Poisson solver in MPI

Louis Jaugey
louis.jaugey@epfl.ch

12 avril 2021

1 Introduction

In this work, we investigate the efficiency of parallelisation over multiple processes and nodes using MPI. The code is a 2D finite differences Poisson equation solver. The work is split horizontally i.e. each process computes $(N/n) \times N$ elements, where N is the number of element in one dimension and n is the number of processes. Communication is handled with ghost lines at the top and bottom of each grid, in which the last line of the grid above and first line of the grid below are copied. The program iterates until the accumulated L_2 error between steps reaches a given tolerance. The code was executed on Intel Broadwell processors running at 2.6 GHz, with 14 cores. Each node contains 2 of these processors i.e. 28 cores/node.

All values presented below are the average of 3 separate executions.

2 Strong scaling

In this section, the (strong) scalability of the code is studied. In an ideal case, the speed-up of the program's parallel section should scale linearly with the number of processes. Speed-up is given by $S = t_1/t_n$, where t_i is the execution time using i processes.

Figures (1) and (2) show the speed-up obtained using gcc/MVAPICH and Intel/Intel-MPI respectively. The number of processes is varied between 1 and 56. Since each node contains 28 cores, up to 2 nodes are used. The values are plotted on a log log graph, which means that polynomial expressions should be linear, with a slope equal to the exponent (since $\log(x^y) = y \cdot \log(x)$).

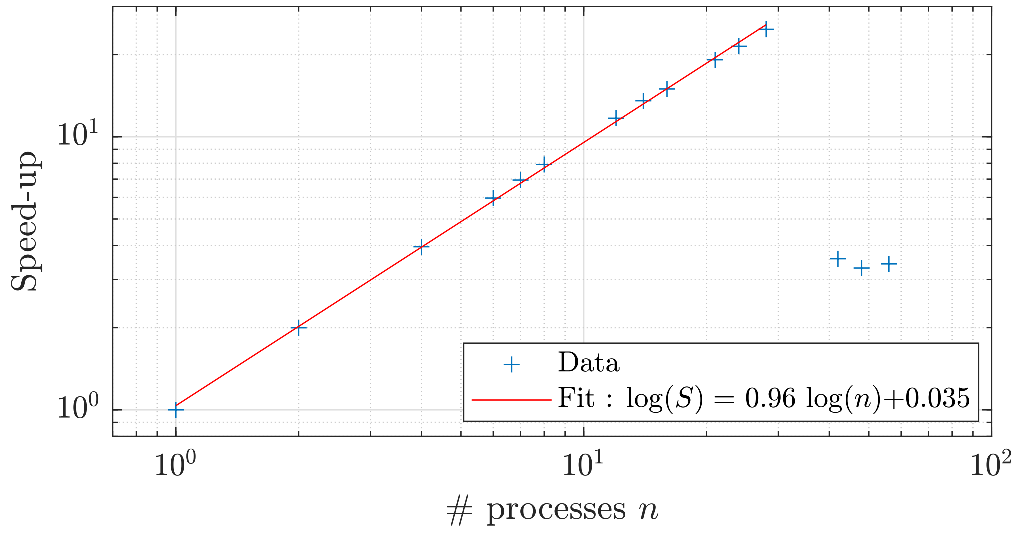

Figure 1 – Strong scaling using MVAPICH

As figure (1) shows, the speed-up is almost linear on a single node, though parallelisation is never perfect so the slope is not exactly 1. A significant drop in speed-up occurs when a second node is used (i.e. when $n > 28$). This is due to the inter-node communication, which is an expensive operation. The fit was done on data obtained with $n \leq 28$ to avoid off-setting the slope because of communication.

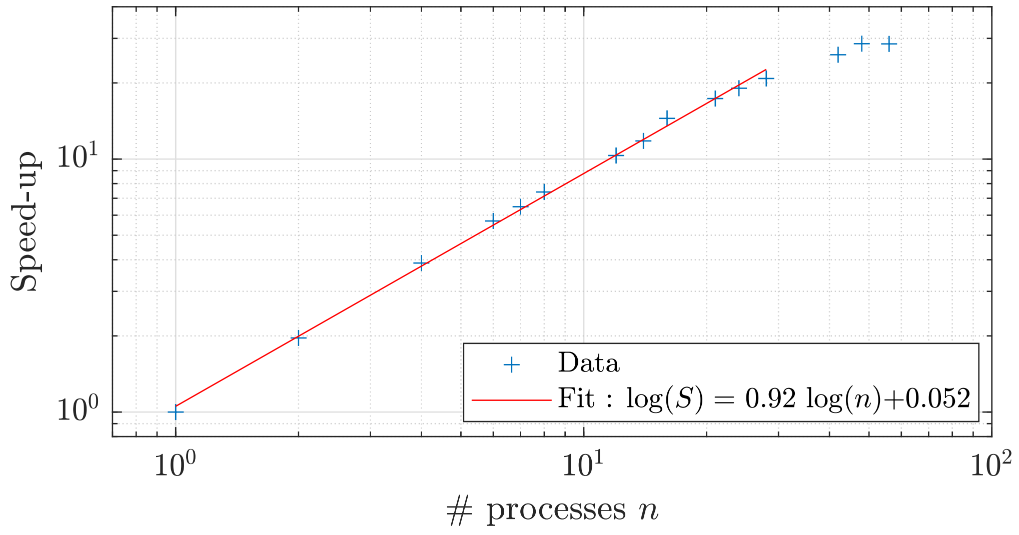

Figure 2 – Strong scaling using Intel-MPI

Intel's compiler provides a slightly lower speed-up. However, this effect is compensated by the fact that Intel executes much faster on a single process, though this not visible on figure (2). The graph shows that the effect of inter-node communication is almost negligible. It seems like Intel's compiler is much better optimised in this regard.

3 Weak scaling

Weak scaling is obtained by varying the number of process with a fixed workload/process. The parallel efficiency is defined by $E = t'_1/t'_i$, where t'_i is the execution time per step for a fixed workload/process. The ideal case would be $t'_i = t'_1 \quad \forall i$, representing an efficiency of 1. Note that since the problem size is proportional to N^2, N is multiplied by $\sqrt{n}$ in order to keep the workload fixed.

Due to technical reasons, the program requires $N \mod n = 0$. In order to get more data points, N was adjusted to the closest value that is divisible by n. This means that the workload is not exactly equal for different number of processes (but it is close).

Figure (3) shows the efficiency E as a function of n. These results were obtained using gcc/MVAPICH2 only.

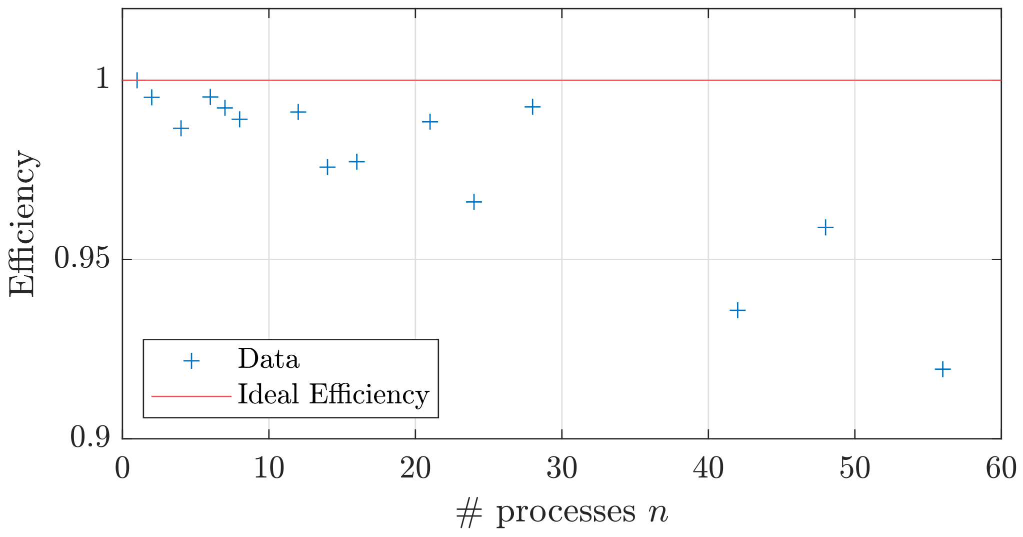

Figure 3 – Weak scaling

As expected, the efficiency decreases with an increasing number of processes. Again, this is due to communications. Indeed, a single process does not have to communicate at all, while the total communication increase with the number of processes. As with strong scaling, the efficiency seems to further decrease when a second node is used.

The noise appearing in the data could be due to the requirement $N \mod n = 0$ discussed above.

4 Additional note

This problem theoretically allows to hide communication latency with some computations. Indeed, one can use non-blocking communication which could happen while the values from the inner grid (the part of the grid that do not use ghost lines) are computed. Once the message is received, the program would compute values that require the ghost lines.

This was implemented but did not reduce the execution time (it actually increased it) using gcc/MVAPICH2, though it did help on Intel's compiler. The values presented earlier were obtained with blocking communications (MPI_Isend/MPI_Irecv and MPI_Wait right after) since the advantage on Intel was discovered late in the project.