Assignment 2

Cecilia Ahnlund, Einar Johansson

March 2025

1 Parallelization description

To parallelize the original sequential implementation, the three nested for loops were merged into a single loop. The index for accessing the input array was calculated using equation (1), where i and j represent the indices of the input value and the stencil element, respectively. Here, num_values denotes the total number of input elements, and EXTENT refers to half the width of the stencil.

$$\text{index} = (i - \text{EXTENT} + j + \text{num\_values}) \bmod \text{num\_values} \tag{1}$$

The input array was then partitioned into p blocks for parallel processing, with the start and end indices for each block calculated using the following equations: (2) and (3) where $q = \text{floor}\left(\frac{n}{p}\right)$ and $rest = n \bmod p$.

$$i_{start}(k) = \begin{cases} k(q+1), & 0 \leq k < rest \\ kq + rest, & otherwise \end{cases} \tag{2}$$

$$i_{end}(k) = i_{first}(k+1) - 1 \tag{3}$$

Once each block had been processed, the results from all processes were collected using the *MPIGather* function, which gathers data from each process in rank order into a single variable on process 0. To enable reuse of the results in subsequent runs, as this variable is NULL on all other processes, process 0 copied the gathered array into a global variable. This variable was then broadcasted to all other processes using the function *MPIBcast* and finally swapped with the previous input array.

2 Performance experiments

To ensure that the algorithm worked as intended, the smaller output files were first tested against the referenced ones for both different numbers of steps and processes. Once this gave the desired result the *MPIWtime* function was once again used to measure how long each run took. The time for one and four steps using one, two, four and eight processes for all different input files were recorded

in order to compare them to a sequential run. The speedup was calculated using the equation (4). The same results are displayed against the ideal speedup in figure 1.

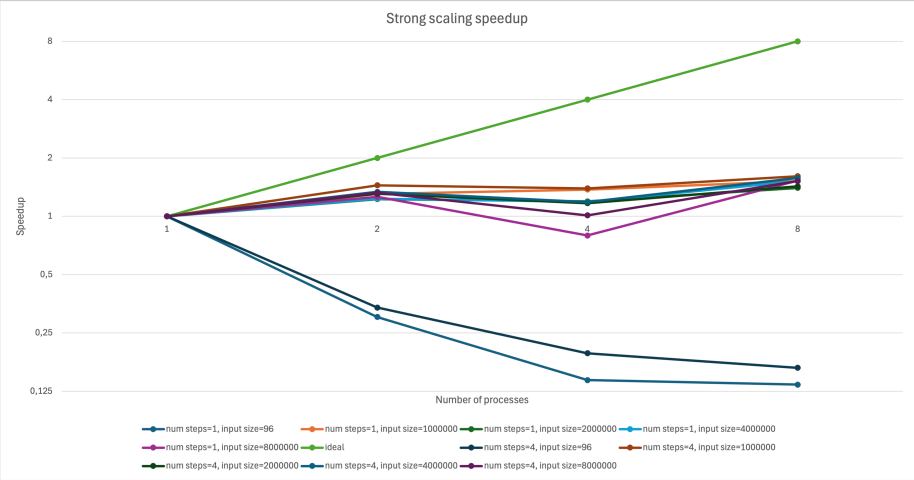

Figure 1: Plot of the speedup for a strong scalability

For a weak scalability the efficiency was calculated with equation (5). The results can be seen in figure 2.

Figure 2: Plot of the efficiency for a weak scalability

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \tag{4}$$

$$\text{Efficiency} = \frac{T_1}{p \cdot T_p} \tag{5}$$

3 Discussion

Figure 1 indicates using smaller input files actually lead to a performance slowdown, particularly when multiple processes are utilized. This suggests that the computational workload is not enough to justify the overhead introduced by parallelization. As a result, the overhead dominates execution time, making single process execution more efficient.

For larger input files it is clear the workload justifies the use of several processes as both tests for eight processes perform the best. However, after the first initial speedup there is a large drop for several tests for the 2000000 sized input file, before most recover and experience a better speedup for the next couple of files. The scaling for this kind of problem is expected to not be linear, since the mathematical operations require access to neighboring values limiting total parallelization. The drop in speedup could therefore have many different reasons. For example since the matrix is allocated as a consecutive array, leading to each value being next to each other in the memory, poor performance might be a result of false sharing. Meaning an entire cache line is thrown out each time a value is replaced. Which results in more cache misses and thereby slower execution.

When comparing the ideal speedup to the actual speedup achieved in figure 1, it becomes clear that the program's performance is suboptimal. Even though larger files experience a small speedup it does not increase much with the number of processes. As already mentioned, these kind of computations are not linear and as the computations are not independent - it is not possible to totally parallelize the program. As more processes are introduced, the amount of inter-process communication increases significantly, leading to more message-passing and thereby a larger overhead and non effective scaling.

When evaluating the efficiency for weak scalability, a noticeable decline in efficiency is observed as the number of processes increases, this can be seen in figure 2. This drop is most pronounced with the smallest input files, which is expected as these workloads are too light to offset the overhead introduced by parallelization. In contrast, the larger input files maintain relatively consistent efficiency across a varying number of processes, suggesting they scale more effectively under increased parallelism.

However the efficiency is still notably low for a large number of processes. Even though theoretically this seem strange as parallelization should improve execution it is not, as mentioned before, linear. As each computation requires data from neighboring values communication will increase with the number of processes, giving a lower efficiency.

input size	num steps	time (seconds)	processes	speedup	efficiency
96	1	0,000013	1	1	1
96	1	0,000043	2	0,302325581	0,151162791
96	1	0,000091	4	0,142857143	0,035714286
96	1	0,000096	8	0,135416667	0,016927083
96	4	0,000022	1	1	1
96	4	0,000065	2	0,338461538	0,169230769
96	4	0,000112	4	0,196428571	0,049107143
96	4	0,000133	8	0,165413534	0,020676692
1000000	1	0,031519	1	1	1
1000000	1	0,024177	2	1,303677048	0,651838524
1000000	1	0,022954	4	1,37313758	0,343284395
1000000	1	0,020702	8	1,522509902	0,190313738
1000000	4	0,097317	1	1	1
1000000	4	0,067342	2	1,445115975	0,722557988
1000000	4	0,069836	4	1,393507646	0,348376912
1000000	4	0,06046	8	1,609609659	0,201201207
2000000	1	0,046479	1	1	1
2000000	1	0,037897	2	1,22645592	0,61322796
2000000	1	0,038922	4	1,194157546	0,298539386
2000000	1	0,033155	8	1,401870005	0,175233751
2000000	4	0,141797	1	1	1
2000000	4	0,108483	2	1,30708959	0,653544795
2000000	4	0,121288	4	1,169093398	0,292273349
2000000	4	0,099293	8	1,42806643	0,178508304
4000000	1	0,097921	1	1	1
4000000	1	0,079409	2	1,23312219	0,616561095
4000000	1	0,082474	4	1,18729539	0,296823848
4000000	1	0,06432	8	1,522403607	0,190300451
4000000	4	0,293155	1	1	1
4000000	4	0,219115	2	1,337904753	0,668952377
4000000	4	0,247344	4	1,185211689	0,296302922
4000000	4	0,186019	8	1,575941167	0,196992646
8000000	1	0,190717	1	1	1
8000000	1	0,151366	2	1,259972517	0,629986258
8000000	1	0,239588	4	0,796020669	0,199005167
8000000	1	0,124574	8	1,530953489	0,191369186
8000000	4	0,582811	1	1	1
8000000	4	0,440701	2	1,32246353	0,661231765
8000000	4	0,576056	4	1,01172629	0,252931573
8000000	4	0,383344	8	1,520334217	0,190041777

Table 1: Data from performance experiments.