Görkem Kadir Solun 22003214

CS426 Project 2 - Parallel Conjugate Gradient

Implementation and Setup

The serial implementation begins by parsing command-line arguments to identify the input and (optionally) output filenames, then reads the matrix dimension N followed by the N×N entries of A and the N-vector b. It allocates contiguous arrays for all vectors and the dense matrix, initializing the solution guess x to zero and setting the residual r=b. The core of the code executes the standard Conjugate Gradient loop: it computes the matrix–vector product Ap, forms the step size α via the inner product of p and Ap, updates x and r, checks the Euclidean norm of the new residual for convergence, and then constructs the new search direction p using the ratio of squared norms. Timing is measured with `clock()` around the iterative phase.

The MPI-parallel version wraps a similar algorithm in distributed form. After `MPI_Init`, rank 0 reads and verifies the input, partitions rows evenly among the worker ranks, and sends each a block of A and b. All processes synchronize, and timing begins with `MPI_Wtime()`. Each worker maintains its local piece of x, r, and p, and in each iteration, they exchange partial p vectors via nonblocking P2P calls to assemble the full vector needed for the local sparse-matrix–vector multiply. They then compute local contributions to the dot products, combine them with `MPI_Allreduce`, update their local solution and residual, and rederive the new search direction. Once converged, all workers send their segment of x back to the master, which gathers the full solution, stops the timer, writes the output, and finalizes MPI.

Strong scaling: $N = 2^{10}\ and\ N = 2^{11}$ where N is the problem size describing the matrix which has a size of NxN and vector which has a size of N.

$Cores \in \{1, 2^4 + 1, 2^5 + 1, 2^6 + 1, 2^7 + 1, 2^8 + 1\}$

Weak scaling: The first one is done by having the element count per processor equal and the second one is done by having the row count per processor equal.

Table/Figure 3: $(N, Cores) \in \{(2^7, 2^2 + 1), (2^8, 2^4 + 1), (2^9, 2^6 + 1), (2^{10}, 2^8 + 1)\}$

Table/Figure 4: $(N, Cores) \in \{(2^{11}, 2^8 + 1), (2^{10}, 2^7 + 1), (2^9, 2^6 + 1), (2^8, 2^5 + 1), (2^7, 2^4 + 1)\}$

Results

Diminishing returns beyond 16 cores, meaning communication latency and synchronization dominate, causing execution time to rise for cores ≥ 32.

Table and Figure 1: Strong Scaling Curve with $N = 2^{10}$, $Cores \in \{1, 2^4 + 1, 2^5 + 1, 2^6 + 1, 2^7 + 1, 2^8 + 1\}$

Core Count	Elapsed Time (s)
1	0.004023
16	0.001184
32	0.002960

64	0.012656
128	0.056406
256	0.037674

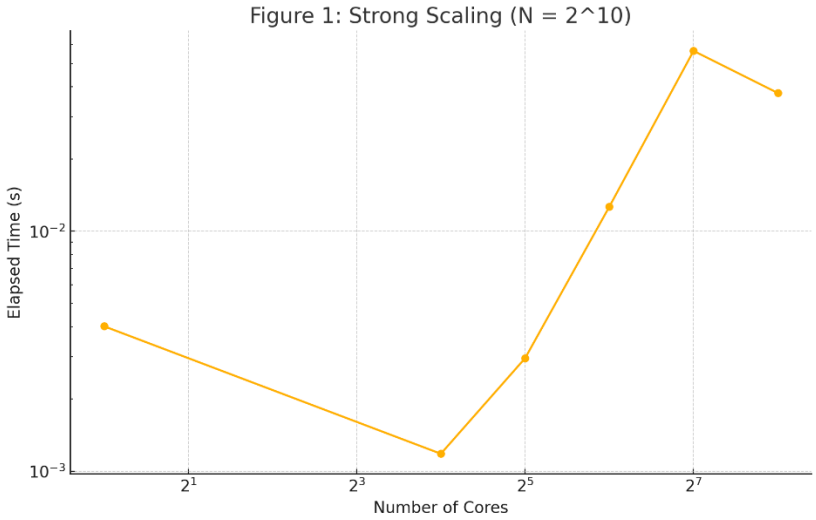


Table and Figure 2: Strong Scaling Curve with $N = 2^{11}, Cores \in \{1, 2^4 + 1, 2^5 + 1, 2^6 + 1, 2^7 + 1, 2^8 + 1\}$

Core Count	Elapsed Time (s)
1	0.017609
16	0.002168
32	0.004520
64	0.013138
128	0.039841
256	0.036201

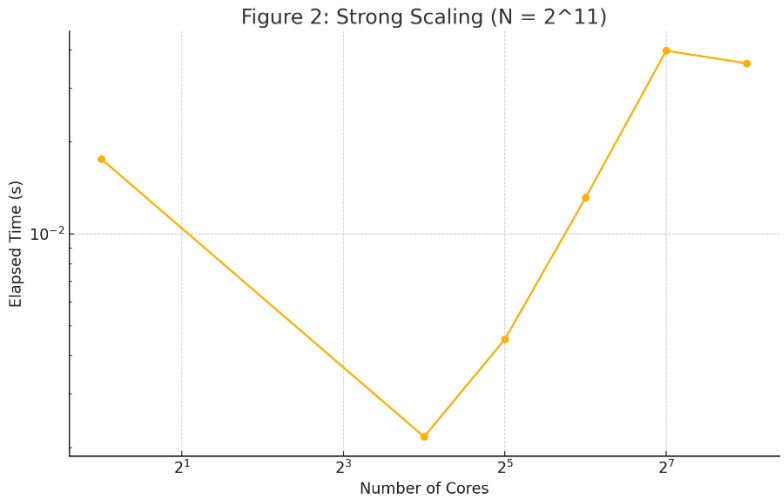


Table and Figure 3: Weak Scaling Curve with $(N, Cores) \in \{(2^7, 2^2 + 1), (2^8, 2^4 + 1), (2^9, 2^6 + 1), (2^{10}, 2^8 + 1)\}$

N, Core Count	Elapsed Time (s)
$(2^7, 2^2 + 1)$	0.000167
$(2^8, 2^4 + 1)$	0.000821
$(2^9, 2^6 + 1)$	0.012789
$(2^{10}, 2^8 + 1)$	0.078752

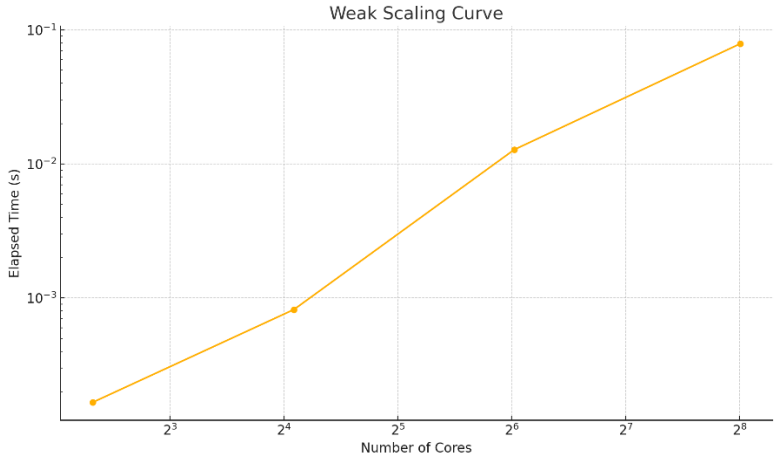
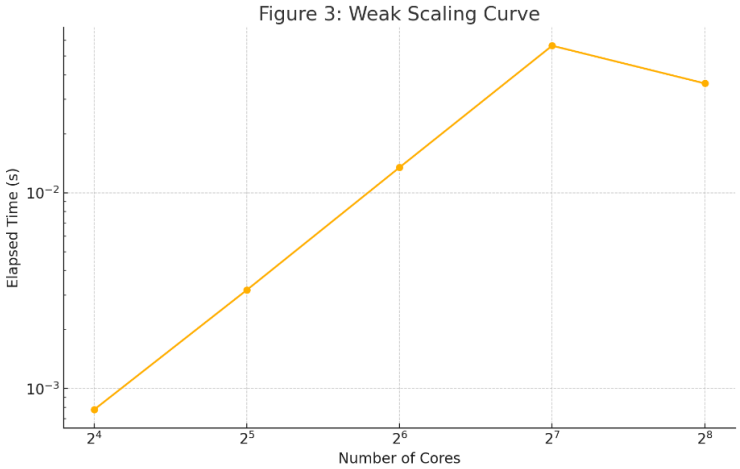


Table and Figure 4: Weak Scaling Curve with $(N, Cores) \in \{(2^{11}, 2^8 + 1), (2^{10}, 2^7 + 1), (2^9, 2^6 + 1), (2^8, 2^5 + 1), (2^7, 2^4 + 1)\}$

N, Core Count	Elapsed Time (s)
$(2^7, 2^4 + 1)$	0.000779
$(2^8, 2^5 + 1)$	0.003183
$(2^9, 2^6 + 1)$	0.013461
$(2^{10}, 2^7 + 1)$	0.056406
$(2^{11}, 2^8 + 1)$	0.036201



Ideal weak scaling keeps time nearly constant; here it increases from 0.78 ms (16 cores) to 56.4 ms (128 cores), then slightly improves at 256 cores (36.2 ms). The steep rise indicates that as the problem/processor ratio stays fixed, communication (point-to-point exchanges and global reductions) overhead grows super linearly.

Discussion and Conclusion

In strong scaling, for both $N = 2^{10}$ and 2^{11}, best speedup occurs at 16-32 cores. Beyond 32 cores, elapsed time increases markedly, communication/latency dominates over per-core computation. The overhead of exchanging vector segments and reductions outweighs local SpMV gains.

Ideal weak scaling maintains constant runtime as both problem size and cores increase. Here, elapsed time grows over an order of magnitude from 16 to 256 cores, indicating significant overhead in distributed matrix-vector multiple times and global reductions. Instead of remaining constant, runtime increases by nearly three orders of magnitude from 4 → 256 cores. As cores rise, each MPI rank handles roughly constant work, but exchanging the full vector across more peers P2P and performing global reductions becomes increasingly expensive. Collective operations (e.g. `MPI_Allreduce`) scale roughly as $O(logP)$, and the overhead of barriers at each CG iteration exacerbates the slowdown. In current form, the parallel CG is not weakly scalable beyond small core counts. Future optimizations like overlap of communication with computation, asynchronous collectives, or pipelining are needed to flatten this curve.

P2P communication during SpMV and `MPI_Allreduce` for dot-products contributes to the non-ideal scaling. For small to moderate core counts, computation dominates; at high counts, communication latency and load imbalance are limiting factors. Additionally, the synchronization barriers inserted around timing exacerbate the effect, so that beyond a certain scale, the cost of coordination eclipses any per-core workload reduction.

The MPI-parallel CG solver shows modest strong scaling up to 32 cores. However, communication overhead severely limits both strong and weak scaling at larger scales. Future optimizations might include overlapping communication and computation, using nonblocking collectives, exploring hybrid `MPI+OpenMP` to reduce inter-node messages, and using topology-aware process mapping to minimize network hops.