Uppsala Universitet

Parallel and Distributed Programming 1TD070

Assignment 2

Agelii, Carl.

May 4, 2024

Parallelization

The task at hand was to parallelize a one dimensional stencil operation. In order to to this, MPI was used. First off, memory is allocated on the main process in order to read an input file. Then, the number of elements was broadcasted to all other processes using MPI_Bcast. Next, each process allocates an equal chunk of doubles for their designated data that was sent out using MPI_Scatter. In order to then perform the stencil operation, a 2·EXTENT array was allocated on each process in order to store the overlapping values. Then, they were sent and received in the correct destinations using MPI_Send and MPI_Receive. MPI_Reduce was used with the MPI_MAX type to get the largest time from the processes. Lastly, MPI_Gather was called to store the entire output.

Experiments

In order to evaluate the performance of the parallel implementation, some experiments were conducted. First off, the strong scalability was tested, meaning that the problem size was fixed to 8 million processes and the amount of processes were varied. After that, the weak scalability was studied by keeping the problem size fixed on each process. This was done by using 1, 2, 4 and 8 million elements on as many threads as a million elements.

Results

The timings from the performance experiments can be seen in table 1 and 2.

Table 1: Strong Scaling Times

Number of Processes	Execution Time (seconds)
1	0.044669
2	0.023079
4	0.011814
8	0.006674

Table 2: Weak Scaling Times

Number of Processes	Execution Time (seconds)
1	0.007032
2	0.005710
4	0.005733
8	0.005908

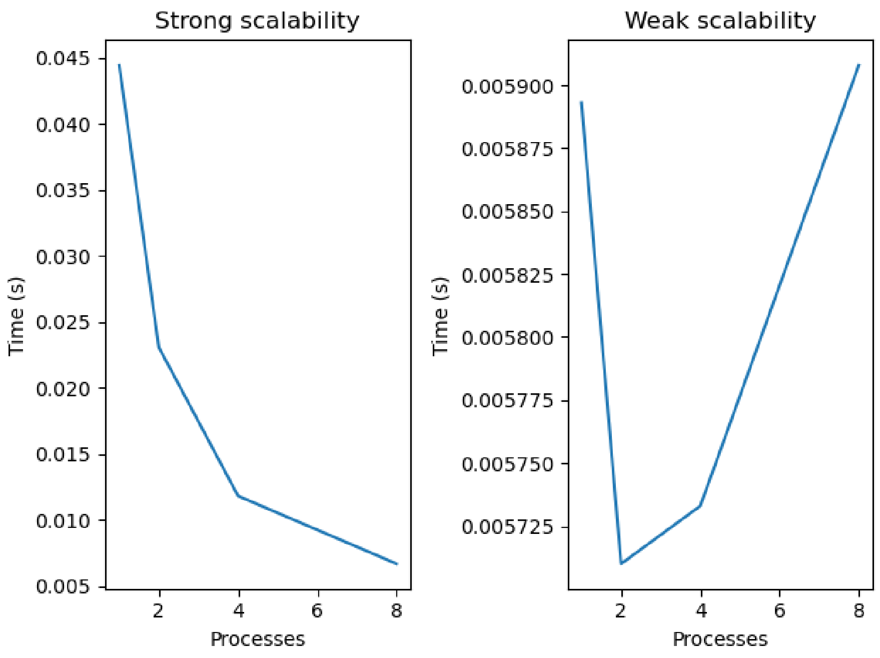

Figure 1: Times for strong and weak scalabilty

Table 3: Speed up for strong scalability

Number of Processes	**Speed up**
1	1
2	1.92
4	3.76
8	6.66

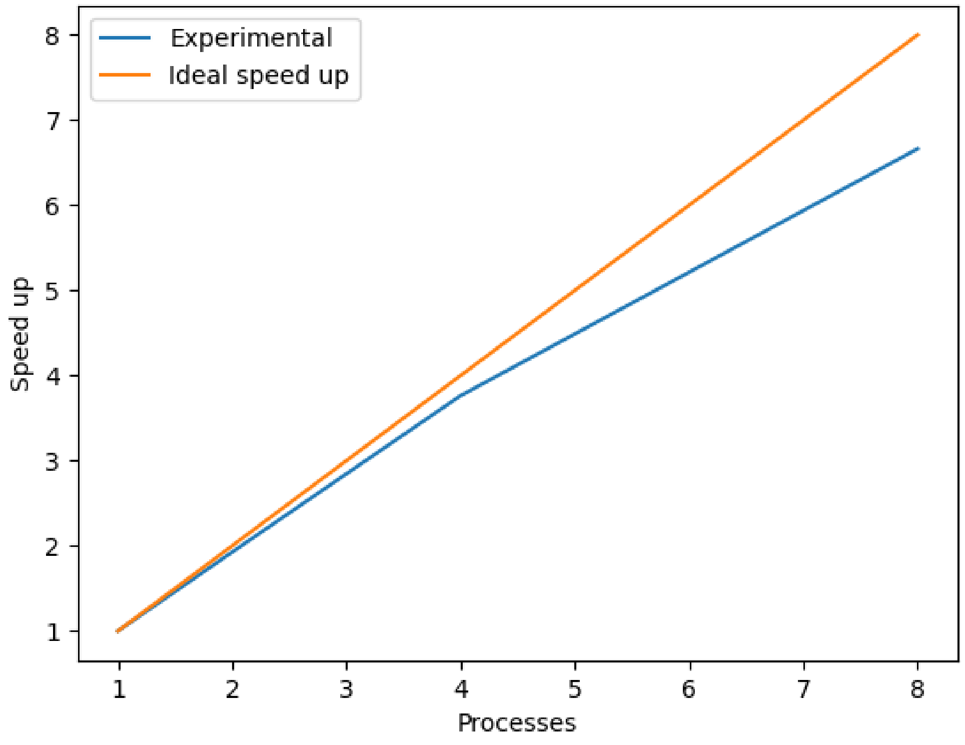

Figure 2: Speed up and ideal speed up for strong scalability.

Discussion

The algorithm seems to yield fairly good results. The speed up in figure 2 almost follows the ideal speed up. The difference is probably because of the communication overhead more processes introduce. A barrier was introduced before the timings started, in order to for all processes to get a ”fair” start. The weak scalability worked very well, virtually giving the same time for each run, except for the run with only one process. This is strange, because this should, in theory, yield the best time. The discrepancy could arise from the communication with itself, i.e. process 0 sends data to process 0.