Parallelizing the Conjugate Gradient Method with MPI

Aaron Dinesh

April 16, 2025

A project submitted in fulfilment
of the requirements for MATH-454

Declaration

I hereby declare that this work is fully my own.

Signed: Aaron Dinesh Date: April 16, 2025

1.1 Introduction

In this assignment, a serial implementation of the Conjugate Gradient Method (CGM) was provided, which required parallelization. In order to parallelize this implementation OpenMPI and the MPI parallelization paradigm was used.

1.2 Serial Analysis

The serial implementation of the CGM was evaluated using the provided `lap2D_5pt_n1000 .mtx` matrix. This matrix was chosen as its large size allowed for obvious parallelization opportunities to be revealed. The `perf` profiling tool was used to analyse the serial code's execution time. From the profiling results, the `mat_vec` function was identified as the primary computational bottleneck, accounting for approximately 86.27% of the total execution time. Consequently, the serial fraction of the program was estimated to be 13.73%. The testing for the serial and parallel code were performed on the Jed Cluster with and Intel Xeon Platinum 8360Y CPU, running at 2.40 GHz.

These values along with Amdahl's and Gustafson's laws for strong and weak scaling were used to calculate a theoretical upper bound on the possible speed-ups that could be achieved using parallelization. The speed-ups were calculated using the following equations [1]:

$$\text{Amdahl's Law (Strong Scaling)} = \frac{1}{(1-\alpha) + \frac{\alpha}{p}}$$

Here α is the fraction of the code that can be parallelized and p is the number of processors. Gustafson's equations are as follows:

$$\text{Gustafson's Law (Weak Scaling)} = (1-\alpha) + \alpha p$$

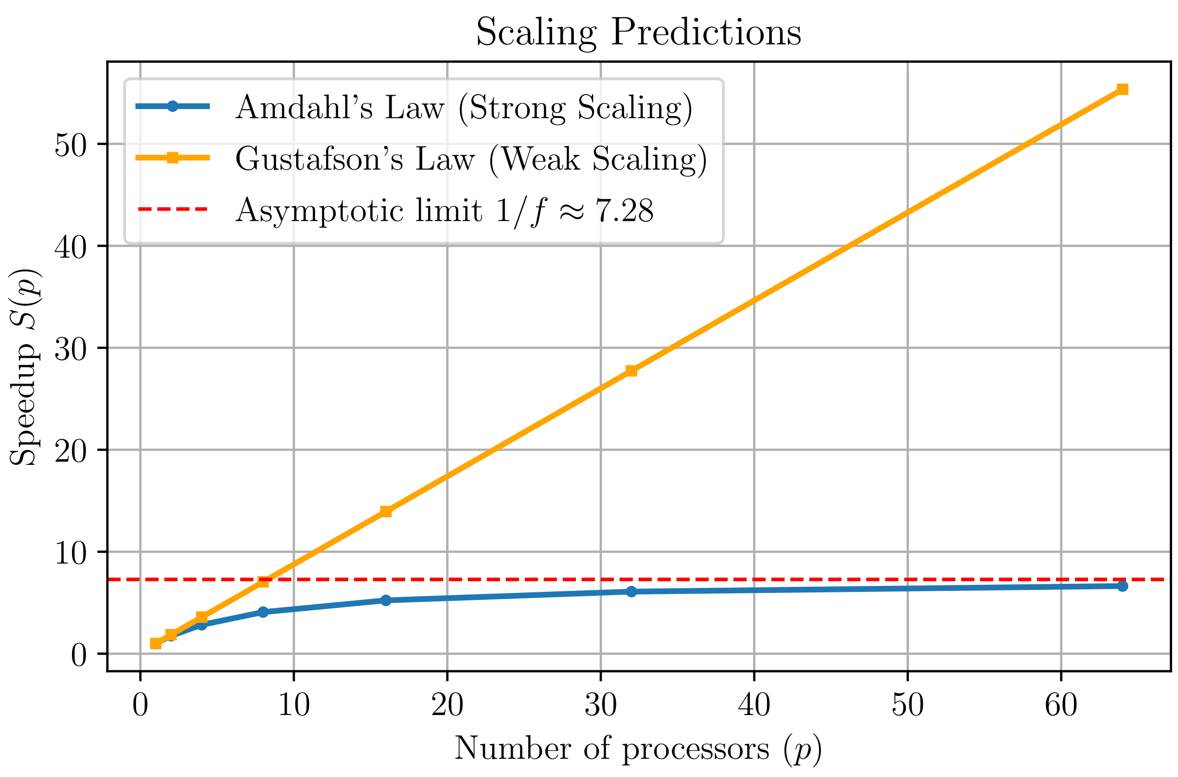

Figure 1: Strong and Weak Scaling Analysis using Amdahl's and Gustafson's Laws

1.3 Parallel Modifications

In order to parallelize the code with the least amount of communications, it was decided that the best course of action would be to read in the matrix on processor 0 and then block distribute the row index, column index and value array to each processor. Doing this when reading the matrix would implicitly parallelize the `mat_vec` function and would reduce the communication needed to requiring just one `MPI_Allgather` after each call to `mat_vec`, rather than a `MPI_Scatterv` and a `MPI_Allgather` before and after the `mat_vec` calls. MPI message packing and unpacking were also used to speed up this initial data transfer. This was enough to show some initial improvements over the serial execution of the code, however, there was a lot more that could be done in the `CGSolverSparse::solve()` function to further improve the parallelization of the code. Further optimization was achieved through the use non-blocking transfers such as `MPI_Iallgatherv` as well as combined communications such as `MPI_Reduce_scatter`. Using these combined communications allows the MPI compiler to perform further optimizations when compiling the code. I also made use of cblas operations such as `cblas_dcopy`, `cblas_daxpy` and `cblas_ddot` to make use of the optimized implementations in the cblas library.

1.4 Parallel Performance

It was expected that the modifications introduced to the codebase would result in an improvement over sequential runtimes. However, experimental results did not support this assumption. During testing, the parallel implementation demonstrated poor scaling behaviour on the Jed

Cluster. To illustrate this, strong and weak scaling experiments were conducted. For Strong Scaling, the matrix `lap2D_5pt_n1000.mtx` and `lap2D_5pt_n200.mtx` were used, with the processor counts ranging from 1 to 64 in powers of 2. The upper limit of 64 was selected based on the fact that no meaningful scaling was observed past this point. In most configurations, the parallel solver performed comparably to the serial implementation however in some cases, performance was slightly worse. The speed-up in the strong scaling case was calculated using the following formula:

$$S(p) = \frac{T_1}{T_p}$$

where the subscript denotes the number of processors used. For the weak scaling experiments, the run time of a single processor was tested with the `lap2D_5pt_n100.mtx` matrix. Then the processor count was raised in powers of 2 and they were tested against the other matrices provided. From these experiments the runtime and the iteration count to convergence were calculated. The speed-up in the weak scaling case is then calculated as follows:

$$S(p) = \frac{T_1/I_1}{T_p/I_p}$$

Where I_p is the iteration count to convergence for p processors. Since weak scaling requires the work done at each test to be constant, the runtime was divided by the iteration count. The results form these experiments are shown in the graphs below.

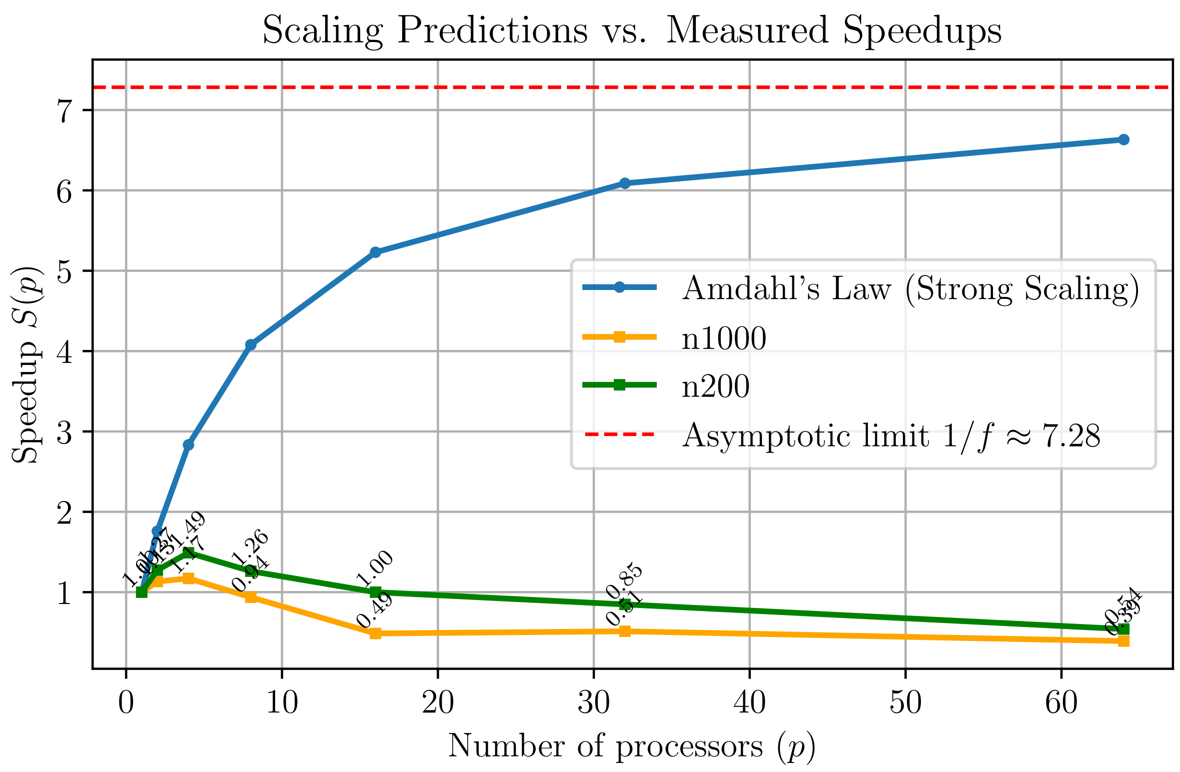

Figure 2: Strong Scaling Analysis

As can be seen from the graph above, minor performance gains can be achieved by switching from a serial code to a few parallel processes. However, past 8 processors no further improvements are observed. A similar story can also be seen when performing a weak scaling analysis, as can be seen in the graph below.

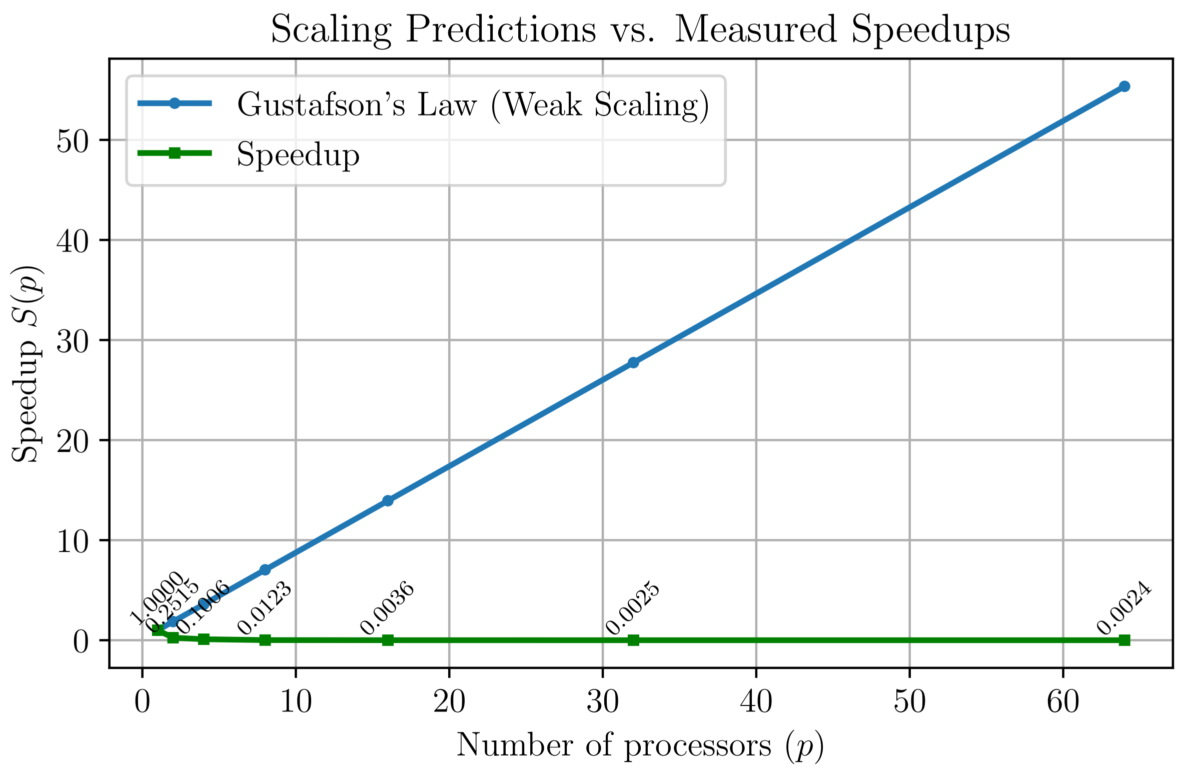

Figure 3: Weak Scaling Analysis

The raw results of the strong and weak scaling analysis can be seen in the tables below.

# Procs Matrix Size	1	2	4	8	16	32	64
n200	1.00	1.13	1.17	0.94	0.49	0.51	0.39
n1000	1.00	1.27	1.49	1.26	1.00	0.85	0.54

Table 1.1: Strong scaling speedup for different matrix sizes

# Procs	1	2	4	8	16	32	64
speedup	1.	0.25147243	0.10058002	0.0122578	0.00355457	0.00252197	0.00238294

Table 1.2: Weak scaling speedup

It is suspected that the small initial performance gains observed in the strong and weak scaling analysis are primarily due to the distribution of the sparse matrix across all participating processors, allowing for some performance gains in the `mat_vec` function. However, this advantage is quickly overshadowed by the substantial cost incurred by the collective communication required to calculate the full matrix vector product. As the dimensionality of the system increases the volume of data involved in these collectives grows correspondingly, leading to significant communication latency that dominates in the overall runtime, especially when considering strong scaling.

To address this issue, several mitigation strategies were explored. The first made use of non-blocking communications such as `MPI_Iallreduce`, to allow for independent work to be completed while the communication continued in the background. While this yielded some improvements, the gains were minor due to the relatively tight coupling between global reductions and subsequent computations. The second made significant use of optimized BLAS functions. The reasoning behind this was that the BLAS functions were optimized for operations on large vectors and matrices. However, as was the case before, the performance gains were minor, yielding further credibility to the idea that that main bottleneck is the communication overhead.

A more effective solution, however, would be to make use of modified CG algorithms that are better suited for parallel execution. For example the Global Reduction pipelining CG algorithm proposed by Cool and Cornelis et al. [2] reformulates the classic CG iteration to allow for multiple operations to be overlapped with global communication. They introduce an auxiliary Krylov basis and structure the algorithm to delay the use of reduction results by ℓ iterations, allowing them to overlap ℓ operations with a single `MPI_Iallreduce` call. Through the use of this algorithm, the team demonstrates strong scaling results up to 1024 nodes with 16 MPI ranks per node on the US Department of Energy's NERSC "Cori" machine.

Bibliography

[1] D. Mishra, "A deep dive into amdahl's law and gustafson's law," 2023. [Online]. Available: https://hackernoon.com/a-deep-dive-into-amdahls-law-and-gustafsons-law

[2] S. Cools, J. Cornelis, P. Ghysels, and W. Vanroose, "Improving strong scaling of the conjugate gradient method for solving large linear systems using global reduction pipelining," 2019. [Online]. Available: https://arxiv.org/abs/1905.06850