Parallelisation of Conjugate Gradient using MPI

Ludek Cizinsky (student ID: 377297)
`ludek.cizinsky@epfl.ch`[*]

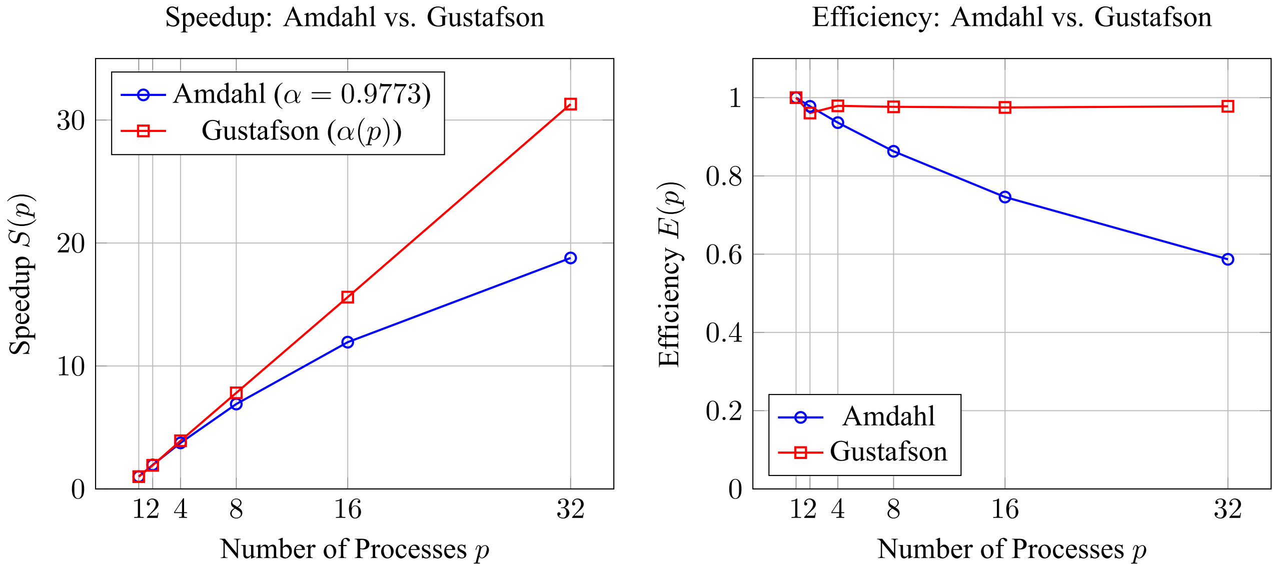


Figure 1. Comparison of predicted speedup and efficiency using Amdahl's and Gustafson's laws.

1 Matrix collection and profiling

My very first step was to collect matrices which I will later use for my strong and weak scaling experiments. Given the assumptions behind the Conjugate Gradient (CG) algorithm, I focused on symmetric, positive semi-definite matrices which are in addition solvable by CG. Given these search criteria, I went over the matrices available at SuiteSparse Matrix collection and selected the candidates displayed in the Table 1.

Name	**N**	**nnz**	**p**	**nnz/p**	T_1	**Iters**	α
`Pres_Poisson`	14,822	715,804	1	715,804	4 s	5,509	0.9319
`thermomech_dM`	204,316	1,423,116	2	711,558	1 s	164	0.9215
`pdb1HYS`	36,417	4,344,765	4	1,086,191	78 s	14,627	0.9722
`nd6k`	18,000	6,897,316	8	862,165	112 s	12,498	0.9733
`crankseg_1`	52,804	10,614,210	16	663,388	85 s	6,172	0.9733
`crankseg_2`	63,838	14,148,858	32	442,152	155 s	8,470	0.9773

Table 1. Matrix statistics and sequential performance results for selected matrices.

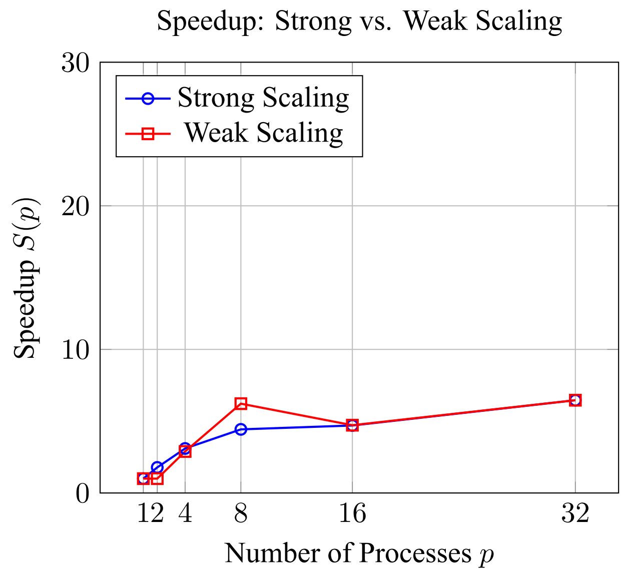
I selected `crankseg_1` to run my `perf` profiler on, since it runs long enough and includes a reasonable number of non-zero entries. The `perf` profiler collected a total of 372k CPU cycles, from which 98.63% correspond to the `CGSolverSparse::solve` method, which is dominated by `MatrixCOO::mat_vec`, accounting for 97.33% of CPU cycles. Looking more closely into the sparse matrix-vector multiplication method, the main workload comes from the loop over non-zero entries of the sparse matrix. For each such entry, the function performs two multiplications and additions. While these operations are trivial, the main cost arises from fetching the matrix and vector entries from memory. Since there is a clear potential for parallelisation, I define α as the proportion of CPU cycles spent in `MatrixCOO::mat_vec`.

[*]All experiments were run on a 2-node setup using Lenovo SR630 V2 machines (Intel Xeon 8360Y @ 2.40 GHz, 492 GB RAM, 3.2 TB NVMe), with 1 MPI task and 16 CPU threads per node. I made use of the EPFL's Jed cluster.

α Estimation and Predicted Speedup and Efficiency: Amdahl vs. Gustafson

To estimate the parallel fraction α, I used the largest matrix from Table 1, `crankseg_2`, which offers the highest computational load and is suitable for strong scaling analysis. For weak scaling, I relied on all listed matrices, each representing a larger problem, measured by the number of non-zero entries, as the number of processes increases. Using these estimates, I computed theoretical speedup and efficiency using both Amdahl's Law Amdahl (1967) and Gustafson's Law Gustafson (1988). As shown in Figure 1, Amdahl's Law predicts diminishing returns with more processors due to its assumption of a fixed problem size. Even with infinite processors, speedup is bounded by the serial portion $(1-\alpha)$ of the code. Gustafson's Law, on the other hand, scales the problem with the number of processors and therefore yields significantly higher speedup and efficiency estimates. This model better reflects practical high-performance computing workloads, where problem sizes grow with available resources. As a result, the impact of the serial portion is reduced, leading to more optimistic and often more realistic parallel performance predictions.

Weak and strong scaling in practice



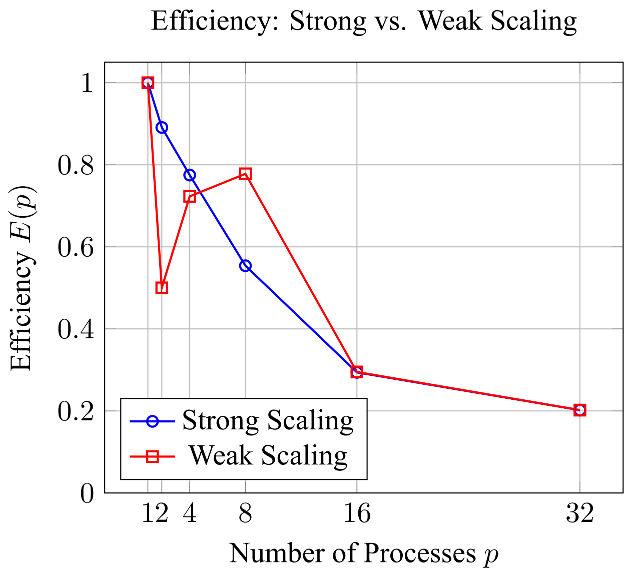


Figure 2. Comparison of observed speedup and efficiency under strong and weak scaling experiments.

Figure 2 reveals that the predicted theoretical speedups in Figure 1 were overly optimistic, especially as the number of processes increases to p = 4, 8, and 16. A likely explanation is the increasing communication overhead introduced by MPI, which limits performance gains beyond the initial speedup.

Further, while weak scaling assumes a constant workload per processor, in practice this condition was not fully met. Ideally, based on the baseline matrix with 715,804 non-zero entries, each processor should handle a comparable amount of work. However, the largest matrix used, `crankseg_2`, provides only 442,152 non-zeros per processor at $p = 32$. Despite the reduced workload per processor, the observed speedup did not improve accordingly. This suggests that the relative cost of serial computation and communication increased, making them more dominant in the overall runtime. Additionally, since these matrices represent problems of varying complexity, solve time may be a better proxy for workload than non-zero count alone.

Finally, the declining efficiency in weak scaling further highlights underutilization of available resources. While my parallelisation strategy focuses on matrix-vector multiplication, other parts of the CG algorithm remain serial. In particular, each iteration involves a bottleneck at the root node, which aggregates partial results and computes the global update. Addressing this bottleneck would be my next improvement step.

References

Gene M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, 1967.

John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.