# Parallelization of a Stencil application

Victoria Sedig
Parallel and Distributed Programming
Uppsala University

20 April 2021

# 1 Introduction

## 1.1 Project

In this project a code for a stencil application was parallelized. The stencil was in this case used to calculate the n;th derivative of values in an array. To do this for each value you need the two values next to it both on the right and the left side.

## 1.2 Parallelization

After variable declarations processor 0 does a MPI_scatter to sort out values to the different processors. It is assumed that the number of values can be equally divided with the number of processors. Since the two values next to the value of the index is needed to calculate the next value means that there will be missing necessary values for each processor when it is going to compute the values on the corners of the array. The problem will occur for the first and last two elements in each array. Therefor each processor sends the elements missing to each other to be able to do the computation. This is done with MPI_send and MPI_recv. After the stencil operation is performed processor 0 will gather all the resulting arrays to the final array. All processors take execution time but only the largest on gets saved.

# 2 Verification of Correctness

The original code and the parallelized code's output files were compared with each other using the diff command. This was made with different sizes, steps and number of cores. For all cases there was no difference.

# 3 Experiments

## 3.1 Description

The experiments were made at the linux host gullviva at the evening. There were only two other persons using the system found with the who command on the linux terminal. The experiments were run with three scripts generating three result files containing the times. Two scripts ran the scripts with

variating number of cores and one script with variating size of the problem. The scripts were re-run 5 times and the times used for the result are the averages. The calculations and plots were made in Matlab.

## 3.2 Results

In Table 1, Table 2, Figure 1 and Figure 2 the result for the speedup is seen. The size of the problem is put to a value that can be evenly divided with all the number of cores tested.

Table 1: The speedup for different number of cores. The sequential timing used to calculate the speedup was $t_{seq} = 1.889678$. The size and steps for the problem was constant and put to $size = 10! = 3628800$ and $steps = 100$.

| #Cores | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| Speedup | 0.7394 | 1.1298 | 1.1585 | 1.6539 | 1.9083 | 2.2169 | 2.4849 | 2.6458 | 2.7139 |

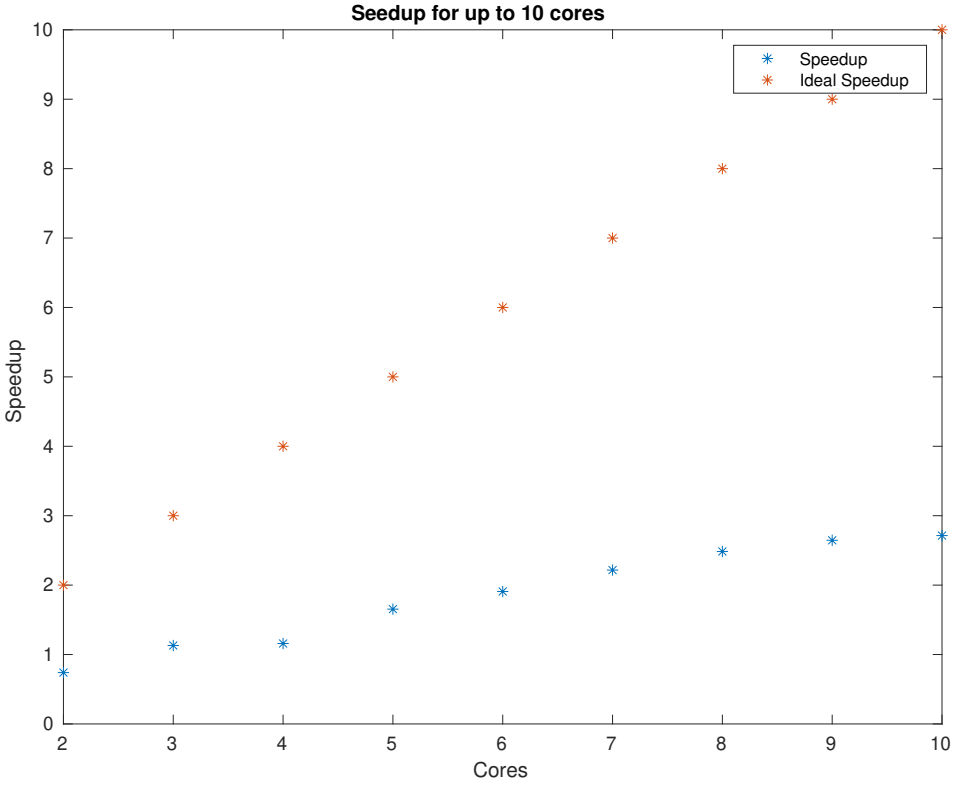


Figure 1: The speedup and the ideal speedup for the parallelized program plotted against the number of cores used. The size and number of steps were kept constant.

Table 2: The speedup for different number of cores. The sequential timing used to calculate the speedup was $t_{seq} = 0.449193$. The size and steps for the problem was constant and put to $size = 1048576$ and $steps = 100$.

| #Cores | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|
| Speedup | 0.5091 | 0.9173 | 1.5384 | 2.3161 | 2.9910 | 0.7284 |

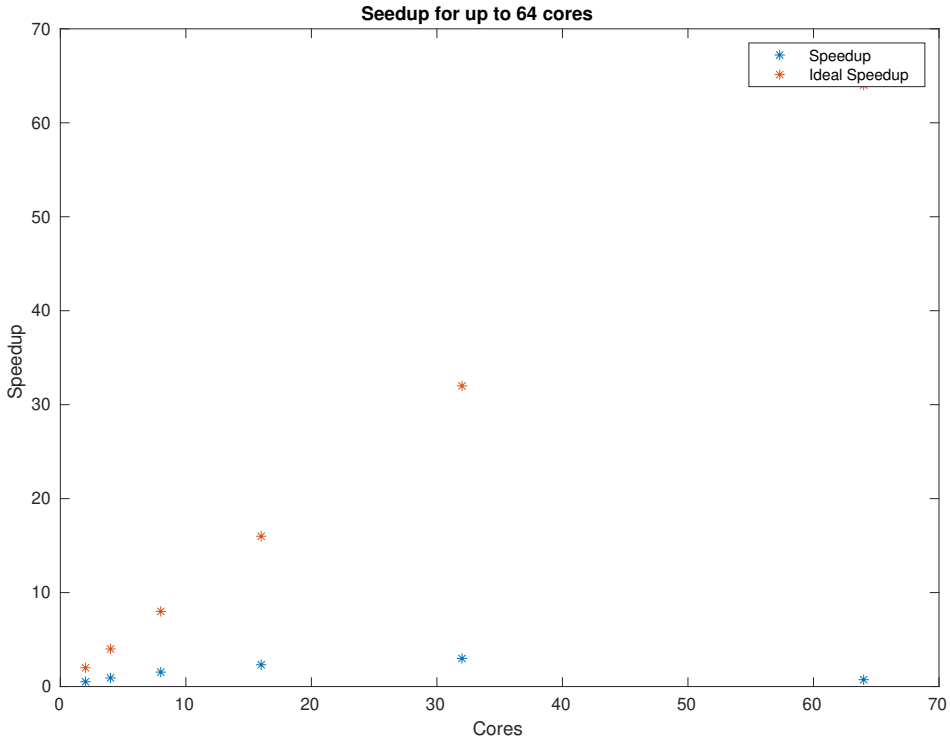


Figure 2: The speedup and the ideal speedup for the parallelized program plotted against the number of cores used. The size and number of steps were kept constant.

In Figure 3 the results for the execution times for both the sequential and parallelized code with increasing number of values is seen. The number of steps were put to 100 and for the parallelized code the number of cores where constantly 16 since that gave a good speedup.

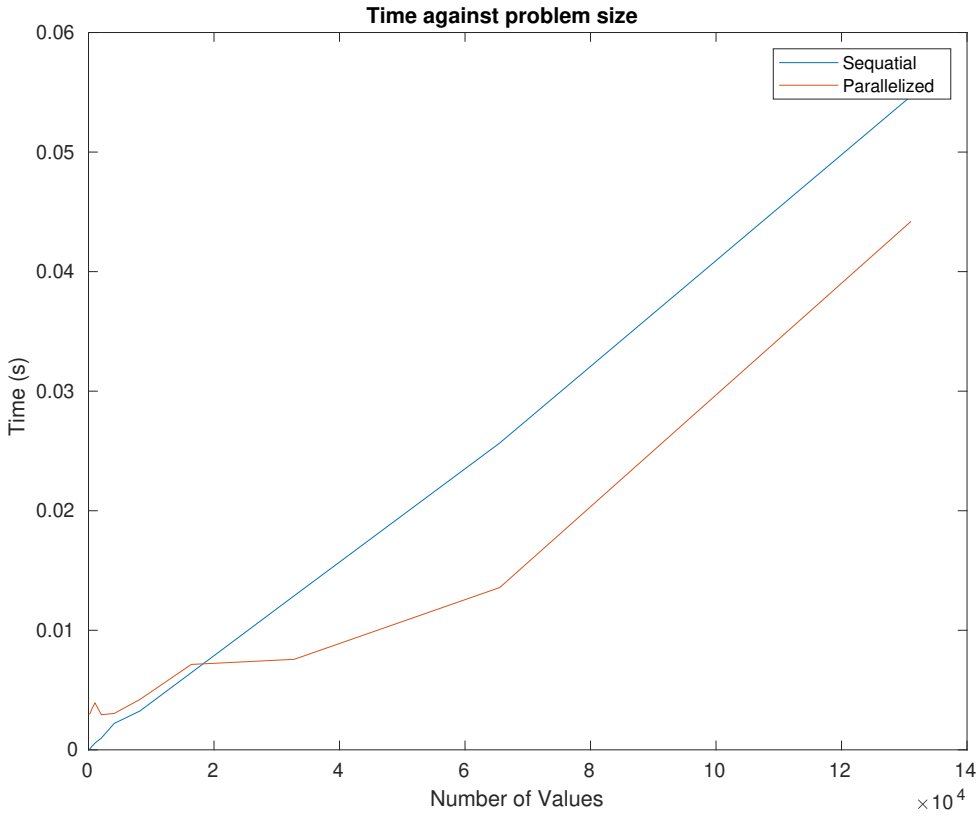


Figure 3: The execution time for the sequential and arallelized code plotted against number of values. The number of steps and cores were kept constant.

# 4 Discussion

In Figure 1 the results is visual that the pattern has a linear behaviour, as expected, however it is not in the capacity of the ideal speedup. This may be because there is extra work of sending all the messages containing values to each other. Also the scatter and gather commands are costly. In Figure 2 there is a similar behavior except the core of 64 which is a lot worse than it is at 32. This could be because it takes time to create a large number of processors.

In both Figure 1 and Figure 2 the speedup is below 1 for 2 cores which means that there were no speedup at all, the execution time is smaller than the sequential code execution time. This may be because all of the extra work that has to be done to do the parallelization and the creation of more than one processor.

In Figure 3 it is visualised that the sequential code is better when the size of the problem is small, which was expected. To create processors and do send and receive takes some time, which means that if the problem is already fast, the extra time to make parallelization affects more. When the size of the problem starts to get big the parallelized code is faster.