PDP Assignment 2

Arvid Ehrengren, Viktor Kangasniemi

May 8, 2025

1 Description of Parallelization

For the implementation of parallelization of the stencil computation, we used MPI and distributed the data among processes in a 1D array. The program begins by reading the entire input array on the root process (rank 0). Afterwards, the process scatters equally sized chunks (given that we can assume that the input array is divisible by number of processes) of the data to all active processes using MPI_Scatter. Each process then applies the stencil to its respective local chunk of the data x times, given by the input. Since the stencil computation requires access of neighboring elements, meaning that each process needs to acquire data from adjacent processes, we use MPI_sendrecv to exchange the specific elements with the neighboring processes. Each process sends the two elements farthest to its left and right and in the same way receives the two corresponding elements from adjacent processes, utilizing module arithmetic to make the first and last process wrap around as neighbors. After all the stencils have been applied, each process sends its final result back to the root process using MPI_Gather. The root process then writes the combined result to the output file.
In the implementation, we also used a MPI_Barrier to synchronize the processes and receive the maximum execution time across all processes, to ensure accurate timing of the stencil computation.

2 Performance experiments

We evaluated both strong and weak scaling of the parallel stencil program. Strong scaling was tested on a fixed input size (input8000000.txt) and varying the number of MPI processes (1,2,4,8). Weak scaling was tested by increasing both the input size and the number of processes proportionally, increasing the input size along with the number of processes used. In both cases we applied the stencil 100 times and recorded the maximum execution time across all processes.

3 Results

3.1 Strong Scaling

Table 1: Strong Scaling Results

Number of Processes	**Execution Time (s)**	**Speedup**
1	1.333500	1.00
2	0.703517	1.90
4	0.509327	2.62
8	0.357372	3.73

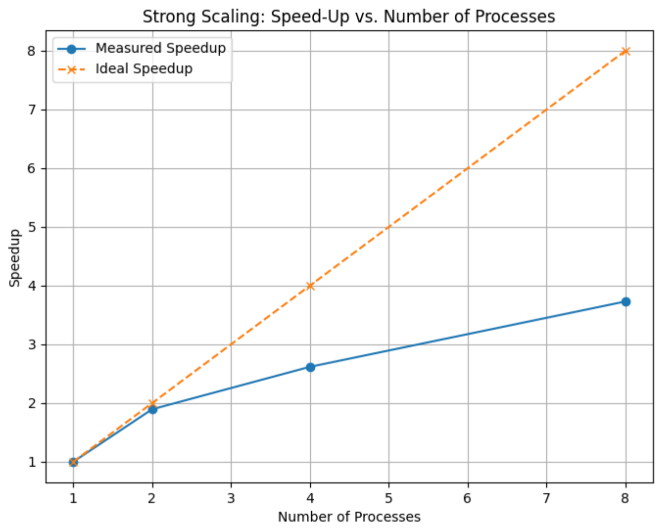

Figure 1: Speedup comparison between 1,2,4 and 8 processes.

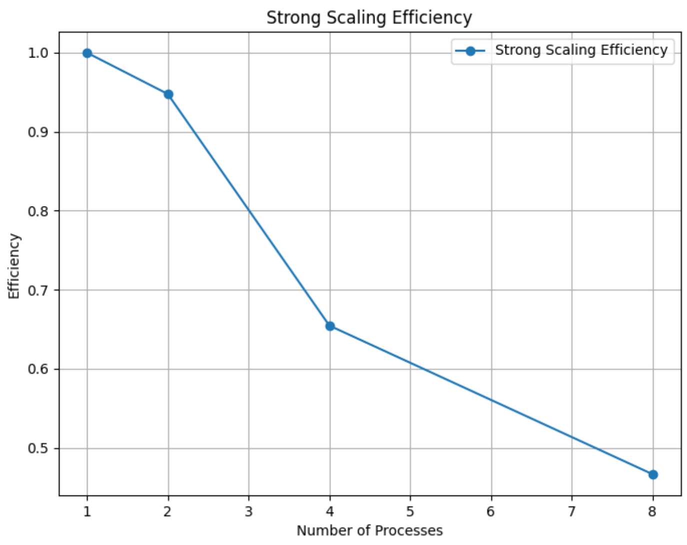

Figure 2: Efficiency comparison between 1,2,4 and 8 processes.

3.2 Weak Scaling

Table 2: Weak Scaling Results

Number of Processes	**Execution Time (s)**	**Speedup**
1	0.172503	1.00
2	0.194696	0.89
4	0.235923	0.73
8	0.355003	0.49

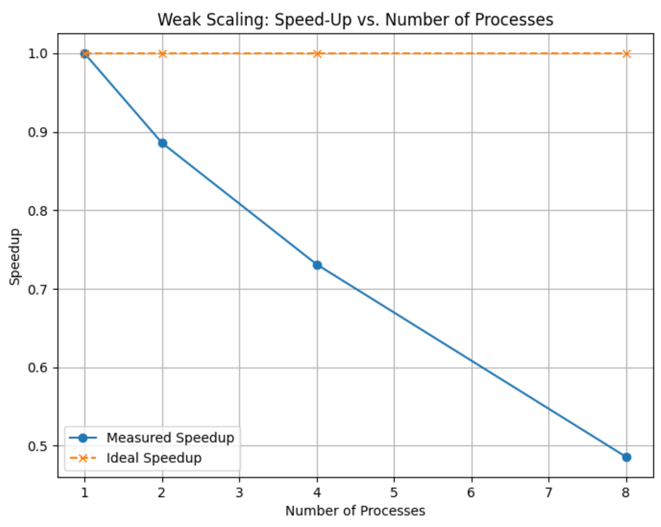

Figure 3: Speedup comparison between 1,2,4 and 8 processes.

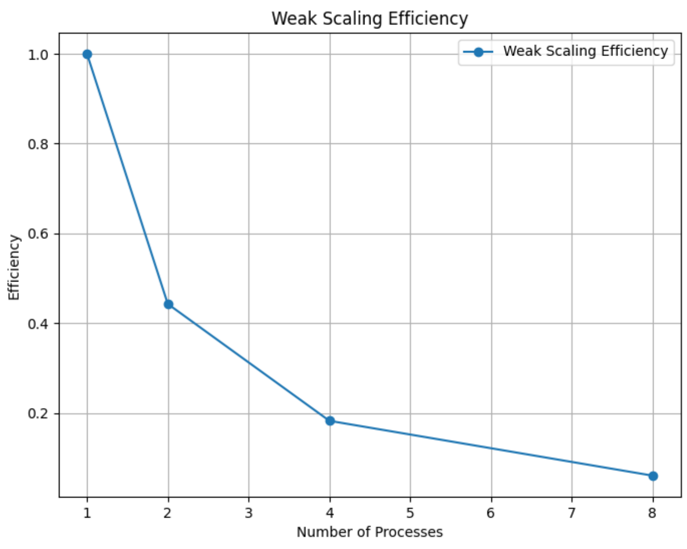

Figure 4: Efficiency comparison between 1,2,4 and 8 processes.

4 Discussion

The performance results of our parallel stencil program generally align with the expectations for both strong and weak scaling.

In the strong scaling experiment, we observe that with great scaling a decrease in execution time increases as the number of processes increases. As seen in Table 1, the speedup achieved with 8 processes is approximately 3.73x compared to the single process which can be seen as an effective parallelization. However, Figure 1 shows how the speedup does not remain linear, especially as the number of processes increases. While some of this can be attributed to communication overhead from the data exchange in the stencil computation, a more significant factor is the limited cache size of Rackham's nodes, where each core has a 32 KB L1 data cache, 256 KB L2 cache, and a 25 MB L3 cache. This means that, as the problem size increases the working data become too large to fit in the cache, leading to a significant increase in cache misses, decreasing the performance. The efficiency plot reflects this, showing decreasing efficiency as the number of processes grows.

For weak scaling tests, in Table 2 we can observe that execution times remained relatively stable, increasing from 0.173 seconds (1 process) to 0.355 seconds (8 processes) which is illustrated in Figures 3 and 4. This indicates that the program scales reasonably well. The small drop in efficiency is expected, but the performance is still relatively acceptable under increasing workloads.

Overall, the results confirm that our parallel implementation using MPI is effective for this type of workload. The implementation allowed us to achieve consistent performance across all tests.