Parallelizing the Conjugate Gradient Method with MPI

Aaron Dinesh

April 13, 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 13, 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*R* Xeon*R* 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:

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

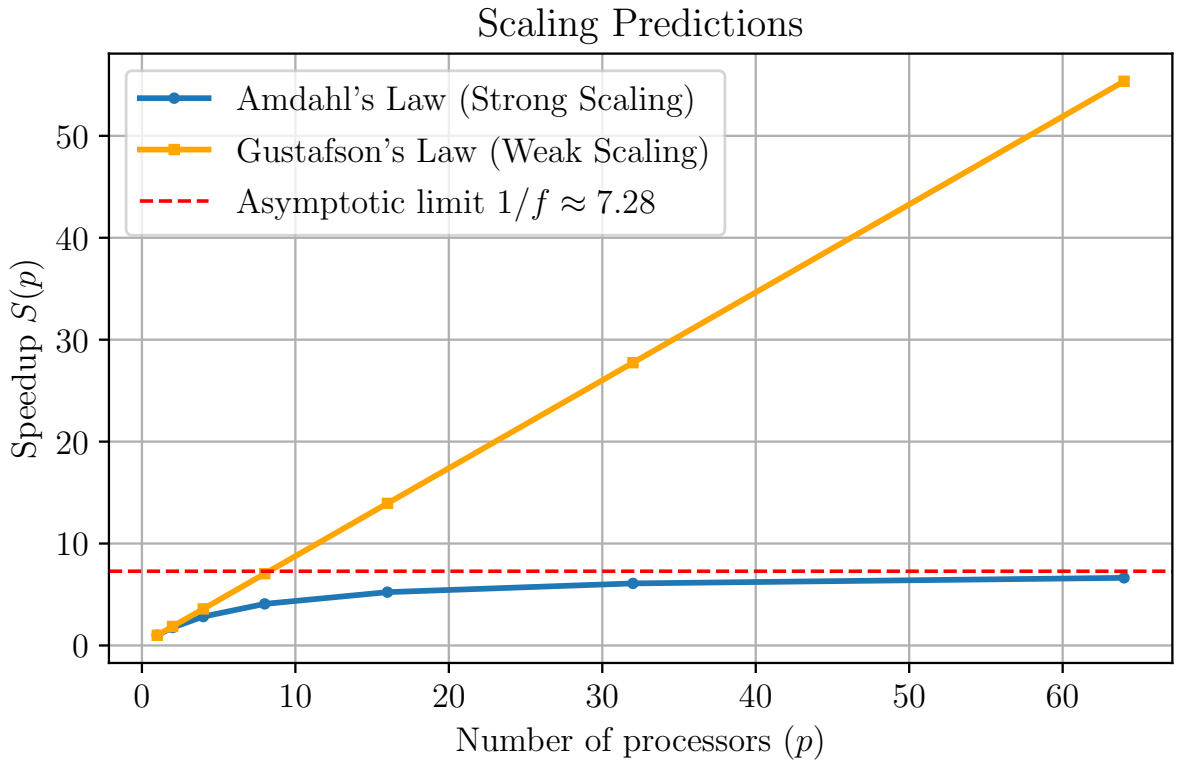

Figure 1: Strong and Weak Scaling Analysis using Amdahl's and Gustafson's Laws

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

I had hoped all the changes that I had implemented would have significantly improved the runtime of the code over the serial implementation. However, this was not the case. In my testing I noticed extremely poor scaling in the parallel case. First I will showcase the perfor-

mance graphs and then discuss the possible reasons and fixes for this.

For the strong scaling analysis I used the `lap2D_5pt_n1000.mtx` matrix and the `lap2D_5pt_n200.mtx` matrix and then varied the processor count from 1 to 64. I decided to stop at 64 since the timing results did not indicate any type of scaling. Most of the time, the parallel code took the same time as the serial code and in some cases did slightly worse as can be seen in the graph below.

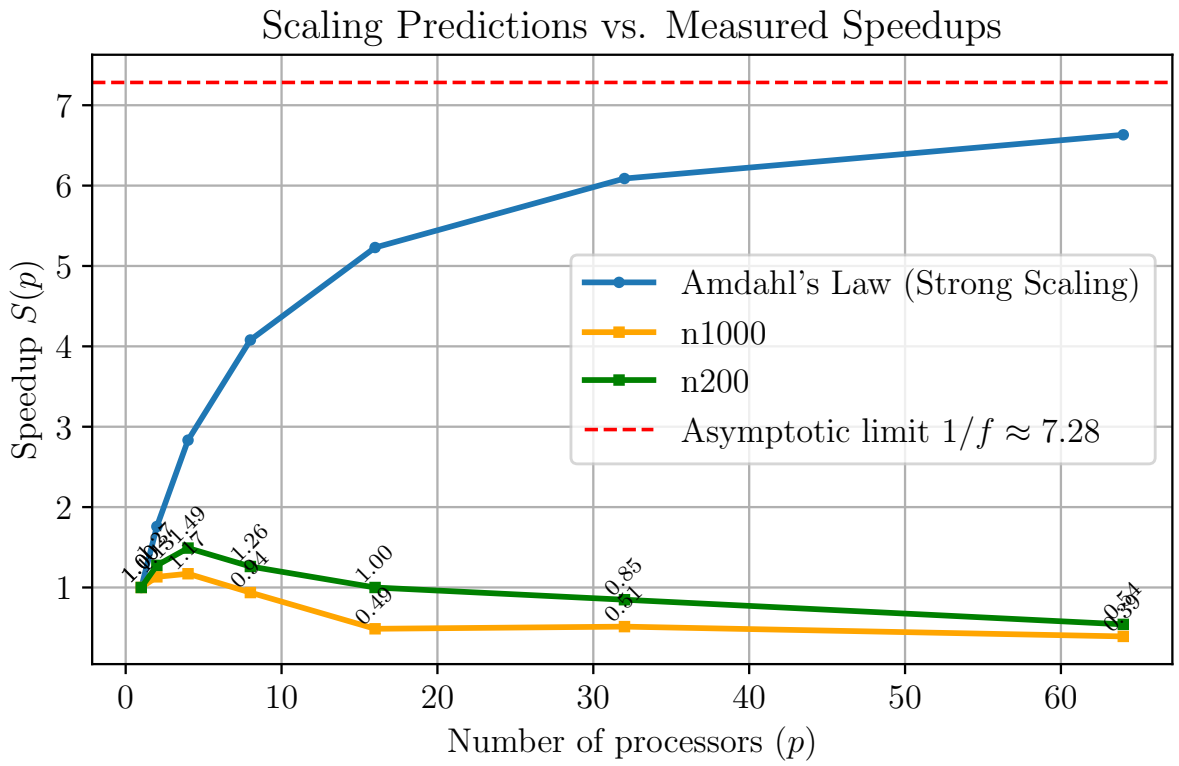

Figure 2: Strong Scaling Analysis

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.

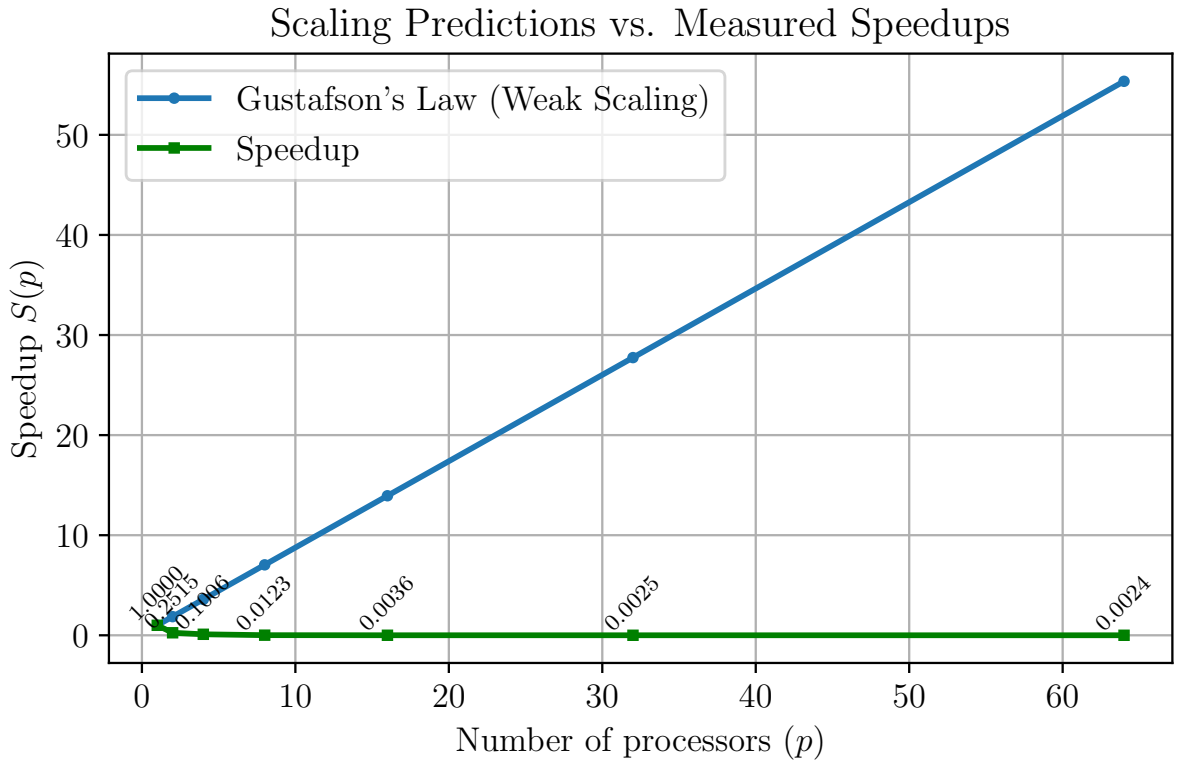

Figure 3: Weak Scaling Analysis

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

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