Parallel and Distributed Programming – Assignment 1 Group 15

Claude Carlsson
Edvin Germer
Ture Hassler

April 13, 2023

Contents

1 Introduction

For this assignment, the goal is to parallelize a provided c program that does a stencil computation in serial. In this context, a stencil is a pattern that performs operations on an array of data, and it is composed of two components: the shape of the stencil and weights used in the computations.

In the performance and results section of the report, we are going to present two different scaling approaches: strong scaling and weak scaling. The difference between these two scaling concepts is that for strong scaling, we hold the problem size constant while increasing the number of processes. We measure the stencil time, and the goal for strong scaling is to achieve a speedup. For weak scaling, the problem size is increased proportionally with the number of processes. The stencil time is measured, and the goal for weak scaling is to maintain a constant time.

UPPSALA UNIVERSITET

1.1 Communication type

The communication type used in this assignment is going to be the message passing interface, or MPI, for short. MPI is a library that provides functions that enable communication between processes.

For this assignment, the reasons for using MPI are twofold. First is to learn how to convert a serial implementation of a program to parallel using MPI. Second, is that MPI is designed to be used in a distributed environment, which enables scalability across a cluster of machines. Furthermore, another parallelization technique such as using OpenMP might offer an easier implementation, however, it is limited to hardware that has shared memory (that is, a single machine), and not distributed memory.

2 Parallelization

We used MPI for our parallelization. We read the data with process 0 and then evenly distributed chunks of the array to each thread using **MPI_Scatter**. We also sent the total number of elements to each process using **MPI_Bcast**.

For each iteration the stencil was applied, each process needs to send the four outmost elements (two on each side) to their respective neighbours. We did this with non blocking point to point communication, using **MPI_Isend** and **MPI_Irecv**. This allows each process to send their message and then continue applying the stencil on the middle elements that do not depend on the other processes, while waiting to receive the values from the other processes.

After each process has performed all computations that are independent of their neighbours, we then use **MPI_Waitall** to synchronize all processes, making sure that each process has received the neighbouring values so they can apply the stencil on the edge elements.

After all processes has finished their computations we then use **MPI_Gather** to send back the data to process 0, and then process 0 continues to write the output to a file.

3 Performance experiments and results

We performed several performance measurements, both for weak and strong scaling. We also included the system specifications for reproducibility of the results.

3.1 System specifications

System: UU Linux Host "Tussilago"
Processor: AMD Opteron(tm) Processor 6282 SE
Operating System: Ubuntu 22.04 x86_64
Compiler version: gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0
MPI version: mpirun (Open MPI) 4.1.2

3.2 Performance experiments

To get an accurate measurement of the parallel performance, we measured the time after reading and distributing the data among all processes, but before writing the output to a file. We then recorded the time as the longest time out of all the processes individual time measurements.

We performed the strong scalability study by measuring the time for varying number of threads, while keeping the size of the array fixed. From this, we then calculated the speedup. This was done for two different problem sizes, using 1 and 8 million elements in the arrays. This was done with 1 and 100 stencil applications.

We performed a weak scalability study by keeping the work per process constant at 1 million elements, and increasing both the number of processes and the number of elements in the array, from 1 to 8 processes and from 1 to 8 million elements. Here we also test with first 1 stencil application and then 100 stencil applications.

3.3 Results

The results of the time measurements of the scalability study can be seen here, including both plots and tables.

3.3.1 Strong scaling, stencil applied 1 time

Table 1: The experiment values from strong scaling when only one stencil is applied, 1M corresponds to one million data points, and the results is in milliseconds

Input size Processes	1M	8M
1	7.7	61.1
2	4.1	32.3
4	4.0	33.5
8	1.1	96.0

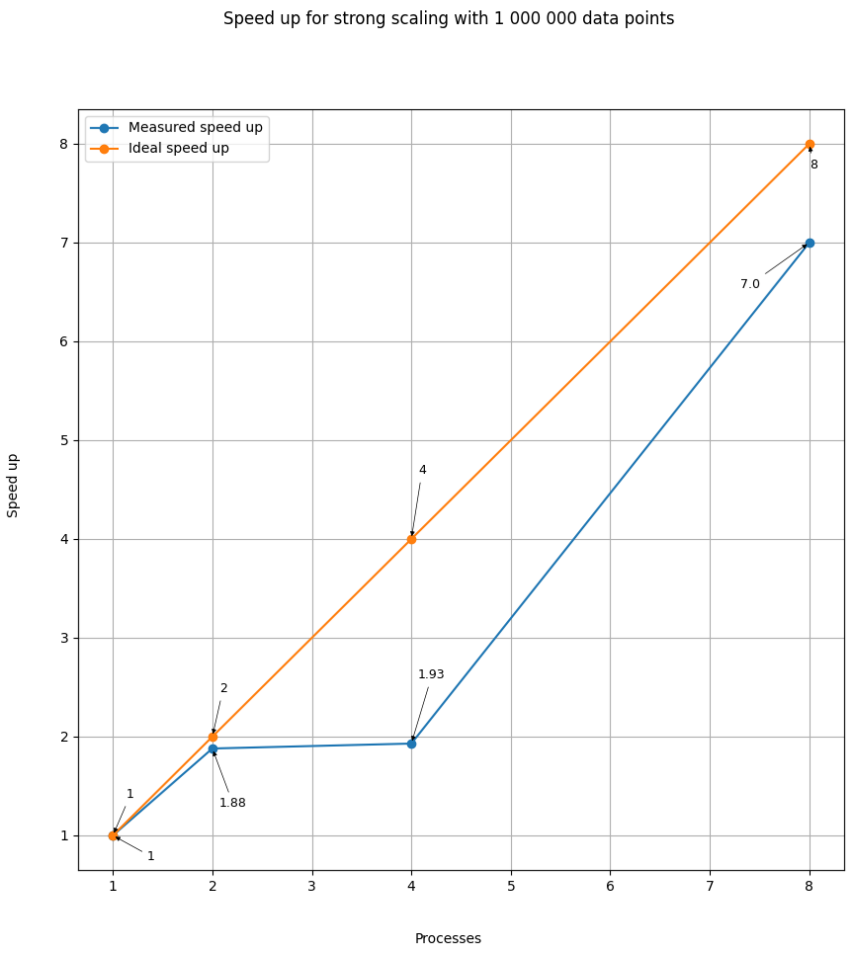

Figure 1: Speed up for strong scaling with one hundred million data points when only one stencil is applied

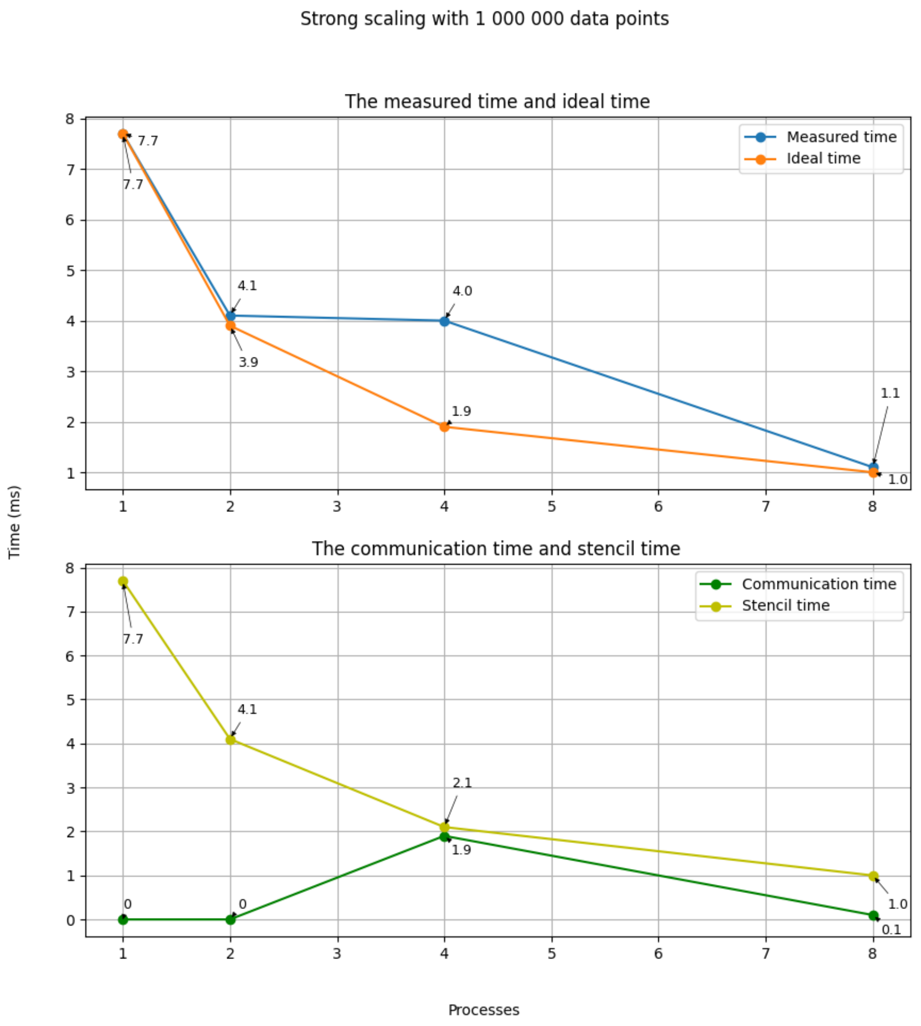

Figure 2: Strong scaling with one million data points as input when only one stencil is applied

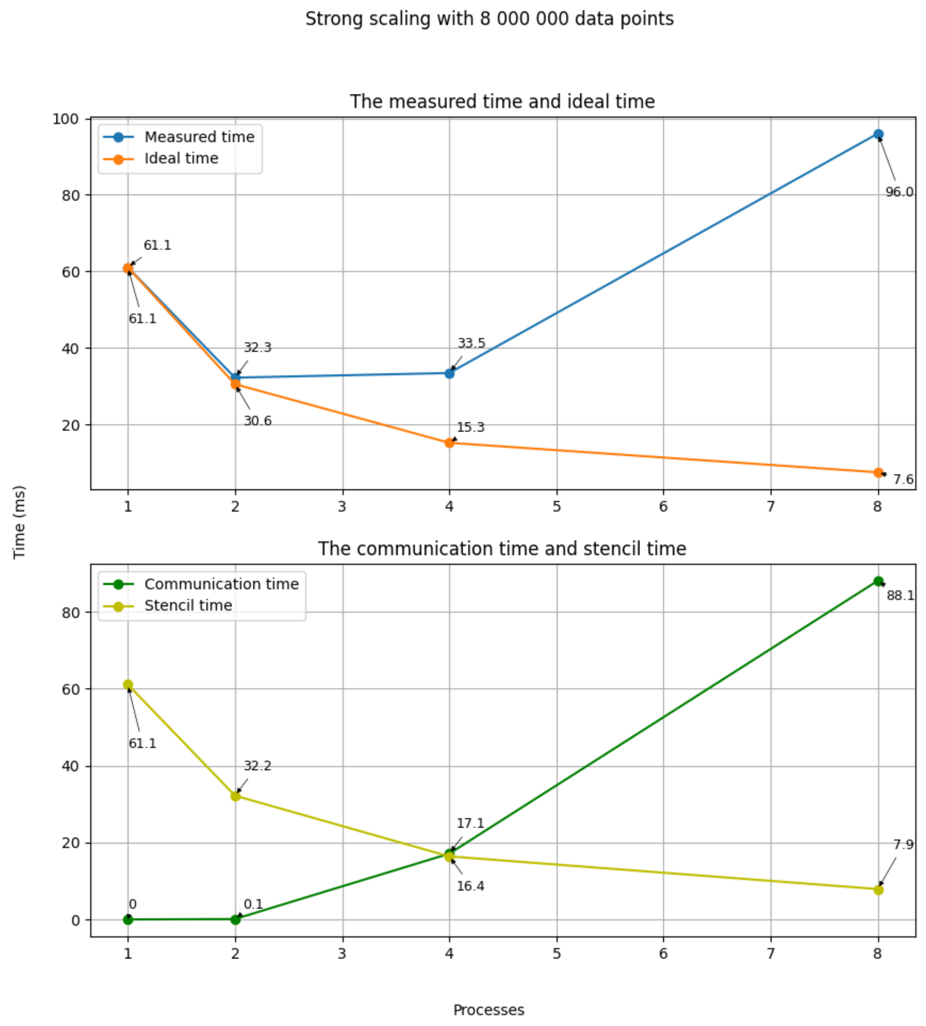

Figure 3: Strong scaling with eight million data points as input when only one stencil is applied

3.3.2 Strong scaling, stencil applied 100 times

Table 2: The experiment values from strong scaling when one hundred stencils are applied, 1M corresponds to one million data points, and the results is in milliseconds

Processes \ Input size	1M	8M
1	274.3	2223.84
2	185.71	1486.01
4	59.18	791.59
8	26.04	471.57

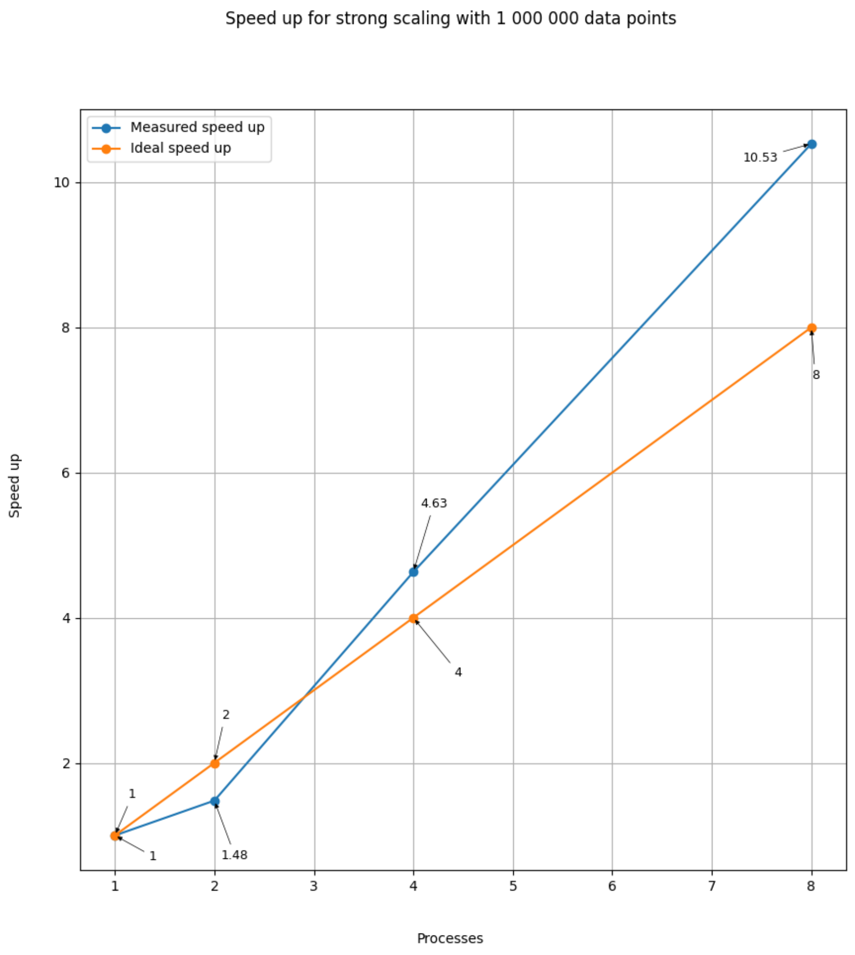

Figure 4: Speed up for strong scaling with one hundred million data points when one hundred stencils are applied

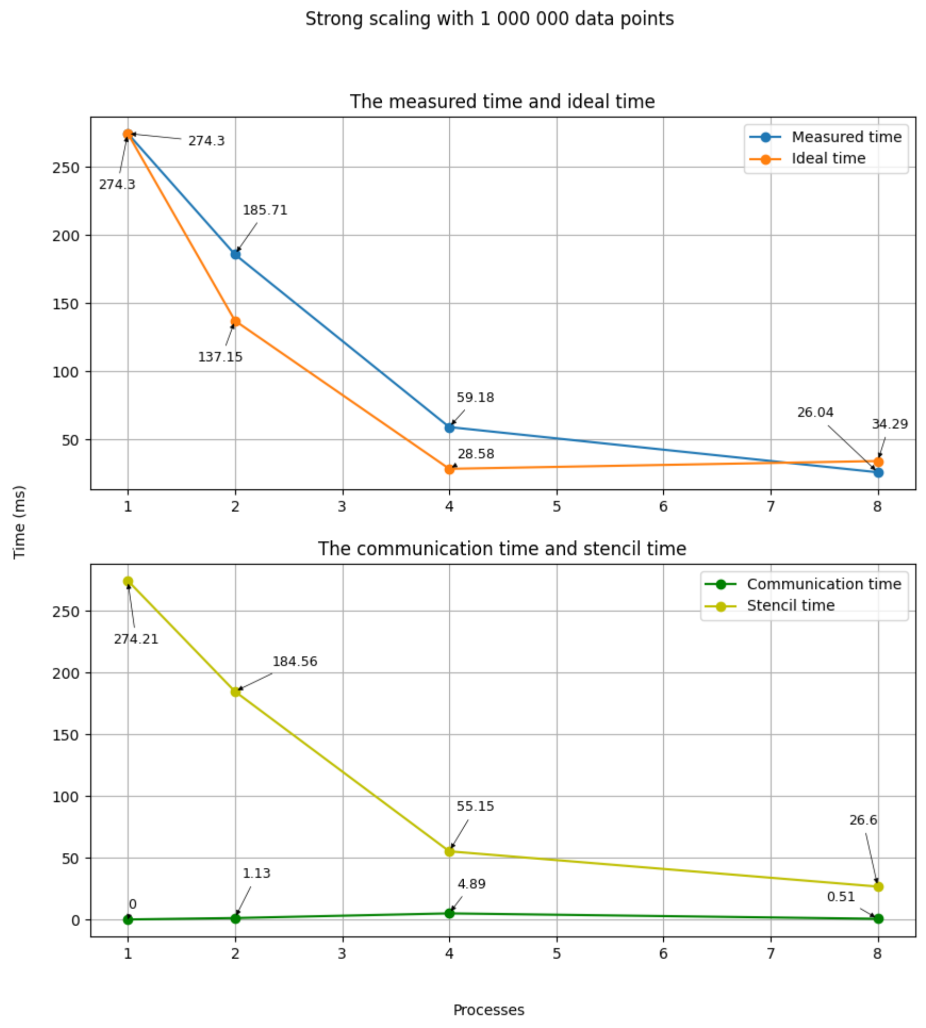

Figure 5: Strong scaling with one million data points as input when one hundred stencils are applied

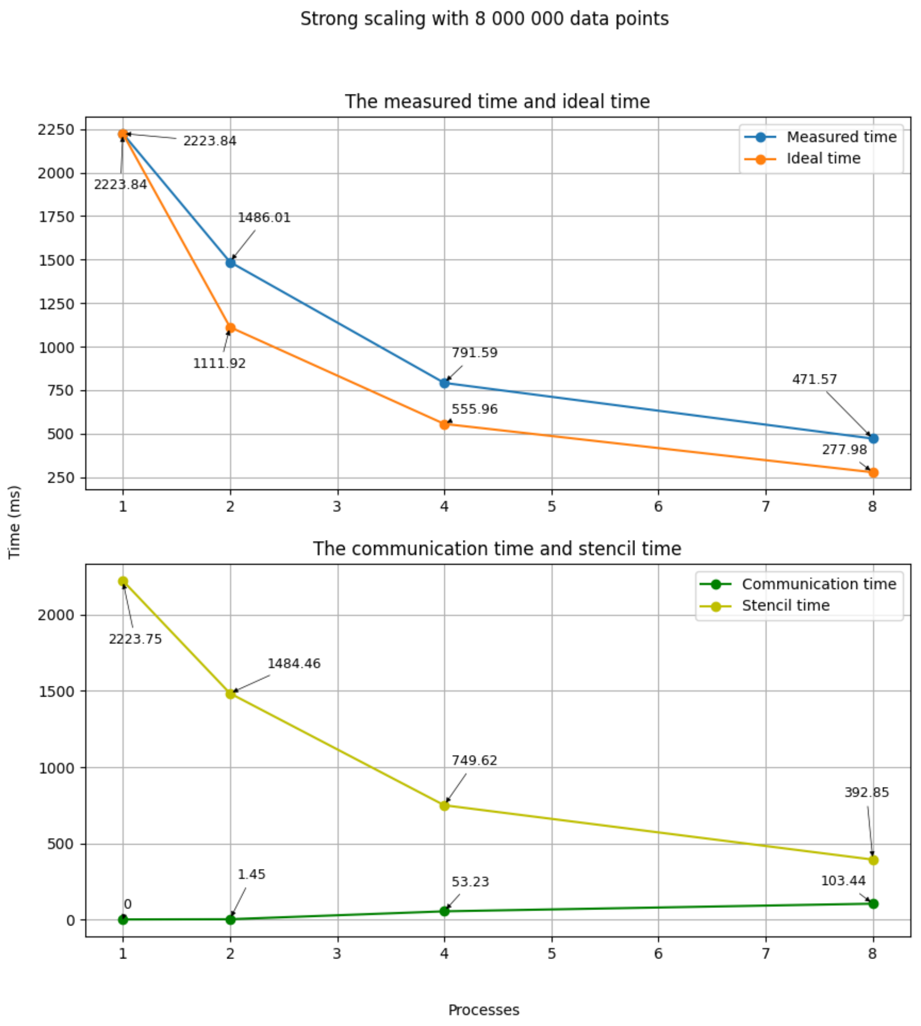

Figure 6: Strong scaling with eight million data points as input when one hundred stencils are applied

3.3.3 Weak scaling, stencil applied 1 time

Table 3: The experiment values from strong weak scaling when only one stencil is applied, 1M corresponds to one million data points, and, the result is in milliseconds

Processes & Input size	Time [ms]
1 & 1M	7.7
2 & 2M	8.1
4 & 4M	18.1
8 & 8M	93.4

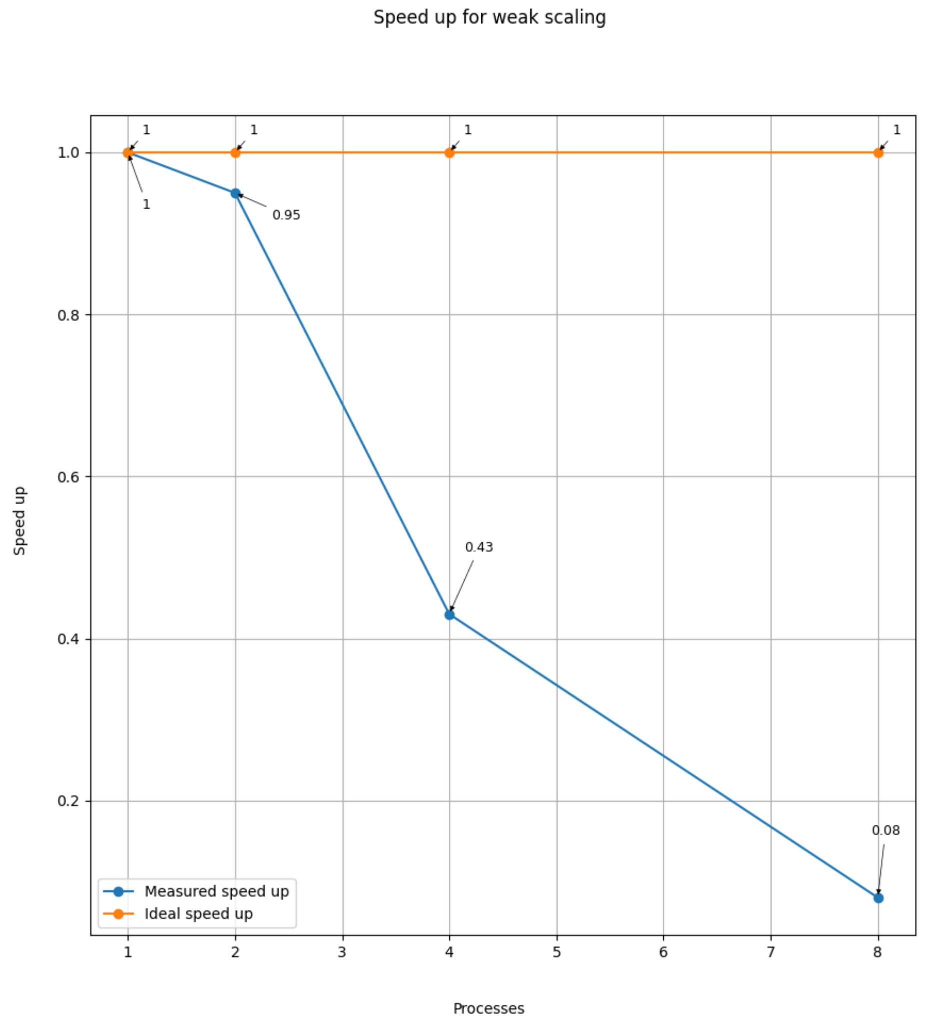

Figure 7: Speed up for weak scaling when only one stencil is applied

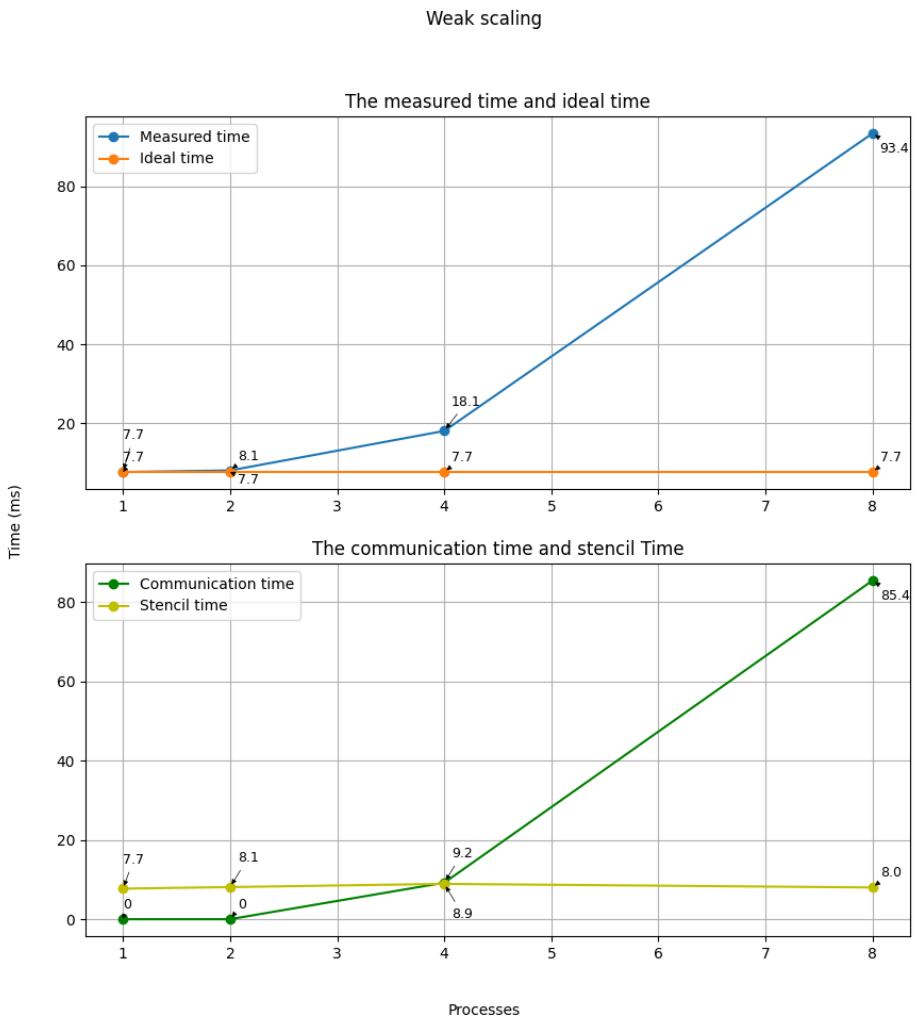

Figure 8: The weak scaling results when only one stencil is applied

3.3.4 Weak scaling, stencil applied 100 times

Table 4: The experiment values from strong weak scaling when one hundred stencils are applied, 1M corresponds to one million data points, and, the result is in milliseconds

Processes & Input size	Time [ms]
1 & 1M	274.15
2 & 2M	379.49
4 & 4M	388.18
8 & 8M	558.02

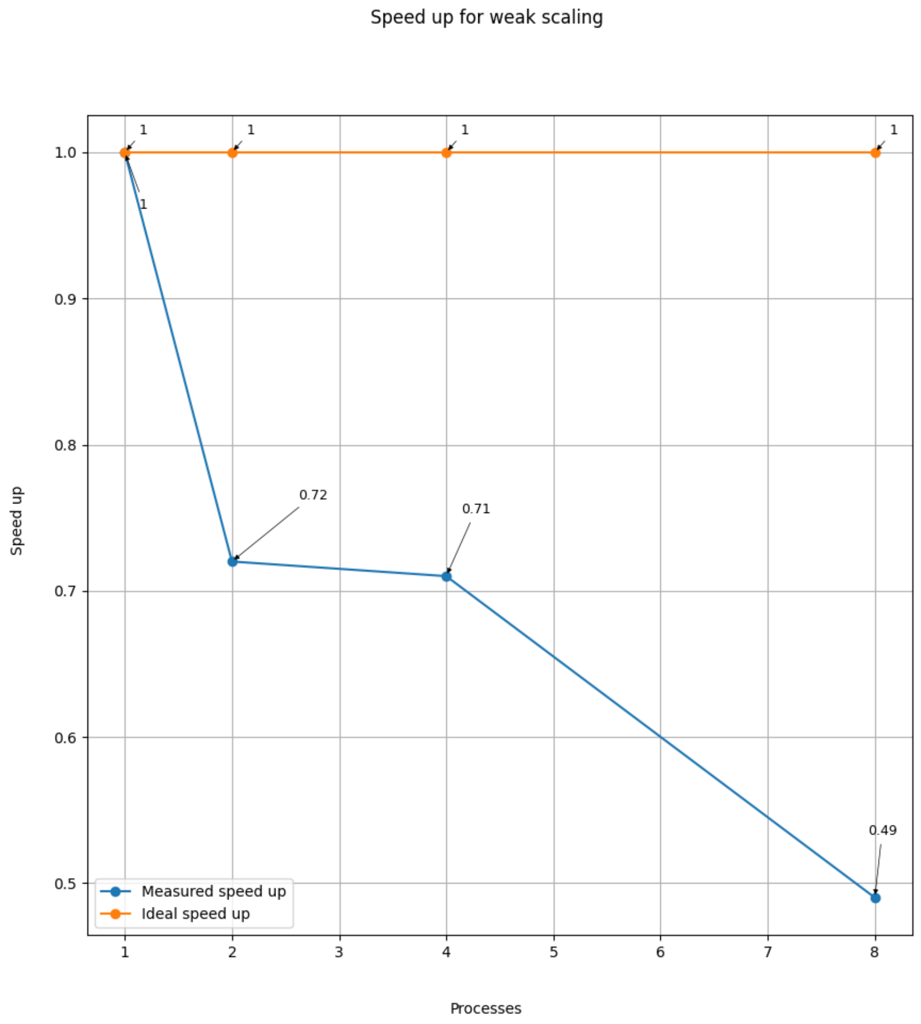

Figure 9: Speed up for weak scaling when one hundred stencils are applied

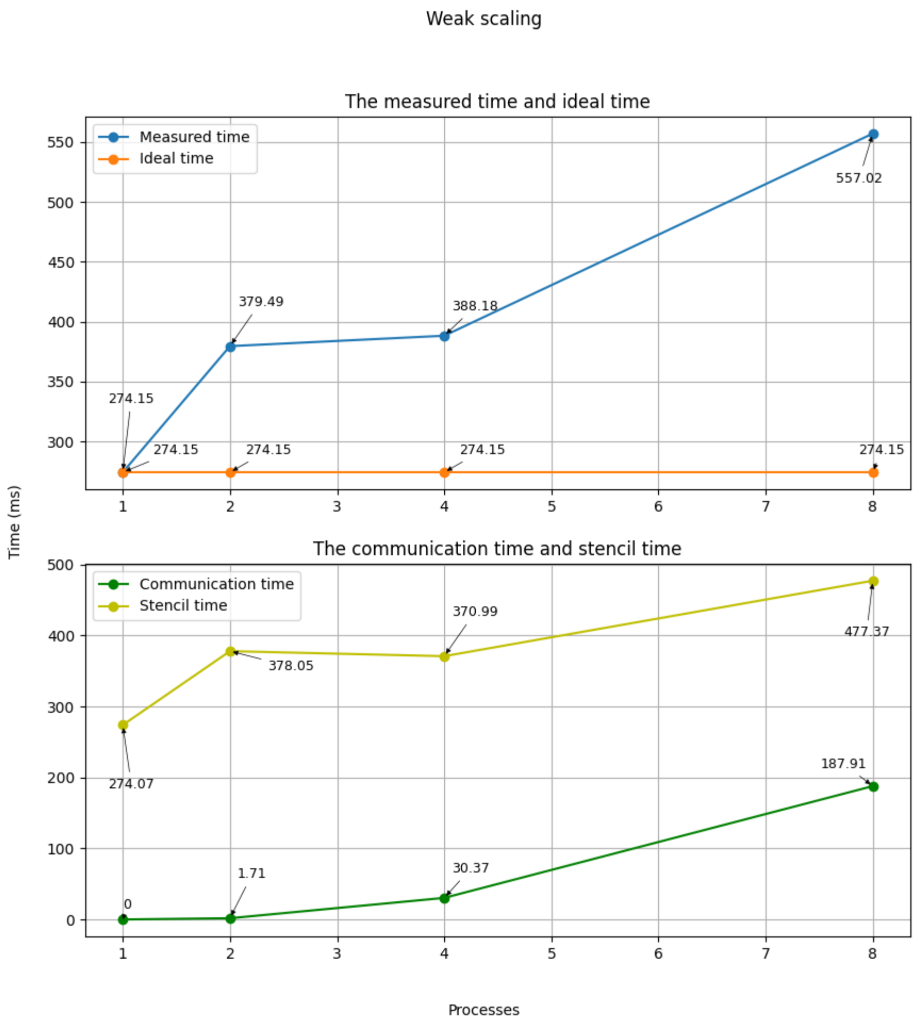

Figure 10: The weak scaling results when one hundred stencils are applied

3.4 Verification

There are a few files associated with this project that can be used to verify our solution. For the input file "input96.txt" there are two output files "output96_1_ref.txt" and "output96_4_ref.txt" the can be compared to the output of our program when the stencil is applied once, or four times. Below, in figures 11 and 12 the program output is plotted together with the input data. The resulting plots were identical for one or many MPI processes.

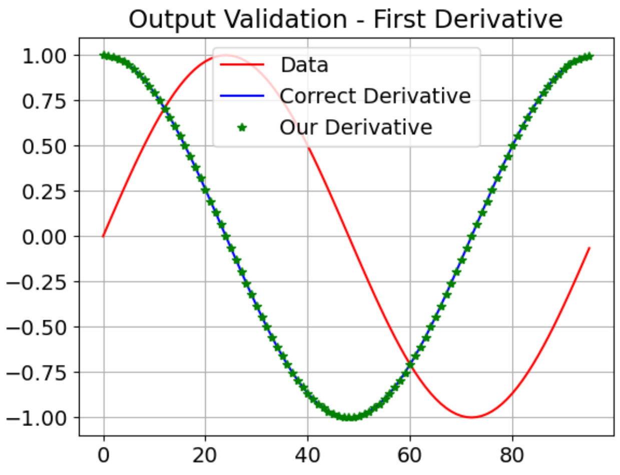

Figure 11: The first derivative of input96.txt

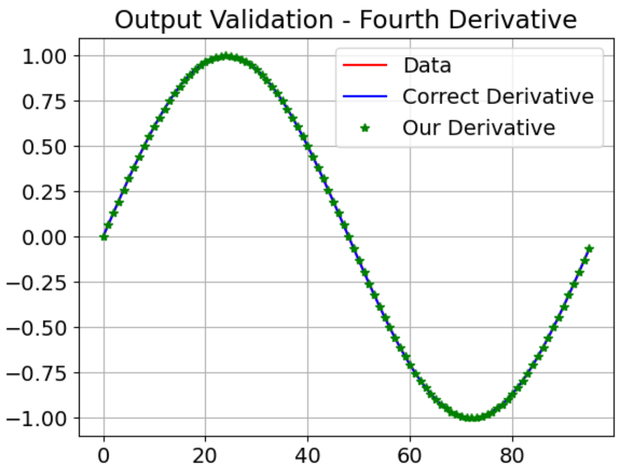

Figure 12: The fourth derivative of input96.txt

4 Discussion and analysis

In our experiments, we have near ideal scaling for strong scaling, except for when are only applying one stencil with eight million data points. For that test, only scale good when going from one to two processes, and the reason for this is a large increase in the communication time that arises when we go beyond two processes for that case. For weak scaling we do not come close to ideal scaling, however, we got much closer to ideal scaling when we are using one hundred stencil applications instead of only one.

The reason for achieving better scaling when increasing the number of stencil applications is that the communication time gets dwarfed by the stencil time, the other is true when a low number of repeated stencil number is used.

4.1 Communication time

When performing the stencil application with multiple MPI processes each process send four elements in total to the neighbouring processes. The first two elements are sent to one neighbour and the last two elements are sent to the opposite neighbour. So, with each process there are two send calls, and two receive calls and one **MPI_Waitall** call, which makes five MPI calls in total, for each stencil application. These five calls are performed in parallel with non-blocking communication so in the ideal case the total time should not increase with an increased number of processes. Even in the worst ideal case, where all communications are made serially, we would not expect the communication time to be worse than linearly increasing as $T(t) = nt + C$, where T is the total communication time, t is the communication time for one process, n is the amount of processes and C is some constant. However, this is not what we observe in our experiments, accurately described in the bottom graphs in figures 2 and 3.

In our experiments we measure close to zero communication time with two processes but when going beyond that, the communication time increases rapidly. The slopes between two-four and four-eight processes are different from each other but the different is not as dramatic as both of them compare to the slope of going form one to two process where we see a huge increase in time.

For smaller input arrays, such as 1.000.000 datapoints as we seen in figure 2, we observe very strange behavior of the communication time between the processes. We find close to zero communication time for one, two and eight processes but a huge increase (roughly 19 times) for four processes. This increase differ somewhat but is directionally true for both one stencil application and one hundred. This result was consistent over many runs and deviates greatly from the theory. We could also see that some processes completed significantly faster, and therefore a significant portion of their time waiting for the other threads to catch up. This was also unexpected since we (aside from process 0 when distributing the data) have a completely even load balance between the different processes.

We have thus determined the communication time to be non-linear, far from the ideal case and dependent on the size of the input array.

4.2 Stencil time

The stencil application section of the program is a perfectly parallelizable task (after receiving the neighbouring elements) and our experiments prove that. In the bottom graphs in figures 2 and 3, we can see that for strong scaling, the total stencil application time decreases close to $\frac{1}{n}$ where n is the number of parallel processes.

4.3 Other solutions

We have experimented with both blocking and non-blocking communication but our results where consistent with both of these methods and we saw no performance increase.

We have also experimented with first initiating the communication then starting with the middle elements in the array, that require no communication between other processes, and then to wait for any of the neighbouring elements to arrive and apply the stencil with those elements. This approach held the most promise of success but when performing the experiments we found no significant increase in the performance and the code became very unreadable.

We settled on an intermediate solution where we start by initiating communication, then applying the stencil on the middle elements and then waiting for ALL neighbouring elements to arrive before applying the stencil on the first and last elements. With this solution the code remains readable, and we find close to equivalent performance as the approach described in the paragraph above.