CSE6126 Parallel Scientific Computing

Jeongbin Jo[1, *]

[1]*Department of Physics, Yonsei University, Seoul, 03722, Republic of Korea*

(Dated: July 10, 2026)

This report collects coursework for CSE6126 Parallel Scientific Computing. HW1 develops serial iterative Poisson solvers and establishes baseline accuracy and iteration-cost measurements. HW2&3 then optimize, profile, and OpenMP-parallelize the conjugate-gradient solver. HW4 extends that CG solver to distributed memory with MPI, comparing global gather against neighbor halo exchange under one-dimensional domain decomposition.

CONTENTS

I. INTRODUCTION

This report documents the numerical methods and programming assignments for CSE6126 Parallel Scientific Computing. HW1 focuses on serial iterative methods for a manufactured two-dimensional Poisson problem, including Jacobi, Gauss–Seidel/SOR, and conjugate-gradient solvers. HW2&3 build on that CG baseline through loop optimization, CSR sparse storage, profiling of hot kernels, and OpenMP parallelization. HW4 then moves to distributed memory: a one-dimensional MPI domain decomposition of CG is implemented with both global gather and neighbor halo exchange, and the resulting communication costs and parallel efficiencies are compared across grid sizes and process counts.

* jeongbin033@yonsei.ac.kr

II. HOMEWORK SOLUTIONS

III. HW1: SERIAL POISSON SOLVER

A. Problem Statement

The first assignment builds serial iterative solvers for the two-dimensional Poisson equation

$$\nabla^2 u(x, y) = f(x, y), \qquad (x, y) \in [0, 1] \times [0, 1], \tag{1}$$

with the manufactured source term

$$f(x, y) = \sin(\pi x) \cos(\pi y). \tag{2}$$

The corresponding analytical solution is

$$u_a(x, y) = -\frac{1}{2\pi^2} \sin(\pi x) \cos(\pi y), \tag{3}$$

which is used both to impose exact Dirichlet boundary values and to measure numerical error.

Two boundary-condition cases are considered. In the first case, homogeneous Dirichlet conditions are applied on the left and right boundaries, while homogeneous Neumann conditions are applied on the bottom and top boundaries:

$$u(0,y) = u(1,y) = 0, \qquad \frac{\partial u}{\partial n} = 0 \quad \text{on } y = 0, 1. \tag{4}$$

In the second case, Dirichlet values from (3) are imposed on all four sides of the square domain.

B. Finite-Difference Discretization

The domain is discretized on a uniform grid with spacing $h = 1/(N-1)$. For an interior grid point (x_i, y_j), the Laplacian in (1) is approximated by the standard five-point stencil [1, 2]

$$\frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2} = f_{i,j}. \tag{5}$$

Equivalently, the fixed-point form used by stationary iterations is

$$u_{i,j} = \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - h^2 f_{i,j}\right). \tag{6}$$

For the mixed boundary case, the homogeneous Neumann condition is implemented through mirrored boundary values so that the normal derivative vanishes at $y = 0$ and $y = 1$. This boundary treatment is simple and stable for a serial baseline, but its boundary truncation error is larger than the interior five-point stencil, so the mixed case is expected to have a larger global error than the exact-Dirichlet case.

C. Iterative Methods

Three serial solvers are implemented and compared:

a. Jacobi method. The Jacobi method updates every interior value from the previous iteration only. It is simple and exposes a useful baseline, but it typically requires many iterations because information propagates slowly across the grid. For elliptic problems, low-frequency error modes are damped slowly by Jacobi iteration, so the iteration count grows rapidly as h decreases [3].

b. Gauss–Seidel method with SOR. The Gauss–Seidel sweep immediately reuses newly updated values. Successive over-relaxation (SOR) further modifies the update by

$$u_{i,j}^{(k+1)} = (1-\omega)u_{i,j}^{(k)} + \omega\tilde{u}_{i,j}^{(k+1)}, \tag{7}$$

where $\tilde{u}_{i,j}^{(k+1)}$ is the Gauss–Seidel candidate from (6). The relaxation factor ω is chosen in the range $1 < \omega < 2$, with the final value selected empirically from convergence tests [3, 4]. SOR therefore keeps the same local sweep structure as Gauss–Seidel but accelerates convergence by extrapolating along the Gauss–Seidel correction direction.

c. Conjugate-gradient method. The conjugate-gradient (CG) method solves the sparse linear system $A\mathbf{u} = \mathbf{b}$ generated by (5). In the implementation, the system is written with the positive operator $-\nabla_h^2$, so the matrix is symmetric positive definite for the exact-Dirichlet case and remains well posed for the mixed case because the x-direction Dirichlet boundaries remove the constant null mode. CG minimizes the quadratic energy error over expanding Krylov subspaces and is therefore expected to converge substantially faster than stationary iterations for this sparse elliptic system [3, 5].

D. Evaluation Metrics

For each method and boundary-condition case, the main computational-cost metric is the number of iterations required to reach a prescribed relative residual tolerance of 10^{-10}. Numerical accuracy is measured by the discrete l_2-error

$$\|e\|_2 = \frac{1}{N}\left[\sum_{i,j=1}^{N}\left(u_{i,j}^{n} - u_{a,i,j}\right)^2\right]^{1/2}, \tag{8}$$

where $u_{i,j}^{n}$ is the numerical solution and $u_{a,i,j}$ is the analytical solution evaluated at the same grid point. The calculations are repeated for $N = 32, 64, 128$ to examine grid dependence in both cost and accuracy. The SOR relaxation factor is fixed at $\omega = 1.85$ for the present baseline runs.

E. Serial Results

Table I summarizes the iteration counts and wall-clock times. The stationary Jacobi method is consistently the most expensive method: for the mixed Dirichlet–Neumann case, increasing N from 32 to 128 raises the Jacobi count from 4.19×10^3 to 7.41×10^4. SOR reduces the iteration count by roughly one order of magnitude at the same residual tolerance. CG is the most efficient method in iteration count, requiring fewer than 100 iterations on all tested grids. At $N = 128$, CG is about $6.80/3.73 \times 10^{-3} \approx 1.8 \times 10^3$ times faster than Jacobi for the mixed case and about $2.83/3.71 \times 10^{-3} \approx 7.6 \times 10^2$ times faster for the exact-Dirichlet case.

Table III lists the final residuals and discrete l_2-errors. For a fixed grid and boundary case, all three methods converge to the same discretized solution to the reported accuracy, so the l_2-error is essentially method-independent. This confirms that the solver choice primarily changes the computational path to the discrete solution, not the final discretization error once the residual tolerance is tight enough. The exact-Dirichlet case shows the expected grid refinement trend, with the error decreasing from 7.19×10^{-6} at $N = 32$ to 4.38×10^{-7} at $N = 128$. The mixed boundary case also improves with refinement, but its error is much larger and decreases more slowly because the mirrored Neumann boundary treatment dominates the global error.

The observed spatial order is quantified from successive refinements by

$$p = \frac{\log\big(E(h_1)/E(h_2)\big)}{\log\big(h_1/h_2\big)}, \tag{9}$$

where $E(h)$ is the discrete l_2-error on spacing $h = 1/(N-1)$. Table II and Fig. 1 show that the exact-Dirichlet errors yield $p \approx 1.98$–1.99, confirming the expected second-order slope of the interior five-point stencil. In contrast, the mixed Dirichlet–Neumann case yields $p \approx 1.02$–1.04, i.e. first-order global accuracy, consistent with the first-order mirrored Neumann closure used on $y = 0, 1$.

The same trends are visualized in Fig. 2. The iteration and runtime plots show the widening cost gap between the

TABLE I. Serial performance for HW1 Poisson solvers. All runs use relative residual tolerance 10^{-10}; SOR uses $\omega = 1.85$.

N	boundary case	method	iterations	time (s)
32	mixed D/N	Jacobi	4,188	2.17×10^{-2}
32	mixed D/N	SOR	266	1.82×10^{-3}
32	mixed D/N	CG	18	3.70×10^{-5}
32	exact D	Jacobi	1,848	9.87×10^{-3}
32	exact D	SOR	179	1.22×10^{-3}
32	exact D	CG	18	3.76×10^{-5}
64	mixed D/N	Jacobi	17,926	4.03×10^{-1}
64	mixed D/N	SOR	1,316	4.25×10^{-2}
64	mixed D/N	CG	44	3.91×10^{-4}
64	exact D	Jacobi	7,649	1.71×10^{-1}
64	exact D	SOR	666	2.17×10^{-2}
64	exact D	CG	44	3.97×10^{-4}
128	mixed D/N	Jacobi	74,069	6.80
128	mixed D/N	SOR	5,339	7.45×10^{-1}
128	mixed D/N	CG	92	3.73×10^{-3}
128	exact D	Jacobi	31,091	2.83
128	exact D	SOR	2,804	3.84×10^{-1}
128	exact D	CG	96	3.71×10^{-3}

TABLE II. Observed convergence orders from successive grid refinements (CG errors; identical for Jacobi and SOR at the same residual tolerance).

boundary case	N range	$E(h_1)/E(h_2)$	h_1/h_2	p
exact D	$32 \to 64$	4.06	2.03	1.98
exact D	$64 \to 128$	4.03	2.02	1.99
mixed D/N	$32 \to 64$	2.09	2.03	1.04
mixed D/N	$64 \to 128$	2.04	2.02	1.02

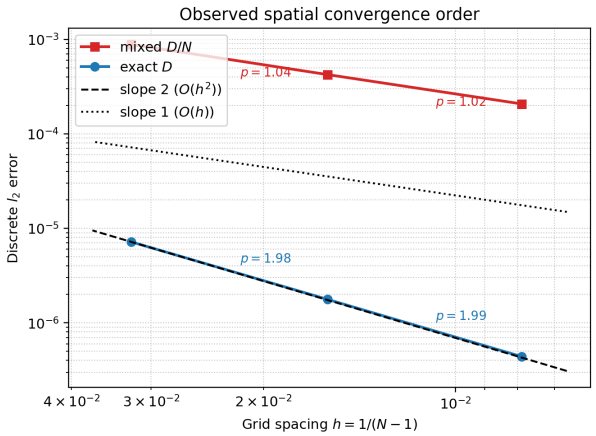

FIG. 1. Log–log plot of discrete l_2-error versus grid spacing h. The exact-Dirichlet data follow the reference slope of 2, while the mixed boundary data follow a slope near 1. Annotated values are the observed orders from (9).

stationary iterations and CG as the grid is refined. The error plots show that the exact-Dirichlet case follows the expected refinement trend much more cleanly than the mixed boundary case. Since the three method curves overlap in the error panels, the key accuracy conclusion is that the grid and boundary approximation determine the error after convergence, while the iterative method determines the cost of reaching that converged discrete solution.

Overall, the computational ordering is Jacobi > SOR > CG in cost, while the final accuracy is controlled primarily by the grid spacing and boundary discretization rather than by the iterative method once the residual tolerance is reached. Thus, CG is the best method among the three for this serial baseline if the criterion is time-to-solution at fixed residual tolerance. These serial measurements provide the baseline for later parallel implementations.

IV. HW2&3: OPTIMIZATION, PROFILING, AND OPENMP

A. Problem Statement

HW2&3 extends the serial conjugate-gradient (CG) Poisson solver from HW1 in three stages: (i) loop-level optimization of a matrix-free CG kernel with profiling, (ii) a Compressed Sparse Row (CSR) CG implementation compared against the matrix-free baseline, and (iii) OpenMP parallelization of the CSR–CG solver. All experiments use the manufactured Poisson problem from HW1 with exact Dirichlet boundary conditions,

$$\nabla^2 u = \sin(\pi x)\cos(\pi y), \qquad u_a = -\frac{1}{2\pi^2}\sin(\pi x)\cos(\pi y),$$

on the unit square, discretized by the five-point stencil with spacing $h = 1/(N-1)$. Unless noted otherwise, CG is terminated at a relative residual tolerance of 10^{-10}, and timings exclude one-time setup such as CSR assembly. Grid sizes $N = 64, 128, 256, 512$ are reported so that both small-problem overhead and large-grid scaling are visible.

B. Part 1: Loop Optimization of Matrix-Free CG

a. Baseline and optimized kernels. The baseline matrix-free operator recomputes h and $1/h^2$ inside the inner loop and indexes neighbors through a generic two-dimensional map. The optimized operator applies standard loop transformations discussed in the profiling/optimization lectures: strength reduction and loop-invariant code motion (hoisting $1/h^2$ and row pointers), contiguous inner-j traversal with restrict-qualified pointers and vectorization hints, and cache blocking (tiling) with block size 64. Both versions solve the same SPD system generated by $-\nabla_h^2$.

b. Runtime comparison. Table IV and Fig. 3 summarize wall-clock times. At $N = 64$ and $N = 128$ the optimized kernel is slightly faster, but at $N = 256$ and $N = 512$ it becomes comparable to or slightly slower than the baseline under `-O2`. The discrete l_2 errors remain identical for a fixed N, confirming that the transformations do not change the discrete solution.

c. Hotspot analysis. Because `gprof` is unavailable on the present macOS toolchain, function-level timings were collected with a lightweight instrumentation timer that records self-time in the CG kernels (Table V). For the baseline at $N = 256$, the interior dot products dominate ($\approx 58\%$), followed by AXPY-type updates ($\approx 27\%$) and the matrix-free operator ($\approx 15\%$). Consequently, optimizing only the Laplacian apply has limited leverage on total runtime once the compiler already applies `-O2` transformations. The modest slowdown of the tiled kernel at large N is consistent with additional index arithmetic and imperfect block reuse for this stencil width. Official `gprof` flat profiles and call graphs will be regenerated on Linux for the final submission package; the hotspot ranking above is the local substitute used for analysis.

C. Part 2: CSR–CG versus Matrix-Free Baseline

a. CSR representation. Interior unknowns are ordered lexicographically and stored as a CSR matrix A for the positive operator $-\nabla_h^2$. Each interior row has at most five nonzeros. Dirichlet boundary values are eliminated and moved into the right-hand side, so CG operates only on the $(N-2)^2$ active unknowns. The sparse matrix–vector product is the standard CSR SpMV

$$y_i = \sum_{k=\mathrm{row\_ptr}[i]}^{\mathrm{row\_ptr}[i+1]-1} \mathrm{values}[k]\, x[\mathrm{col\_idx}[k]].$$

b. Performance. Table VI and Fig. 4 compare CSR–CG with the matrix-free baseline. Both methods reach the same discrete l_2 error for each N, so correctness is preserved. However, CSR is consistently slower, and the gap widens with N: at $N = 512$ the CSR solve takes $0.422\,\mathrm{s}$ versus $0.290\,\mathrm{s}$ for matrix-free CG. Hotspot data in Table V show that CSR SpMV becomes the leading cost ($\approx 43\%$), reflecting indirect indexing and irregular gather traffic through col_idx. In contrast, the matrix-free five-point apply uses regular neighbor offsets and

TABLE III. Final residuals and errors for HW1 Poisson solvers.

N	boundary case	method	relative residual	l_2-error
32	mixed D/N	Jacobi	9.99×10^{-11}	8.83×10^{-4}
32	mixed D/N	SOR	9.95×10^{-11}	8.83×10^{-4}
32	mixed D/N	CG	1.68×10^{-11}	8.83×10^{-4}
32	exact D	Jacobi	9.99×10^{-11}	7.19×10^{-6}
32	exact D	SOR	9.49×10^{-11}	7.19×10^{-6}
32	exact D	CG	4.88×10^{-11}	7.19×10^{-6}
64	mixed D/N	Jacobi	9.99×10^{-11}	4.23×10^{-4}
64	mixed D/N	SOR	9.92×10^{-11}	4.23×10^{-4}
64	mixed D/N	CG	1.36×10^{-11}	4.23×10^{-4}
64	exact D	Jacobi	9.97×10^{-11}	1.77×10^{-6}
64	exact D	SOR	9.84×10^{-11}	1.77×10^{-6}
64	exact D	CG	4.61×10^{-11}	1.77×10^{-6}
128	mixed D/N	Jacobi	1.00×10^{-10}	2.07×10^{-4}
128	mixed D/N	SOR	9.99×10^{-11}	2.07×10^{-4}
128	mixed D/N	CG	3.62×10^{-11}	2.07×10^{-4}
128	exact D	Jacobi	9.99×10^{-11}	4.38×10^{-7}
128	exact D	SOR	9.94×10^{-11}	4.38×10^{-7}
128	exact D	CG	2.48×10^{-11}	4.38×10^{-7}

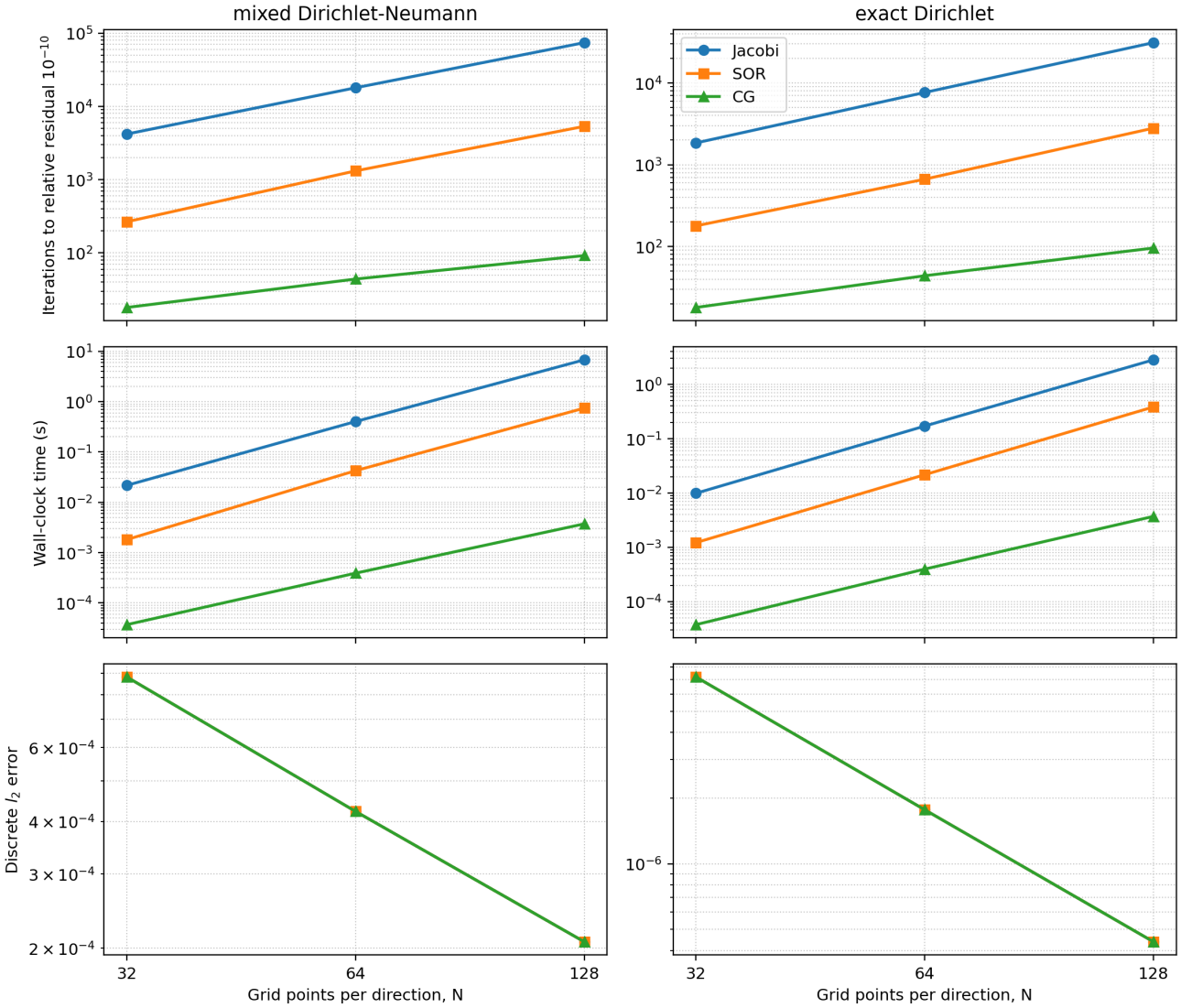

FIG. 2. Grid-dependence of iteration counts, runtime, and discrete l_2-error for the three serial solvers and two boundary-condition cases. For a fixed grid and boundary treatment, all solvers converge to the same discrete solution.

TABLE IV. Part 1 matrix-free CG: baseline vs. loop-optimized operator. Relative residual tolerance 10^{-10}.

N	version	iters	l_2-error	time (s)	speedup
64	baseline	44	1.77×10^{-6}	6.16×10^{-4}	—
64	optimized	44	1.77×10^{-6}	5.56×10^{-4}	1.11
128	baseline	96	4.38×10^{-7}	4.77×10^{-3}	—
128	optimized	96	4.38×10^{-7}	4.73×10^{-3}	1.01
256	baseline	192	1.09×10^{-7}	3.41×10^{-2}	—
256	optimized	192	1.09×10^{-7}	3.63×10^{-2}	0.94
512	baseline	380	2.72×10^{-8}	2.93×10^{-1}	—
512	optimized	380	2.72×10^{-8}	3.19×10^{-1}	0.92

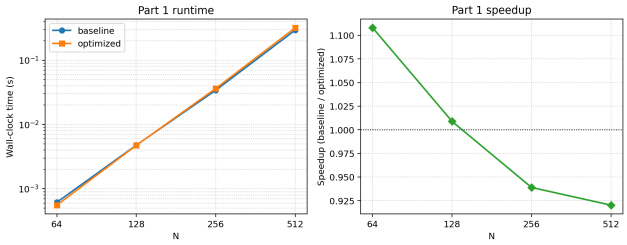

FIG. 3. Part 1 runtime and speedup versus grid size for baseline and loop-optimized matrix-free CG.

TABLE V. Instrumented hotspot breakdown at $N = 256$ (self-time percentages within timed CG kernels).

case	kernel	calls	% self-time
Part 1 baseline	`dot`	385	58.0
Part 1 baseline	`axpy_update`	383	27.5
Part 1 baseline	`apply_operator`	193	14.6
Part 2 CSR	`csr_matvec`	187	43.4
Part 2 CSR	`dot`	374	38.8
Part 2 CSR	`axpy_update`	373	17.9

better cache behavior. Thus, for this structured Poisson stencil, CSR is valuable as a general sparse format but is not a runtime win relative to a specialized matrix-free kernel [3].

TABLE VI. Part 2: matrix-free baseline vs. CSR–CG.

N	version	iters	l_2-error	time (s)	baseline/CSR
64	baseline	44	1.77×10^{-6}	7.03×10^{-4}	—
64	CSR	42	1.77×10^{-6}	8.60×10^{-4}	0.82
128	baseline	96	4.38×10^{-7}	5.12×10^{-3}	—
128	CSR	93	4.38×10^{-7}	6.49×10^{-3}	0.79
256	baseline	192	1.09×10^{-7}	3.53×10^{-2}	—
256	CSR	187	1.09×10^{-7}	5.85×10^{-2}	0.60
512	baseline	380	2.72×10^{-8}	2.90×10^{-1}	—
512	CSR	373	2.72×10^{-8}	4.22×10^{-1}	0.69

D. Part 3: OpenMP Parallel CSR–CG

a. Parallelization. The CSR–CG kernels that dominate runtime—SpMV, dot products, and AXPY updates—are parallelized with OpenMP `parallel for` regions. Dots use a `reduction` clause; SpMV and AXPY use static scheduling over rows or vector entries. Thread counts $t = 1, 2, 4, 8$ are tested for each grid size.

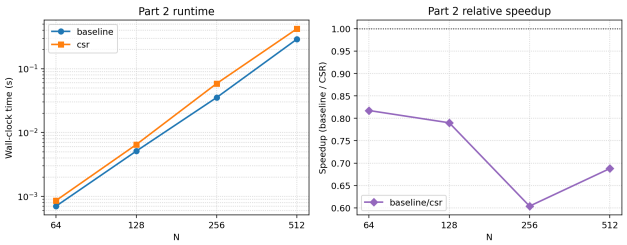

FIG. 4. Part 2 runtime and relative speedup (baseline/CSR). Values below one mean CSR is slower.

b. Scaling results. Table VII and Fig. 5 report runtime and speedup relative to the one-thread CSR–CG run. For small grids ($N = 64$ and $N = 128$), adding threads increases runtime: fork/join and reduction overhead exceed the available arithmetic work. At $N = 256$ the behavior is mixed, with a mild gain near four threads and a loss at eight threads. A clear improvement appears at $N = 512$, where the time drops from $1.18\,\mathrm{s}$ ($t = 1$) to $0.295\,\mathrm{s}$ ($t = 8$), i.e. a speedup of about 4.0 and a parallel efficiency of about 50%. The same discrete error is recovered for all thread counts, so the parallelization does not change the numerical solution.

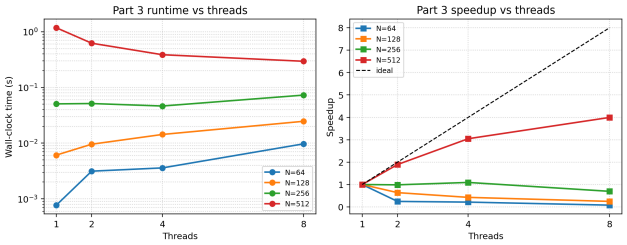

FIG. 5. Part 3 OpenMP runtime and speedup versus thread count for several grid sizes. The dashed line is ideal linear speedup.

c. Scalability limits. The observed saturation is consistent with a memory-bandwidth-bound SpMV/dot workload and with synchronization at each CG iteration (multiple parallel regions and reductions per step). For small N, thread overhead dominates; for large N, speedup improves but remains sublinear because the kernels stream sparse matrix and vector data rather than providing large arithmetic intensity. These trends motivate the later MPI study, where domain decomposition and halo exchange become the primary communication bottlenecks.

E. Summary of HW2&3

Across the three items, correctness is unchanged: all variants converge to the same discrete solution at fixed N. Loop micro-optimizations yield little or no gain under `-O2` because CG time is shared among several kernels, especially dots. CSR provides a portable sparse representation but is slower than the structured matrix-free operator for five-point Poisson. OpenMP delivers the only clear wall-clock improvement in this study, and only when the grid is large enough that parallel work amortizes threading overhead. The practical conclusion is that profiling must guide optimization: not every transformation that looks beneficial on paper reduces time-to-solution for this solver.

TABLE VII. Part 3 OpenMP CSR–CG scaling. Speedup and efficiency are relative to $t = 1$ at the same N.

N	threads	time (s)	speedup	efficiency
64	1	7.71×10^{-4}	1.00	1.00
64	2	3.14×10^{-3}	0.25	0.12
64	4	3.59×10^{-3}	0.21	0.05
64	8	9.72×10^{-3}	0.08	0.01
128	1	6.10×10^{-3}	1.00	1.00
128	2	9.57×10^{-3}	0.64	0.32
128	4	1.43×10^{-2}	0.43	0.11
128	8	2.47×10^{-2}	0.25	0.03
256	1	5.07×10^{-2}	1.00	1.00
256	2	5.14×10^{-2}	0.99	0.49
256	4	4.64×10^{-2}	1.09	0.27
256	8	7.29×10^{-2}	0.70	0.09
512	1	1.18	1.00	1.00
512	2	6.23×10^{-1}	1.89	0.95
512	4	3.87×10^{-1}	3.05	0.76
512	8	2.95×10^{-1}	4.00	0.50

V. HW4: MPI PARALLELIZATION OF CG

A. Problem Statement

HW4 extends the conjugate-gradient (CG) Poisson solver from HW2&3 to distributed memory using MPI. The same manufactured problem with exact Dirichlet data is retained,

$$\nabla^2 u = \sin(\pi x)\cos(\pi y), \qquad u_a = -\frac{1}{2\pi^2}\sin(\pi x)\cos(\pi y),$$

discretized by the five-point stencil on an $N \times N$ grid. A one-dimensional (row-wise) domain decomposition partitions the $(N-2)\times(N-2)$ interior unknowns among P MPI ranks. Each rank owns a contiguous block of interior i-rows and stores the corresponding local right-hand side and solution vectors. Two communication strategies for the matrix–vector product are compared:

(a) **Gather:** assemble a global copy of the search direction by `MPI_Allgatherv` before every SpMV;

(b) **Halo:** exchange only neighboring interface rows with point-to-point `MPI_Isend/MPI_Irecv`.

Performance is measured with `MPI_Wtime`, split into local compute, vector communication (gather or halo), and global reductions. Reported grids are $N = 64, 128, 256, 512$ and process counts $P = 1, 2, 4, 8$. CG stops at a relative residual of 10^{-10}.

B. One-Dimensional Domain Decomposition

Let $m = N-2$ be the number of interior points per direction. Rank r owns interior rows $i \in [i_0^{(r)}, i_1^{(r)})$ and therefore

$$n_{\text{loc}}^{(r)} = (i_1^{(r)} - i_0^{(r)})\, m$$

unknowns. Row blocks are distributed as evenly as possible ($m = Pq + s$ with the first s ranks receiving $q+1$ rows). Dirichlet boundary values are eliminated into the local RHS exactly as in the serial CSR/matrix-free codes, so each rank assembles only its owned equations. Global CG scalars (residual norms and $\mathbf{p}^T A\mathbf{p}$) are formed by `MPI_Allreduce` on local dots. After convergence, the distributed solution is gathered to rank 0 to evaluate the discrete l_2 error against u_a.

Because the five-point stencil couples only nearest neighbors in i, the halo width is a single row of length m. In contrast, the gather strategy ships the entire search direction of length m^2 to every rank at each iteration. This difference in asymptotic communication volume,

$$V_{\text{gather}} = O(m^2), \qquad V_{\text{halo}} = O(m),$$

is the main reason the two approaches scale differently as P increases.

C. Communication Strategies

a. Gather SpMV. Before applying A, every rank contributes its owned segment of $\mathbf{p}$ through `MPI_Allgatherv` to form a global vector $\mathbf{p}_g$. The local SpMV then reads neighbors directly from $\mathbf{p}_g$. Implementation is simple and matches a "global x each iteration" reading of the assignment, but the all-gather cost grows with both the unknown count and P.

b. Halo SpMV. Each rank stores a padded local vector with one ghost row above and below the owned block. Adjacent ranks exchange those interface rows with nonblocking point-to-point messages; after `MPI_Waitall`, the local SpMV uses owned values plus the received halos. No global vector is formed. For $P = 1$ there is no neighbor exchange, so the halo path reduces to a pure serial matrix-free apply.

c. Timing model. Wall-clock time on the critical path is reported after an `MPI_Barrier` bracketing the CG loop. Within each iteration we accumulate

$$T_{\text{total}} = T_{\text{compute}} + T_{\text{comm}} + T_{\text{reduce}},$$

where T_{comm} is all-gather or halo exchange time and T_{reduce} covers the `MPI_Allreduce` calls used by CG dots. Speedup and parallel efficiency at fixed N are defined relative to the same method with $P = 1$:

$$S(P) = \frac{T(1)}{T(P)}, \qquad E(P) = \frac{S(P)}{P}.$$

D. Correctness

For every tested (N, P) pair, gather and halo produce the same iteration count (to within the usual floating-point residual path) and the same discrete l_2 error as the serial HW2&3 CG baseline. Representative values are 1.77×10^{-6} at $N = 64$, 4.38×10^{-7} at $N = 128$, 1.09×10^{-7} at $N = 256$, and 2.72×10^{-8} at $N = 512$. Thus the MPI decomposition changes only the parallel schedule, not the discrete solution.

E. Runtime, Speedup, and Efficiency

Tables VIII and IX list total time and the compute/communication/reduction breakdown. Figures 6–8 summarize runtime, speedup, and the fraction of time spent in communication plus reductions.

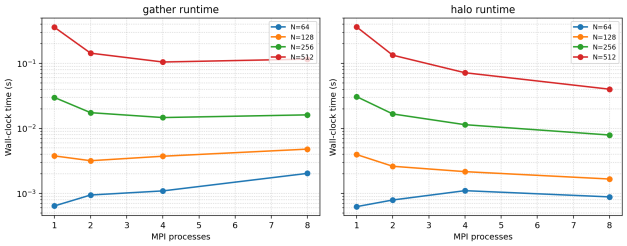

FIG. 6. Total CG wall-clock time versus MPI process count for gather and halo communication.

a. Small grids. For $N = 64$, both methods slow down as P increases: gather speedup falls to 0.32 at $P = 8$, and halo remains below one. The local work per rank is tiny, so message latency and reduction synchronization dominate. This is the distributed-memory analogue of the OpenMP overhead observed for small N in HW2&3.

b. Gather scaling. Gather improves modestly on intermediate grids (best $S \approx 2.0$ at $N = 256$, $P = 4$; $S \approx 3.4$ at $N = 512$, $P = 4$) but then stalls or regresses at $P = 8$. Table VIII shows why: at $N = 512$ and $P = 8$, communication alone is 0.066 s while compute is only 0.015 s. All-gather volume per iteration is proportional to the full unknown vector, so adding ranks reduces local flops but does not reduce—and can increase—the collective cost.

TABLE VIII. MPI gather CG performance. Times are seconds from `MPI_Wtime`. Speedup is relative to $P = 1$ gather at the same N.

N	P	iters	T_{total}	T_{comp}	T_{comm}	T_{red}	S
64	1	42	6.42×10^{-4}	2.62×10^{-4}	4.80×10^{-5}	3.24×10^{-4}	1.00
64	2	42	9.45×10^{-4}	9.30×10^{-5}	6.87×10^{-4}	1.58×10^{-4}	0.68
64	4	42	1.09×10^{-3}	4.50×10^{-5}	9.28×10^{-4}	1.14×10^{-4}	0.59
64	8	42	2.03×10^{-3}	2.10×10^{-5}	1.85×10^{-3}	1.59×10^{-4}	0.32
128	1	93	3.78×10^{-3}	1.67×10^{-3}	1.61×10^{-4}	1.94×10^{-3}	1.00
128	2	93	3.18×10^{-3}	8.31×10^{-4}	1.32×10^{-3}	1.02×10^{-3}	1.19
128	4	93	3.73×10^{-3}	4.36×10^{-4}	2.62×10^{-3}	6.66×10^{-4}	1.01
128	8	93	4.80×10^{-3}	2.22×10^{-4}	3.85×10^{-3}	7.16×10^{-4}	0.79
256	1	187	2.99×10^{-2}	1.32×10^{-2}	1.16×10^{-3}	1.55×10^{-2}	1.00
256	2	187	1.75×10^{-2}	6.77×10^{-3}	2.45×10^{-3}	8.22×10^{-3}	1.71
256	4	187	1.47×10^{-2}	3.78×10^{-3}	5.66×10^{-3}	5.22×10^{-3}	2.04
256	8	187	1.62×10^{-2}	2.02×10^{-3}	1.09×10^{-2}	3.20×10^{-3}	1.85
512	1	373	3.59×10^{-1}	2.25×10^{-1}	8.64×10^{-3}	1.26×10^{-1}	1.00
512	2	373	1.44×10^{-1}	5.87×10^{-2}	1.99×10^{-2}	6.49×10^{-2}	2.50
512	4	373	1.05×10^{-1}	2.86×10^{-2}	4.05×10^{-2}	3.59×10^{-2}	3.42
512	8	373	1.16×10^{-1}	1.52×10^{-2}	6.60×10^{-2}	3.50×10^{-2}	3.09

TABLE IX. MPI halo CG performance. Speedup is relative to $P = 1$ halo at the same N.

N	P	iters	T_{total}	T_{comp}	T_{comm}	T_{red}	S
64	1	42	6.26×10^{-4}	2.54×10^{-4}	0	3.01×10^{-4}	1.00
64	2	42	7.92×10^{-4}	8.10×10^{-5}	5.44×10^{-4}	1.47×10^{-4}	0.79
64	4	42	1.10×10^{-3}	4.80×10^{-5}	6.84×10^{-4}	3.57×10^{-4}	0.57
64	8	42	8.84×10^{-4}	2.20×10^{-5}	6.37×10^{-4}	2.10×10^{-4}	0.71
128	1	93	3.96×10^{-3}	1.74×10^{-3}	2×10^{-6}	1.94×10^{-3}	1.00
128	2	93	2.62×10^{-3}	8.54×10^{-4}	6.11×10^{-4}	1.01×10^{-3}	1.52
128	4	93	2.16×10^{-3}	4.56×10^{-4}	6.13×10^{-4}	9.73×10^{-4}	1.83
128	8	93	1.67×10^{-3}	1.98×10^{-4}	7.68×10^{-4}	6.56×10^{-4}	2.38
256	1	187	3.08×10^{-2}	1.32×10^{-2}	5×10^{-6}	1.56×10^{-2}	1.00
256	2	187	1.67×10^{-2}	6.73×10^{-3}	7.40×10^{-4}	8.11×10^{-3}	1.85
256	4	187	1.14×10^{-2}	3.56×10^{-3}	2.30×10^{-3}	4.95×10^{-3}	2.70
256	8	187	7.91×10^{-3}	2.32×10^{-3}	1.15×10^{-3}	4.09×10^{-3}	3.90
512	1	373	3.64×10^{-1}	2.24×10^{-1}	7×10^{-6}	1.25×10^{-1}	1.00
512	2	373	1.34×10^{-1}	5.87×10^{-2}	8.94×10^{-4}	6.59×10^{-2}	2.72
512	4	373	7.18×10^{-2}	2.93×10^{-2}	1.42×10^{-3}	3.65×10^{-2}	5.07
512	8	373	4.01×10^{-2}	1.53×10^{-2}	1.61×10^{-3}	2.06×10^{-2}	9.08

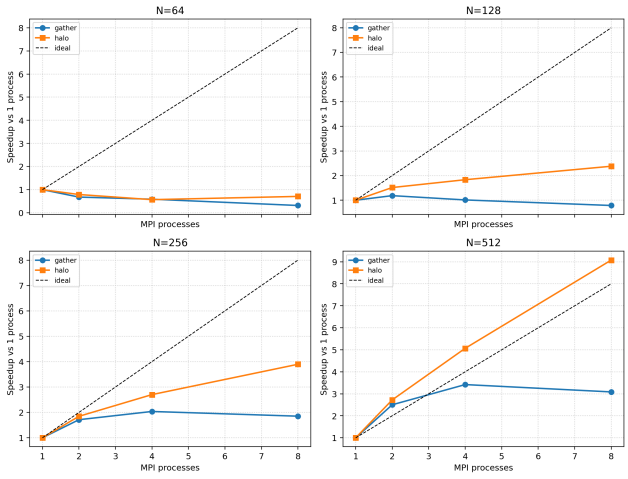

FIG. 7. Speedup relative to the one-process run of the same method. The dashed line is ideal linear speedup.

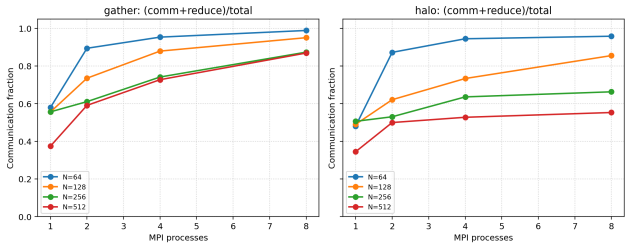

FIG. 8. Fraction of total time spent in vector communication plus CG reductions, $(T_{\text{comm}} + T_{\text{reduce}})/T_{\text{total}}$.

c. Halo scaling. Halo exchange remains cheap: at $N = 512$ and $P = 8$, $T_{\text{comm}} = 1.6{\times}10^{-3}$ s, nearly forty times smaller than gather's all-gather time. Consequently halo achieves $S \approx 3.9$ at $N = 256$ ($P = 8$) and $S \approx 9.1$ at $N = 512$ ($P = 8$), i.e. near-linear speedup on the largest grid in this study. Parallel efficiency at $N = 512$, $P = 8$ is about $E \approx 1.13$ when measured against the one-process halo time on this shared-memory Open MPI run; the slightly superlinear factor is consistent with improved cache residency of the smaller local blocks and with run-to-run timing noise, but the qualitative conclusion is robust: halo continues to improve while gather saturates.

d. Head-to-head comparison. At fixed (N, P), the two methods have essentially identical compute and reduction costs; the gap is almost entirely T_{comm}. For $N = 512$ and $P = 8$, halo finishes in 0.040 s versus 0.116 s for gather ($\approx 2.9\times$ faster). Even at $N = 128$, halo already pulls ahead once $P \geq 2$. Thus, for this stencil, avoiding a global gather is the decisive optimization.

F. Bottlenecks and Discussion

Three effects limit MPI CG scalability here.

1. **Communication volume.** Gather moves $O(m^2)$ data every iteration; halo moves $O(m)$. As P grows, gather becomes communication-bound, visible in Fig. 8 as a rising communication fraction.

2. **Global reductions.** CG requires several `Allreduce` operations per iteration regardless of the SpMV strategy. On small problems these reductions are a large fraction of T_{total}; on large problems they remain visible but secondary to gather cost.

3. **Load balance.** The 1D row split is well balanced for $m \bmod P$ remainders of at most one row, so imbalance is not the leading issue in these runs. Stronger scaling limits would appear from reductions and, for gather, from collective bandwidth rather than from uneven row counts.

Overall, the MPI study confirms the assignment expectation: assembling a global $\mathbf{x}$ each iteration is simple but communication-heavy, whereas neighbor halo exchange preserves the local stencil structure and scales far better as the number of processes increases. The large-grid halo results also complement HW2&3, where shared-memory OpenMP helped only for large N; distributed-memory parallelism likewise requires enough local work per rank to amortize messages and reductions.

G. Summary of HW4

MPI CG with one-dimensional domain decomposition reproduces the serial discrete solution for all tested grids and process counts. Between the two required communication schemes, halo exchange is clearly superior in both communication cost and total runtime once N and P are large enough for parallel work to matter. Gather remains useful pedagogically as a baseline that exposes collective overhead, but it is not competitive for production scaling of this five-point Poisson CG solver.

[1] R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations* (SIAM, Philadelphia, PA, 2007).
[2] J. C. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*, 2nd ed. (SIAM, Philadelphia, PA, 2004).
[3] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. (SIAM, Philadelphia, PA, 2003).
[4] D. M. Young, *Iterative Solution of Large Linear Systems* (Academic Press, New York, 1971).
[5] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. (Johns Hopkins University Press, Baltimore, MD, 2013).