

### 4.1 Strong scaling

Table 1: Execution time of computation and transfer time while keeping problem size fixed, and increasing the amount of processors

| Processor amount | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| Execution time (s) | 0.69 | 0.61 | 0.183 | 0.0893 | 0.047 | 0.0302 |
| Transfer time | NaN | 0.0389 | 0.0462 | 0.0553 | 0.0711 | 0.0938 |

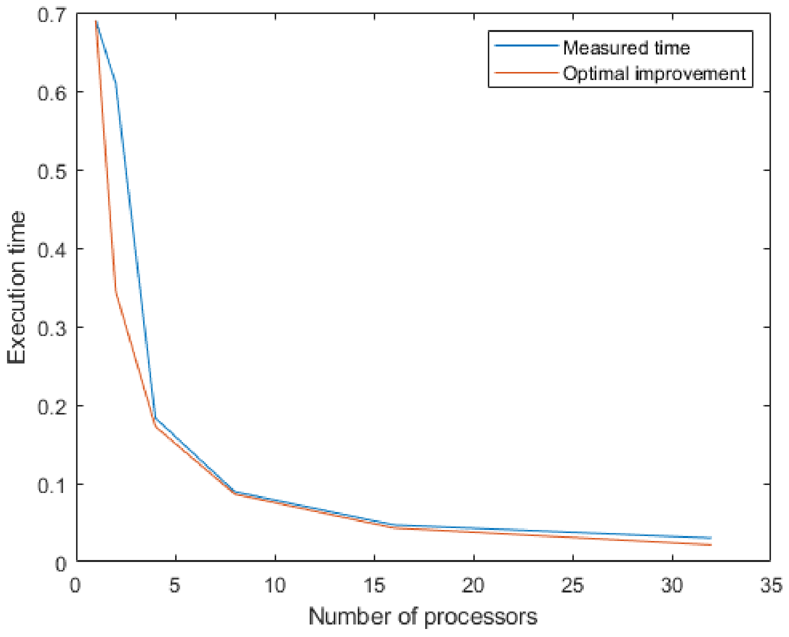

Figure 1: Caption

### 4.2 Weak scaling

Table 2: Total execution time while increasing problem size, yet making sure that workload per processor is constant

| Processor amount | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Problem size | $10^6$ | $2 * 10^6$ | $4 * 10^6$ | $8 * 10^6$ |
| Total execution time(except read/write)(s) | 0.122 | 0.1559 | 0.204 | 0.34 |

## 5 Discussion

Regarding the strong scaling of the implementation, we see that the execution time of the main computational part is closely correlated to the optimal time, where the optimal time is a $1/x$ curve. The transfer time between each processor increases when increasing the amount of processors, this is probably because the communication buffers are more occupied. This becomes certainly visible when a lot of processors are used, since the amount of raw computation gets much smaller and the communication buffers are used more frequently. Regarding the weak scaling, keeping workload constant and increasing need for communication clearly makes the total time for execution longer, as expected.