Parallel Conjugate Gradient Method with Stencil-Based Matrix-Vector Multiplication

Parallel and Distributed Programming

Yangmei Lin

VT 2024

1 Introduction

The Conjugate Gradient method is one of the most widely used iterative methods that form the basis of many scientific and industrial applications. It aims to solve systems of the form $A\mathbf{x} = \mathbf{b}$, where A is a sparse symmetric positive definite matrix [1], $\mathbf{b}$ is a known vector, and $\mathbf{x}$ is the approximation solution we look for. Many such large symmetric linear systems arise from the discretization of Partial Differential Equations [2]. The background of the equation we solved in this project is derived from the discretization of the Poisson equation, where the matrix A is the discrete operation [3]. Due to the specific structure of A, we do not need to explicitly define A when computing $A\mathbf{b}$; instead, we can use stencil computation.

The operations involved in the iteration process of the CG method are well-suited for parallelization. The parallel implementation is based on MPI, which operates in a distributed memory environment and enables processes to communicate with each other through message passing [4].

2 Implementation

2.1 Sequential Algorithm

We target solving the equation $A\mathbf{u} = \mathbf{b}$. Elements of $\mathbf{b}$ are defined by the coordinates of a two-dimensional mesh. The size of the mesh is 1×1 and is evenly discretized into $n+1$ intervals. The components of $\mathbf{b}$ are therefore computed as $b_{ij} = 2h^2(x_i(1-x_i)+y_i(1-y_i))$, where $h = \frac{1}{n+1}$.

The matrix A is defined by a five-point stencil as shown below:

$(j+1)$		-1	
(j)	-1	4	-1
$(j-1)$		-1	
	$(i-1)$	(i)	$(i+1)$

For inner points on the mesh (which are associated with the elements of vectors $\mathbf{u}$ and $\mathbf{b}$), we have the following equation:

$$-u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} + 4 \times u_{i,j} = b_{i,j}$$

With Dirichlet boundary conditions, the values of the boundary points on the mesh are zero. The factors in the formula represent the non-zero elements in a row of A. A point on the mesh has a maximum of 4 defined neighbors and a minimum of 2, so the non-zero

elements in a row of A are maximum 5 and minimum 3. Since we have a total of $N = n \times n$ inner points, the size of A is $N \times N$.

With A and $\mathbf{b}$ ready, we can perform CG to solve the problem. The algorithm terminates when the number of iterations exceeds the maximum allowed iterations, or when the error, defined as the residual $= \mathbf{b} - A\mathbf{x}$ smaller than the tolerance error [5].

The pseudo-code is provided below:

Algorithm The Conjugate Gradient (CG) Algorithm

Define A and b based on the input parameter n
Define maximum iterations $imax$ and tolerance error $rmax$
Initialize $u = 0$, $g = -b$, $d = b$
$q_0 = g^T g$
for $it = 1, 2, \ldots$ until $it == imax$ or $|b - Au| < rmax$ **do**
    $q = Ad$
    $\tau = q_0/(d^T q)$ ▷ Step length
    $u = u + \tau d$ ▷ Approximate solution
    $g = g + \tau q$ ▷ New residual
    $q_1 = g^T g$
    $\beta = q_1/q_0$ ▷ Improvement
    $d = -g + \beta d$ ▷ New search direction
    $q_0 = q_1$
end for
return u, the approximation solution

2.2 Parallel Algorithm Using MPI

The parallel algorithm employs MPI to distribute the workload among processes.

2.2.1 Initialization

Mesh points are distributed among p processes. To simplify the setting, we assume a 2D grid of processes, for example, $p = 9 = 3 \times 3$, as shown in Figure 1, and we partition the mesh points using block partitioning. For better load balancing, if n is not divisible by $\sqrt{p}$, the extra rows or columns are evenly distributed among some of the processors, so that the load difference between processes is $2\sqrt{p} - 1$ at most.

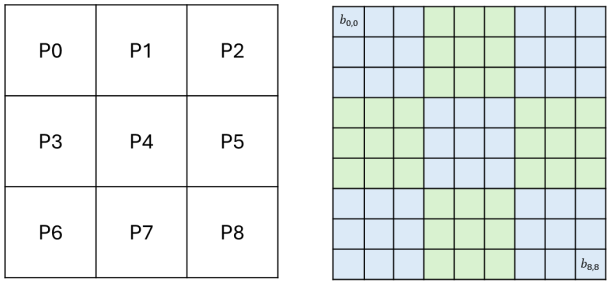

Figure 1: Logical architecture of mesh points, $n = 9$, two colors used to differentiate blocks

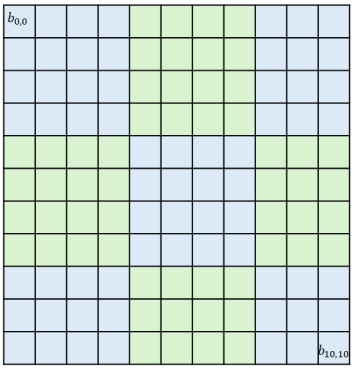

Figure 2: n is indivisible by $\sqrt{p}$, $n = 11$, two colors used to differentiate blocks

The elements of **b** are associated with the coordinates of points on the original mesh. Thus we need to record the corresponding start and end indices of the mesh points block for each processor. Then, we initialize the subset entries of b on each process.

Each iteration consists of three basic operations: matrix-vector multiplication, vector updates, and scalar products. All calculations are element-wise. Therefore, the vectors used in iterations don't need to be stored in arrays but remain the same shape as **b**, storing the corresponding portion in each process locally.

2.2.2 Stencil-Based Matrix-Vector Multiplication

In the iterations, the first step calculates $\mathbf{q} = A\mathbf{d}$. When we apply A to **d**, it actually means $q_{ij} = -d_{i-1,j} - d_{i+1,j} - d_{i,j-1} - d_{i,j+1} + 4 * d_{i,j}$. Figure 3 visualizes this process.

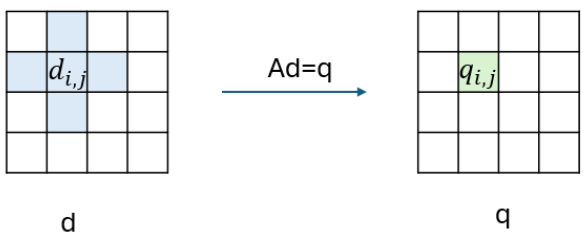

Figure 3: Five-point stencil applied on **d**

After we distribute mesh points among processes, some required data are stored on other processes as boundary points, so there is communication to exchange this data. We do this by adding one layer of halo points on $\mathbf{d}$, wherein the boundary is set to zeros.

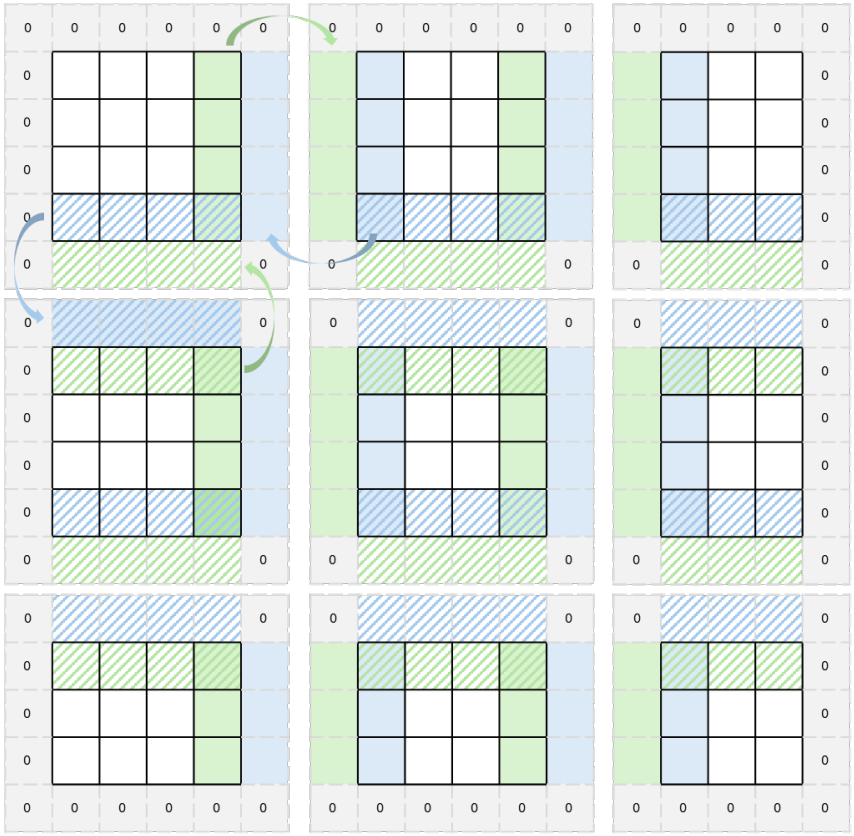

Figure 4: Blocks of mesh points with onr layer of halo points

The data exchange occurs at the beginning of each iteration. The function used here is `MPI_Sendrecv` instead of a pair of `MPI_Send` and `MPI_Recv` functions. `MPI_Sendrecv` allows the process to simultaneously send and receive data in a single function call, which can be more efficient and make the code concise. Additionally, it reduces the likelihood of deadlock situations where processes are waiting indefinitely for each other to send or receive data [5].

The exchange rule is, that if the current process has neighbors in our logical processes structure, it will send or receive boundary data to or from that neighbor. The received data will be stored in the added halo points. Since we add a layer of halo points when we do the five-point stencil, there will be a shift by 1 for elements of $\mathbf{d}$. Thus, the formulation we have now is :

$$q_{ij} = -d_{i,j+1} - d_{i+2,j+1} - d_{i+1,j} - d_{i+1,j+2} + 4 \times d_{i+1,j+1}$$

The shift also occurs when we perform vector updates and scalar products with $\mathbf{d}$.

2.2.3 Scalar Products

The scalar product takes two vectors and returns a scalar quantity. The rule is to iterate over the elements and multiply corresponding elements together, then sum up the results.

In the parallel implementation, each process is responsible for computing the partial scalar product locally, and then summing up the results using `MPI_Allreduce` with `MPI_SUM`. This function aggregates all the results, sums them, and stores the result in all processes.

2.2.4 Vector Updates

This operation will add two vectors with scalar factors. A typical equation is $\mathbf{c} = \alpha\mathbf{a} + \beta\mathbf{b}$. It is performed entirely locally with no communication between processes.

2.2.5 Output

The iteration will terminate after 200 iterations for performance examination. The output is the norm of vector $\mathbf{g}$ which represents the residual. Before printing the result, we need to aggregate the local scalar product of $\mathbf{g}T\mathbf{g}$ onto Process 0.

3 Experiments and results

3.1 Verification of Implementation

The correctness of the implementation is verified by comparing the results with those obtained using the GC solver in `scipy` in Python (as shown in Table 1 with the same number of iterations.

Table 1: Verification of the implementation with the number of iteration is 200

n	R_m(**MPI_CG**)	R_s(`scipy.linalg.cg`)	$R_m - R_s$
256	0.0000381655	0.0000381655	0
512	0.0044322618	0.0044322618	0
768	0.0063415660	0.0063415660	-1×10^{-10}
1024	0.0072754062	0.0072754063	0
1280	0.0076202597	0.0076202597	0
1536	0.0078980946	0.0078980949	-3×10^{-10}
1792	0.0081421283	0.0081421282	0
2048	0.0082487287	0.0082487286	-1×10^{-10}

3.2 Strong scalability

The strong scalability is measured by increasing the number of processes while keeping the problem size constant. In this case, the problem size is $n = 2048$. The speed-up is defined as $S = \frac{T_s}{T_p}$ and the efficiency is defined as $E = \frac{S}{p}$. The execution time is the maximum of the times measured across all processes.

Table 2: Execution time, speedup and efficiency for different number of processes

Number of processes	Execution time (seconds)	Speedup	Efficiency
1	7.589	1.000	1.000
4	2.434	3.118	0.773
9	2.026	5.336	0.416
16	1.551	4.893	0.286
25	1.085	6.996	0.780
36	0.835	9.091	0.252
49	0.476	15.947	0.325
64	0.263	28.863	0.451

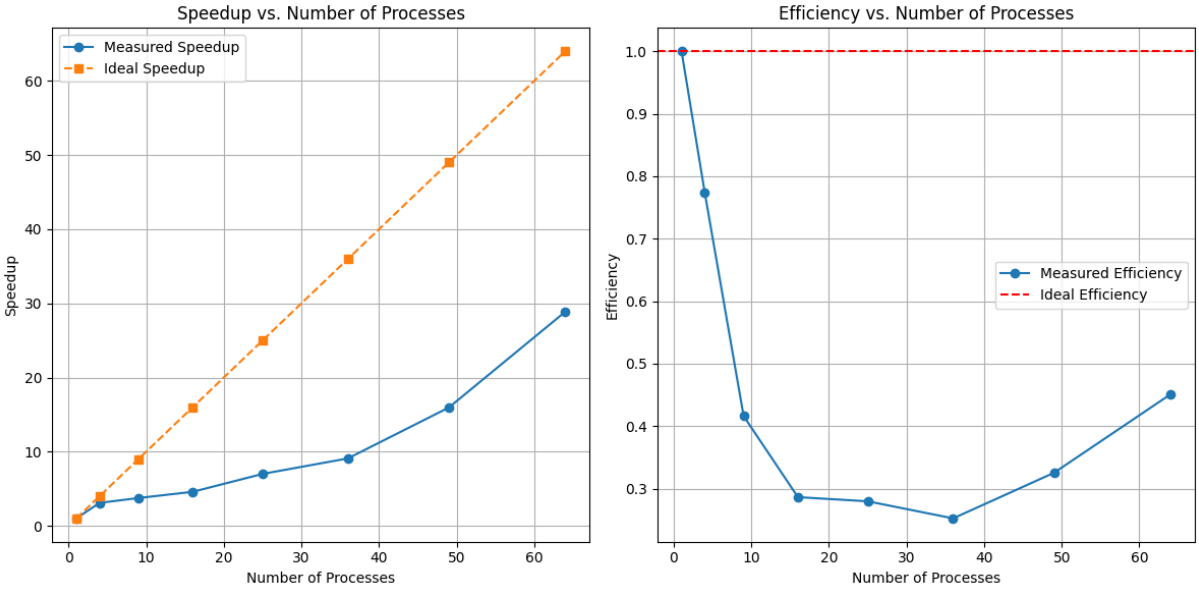

Figure 5: Strong scalability

As shown in Table 2 and Figure 6, the execution time decreases as the number of processes increases, and shows higher speedup and efficiency. The difference between the real and the ideal is the result of the boundary data exchange between processes, and the global sum reduction when calculating scalar products.

We can also observe that efficiency rises when p is larger. The reason may be that for larger p, the better cache performance offsets the inefficiency that comes from the communication overhead. We use `perf stat -e cache-misses <mpi_program>` to measure the average

cache misses for different numbers of processes. We observe from Table that the decrease in cache misses is not linear with the number of processes. When p is larger, the cache hit is significantly improving.

Table 3: Cache performance with different number of processes

Number of processes	**Average number of cache misses (M_i)**	$\frac{M_1}{pM_i}$
1	230,062,410	1.000
4	57,146,464	1.006
9	29,280,675	0.873
16	19,206,511	0.749
25	8,549,366	1.076
36	5,195,397	1.230
49	2,597,947	1.807
64	896,686	1.807

3.3 Weak scalability

Weak scalability is measured by increasing the number of processes while keeping the problem size per process constant but increasing the overall problem size. In this case, $n = 256$ on per process.

Table 4: Execution time and efficiency for different number of processes

Number of processes	**Execution time (seconds)**	**Efficiency**
1	0.110	1.000
4	0.114	0.965
9	0.128	0.859
16	0.143	0.769
25	0.243	0.453
36	0.275	0.400
49	0.302	0.364
64	0.237	0.464

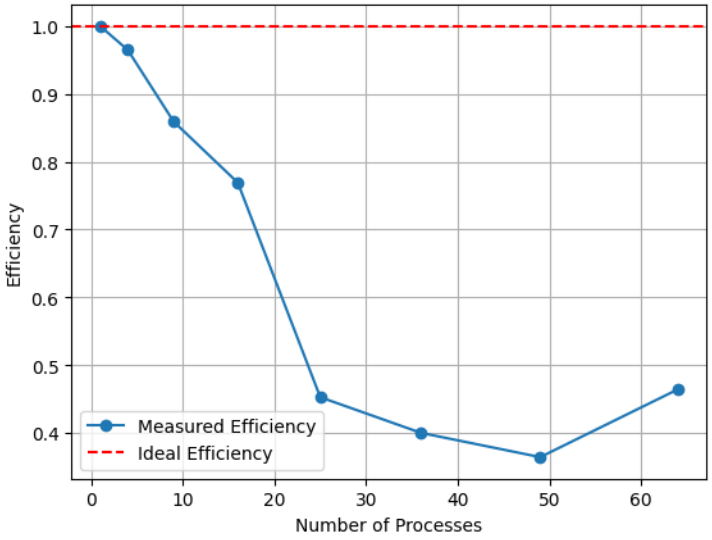

Figure 6: Weak scalability

The execution time increases slightly as the number of processes increases, indicating that the algorithm scales well with the problem size. The efficiency decreases because of the increased communication overhead as more processes are added, for example, more time for waiting for simultaneous in each iteration, and global sum reduction.

4 Discussion

The outputs for different values of input parameter n are shown in Table 5.

Table 5: Residuals for increasing discretized points of mesh

n	**residual** ($\|g\|_2$)
256	3.82×10^{-5}
512	4.43×10^{-3}
768	6.34×10^{-3}
1024	7.28×10^{-3}
1280	7.62×10^{-3}
1536	7.90×10^{-3}
1972	8.14×10^{-3}
2048	8.25×10^{-3}

The residuals increase as n increases. The reason is the rate of convergence of CG depends on the condition number of A. For Poissons's equation, $cond(A)$ is $\mathcal{O}(n^2)$ [6]. Therefore, for larger n, it requires more iterations to achieve the same accuracy. For example, with $n = 1024$, the iteration count would be 722, with a residual of 9.75×10^{-6}.

To improve the performance of the code, we can use MPI persistent communications for data exchange instead of `MPI_Sendrecv`. MPI persistent communications are nonblocking

send and receive operations. They can reduce communication overhead like redundant message setup in programs that repeatedly call the same point-to-point message passing routines with the same arguments. For example, in this project, they exchange border elements with neighbors in each iteration.

The data exchange for columns is more complex than for rows since these elements are not stored contiguously. This issue in this project is resolved by storing the column elements in a new array and, after receiving them, storing them back into **b**. An alternative method is to construct an `MPI_Datatype` to group the column data.

We can also consider using GPU acceleration, which is specialized for compute-intensive, massively parallel computation [7].

References

[1] Siegfried Cools et al. "Improving strong scaling of the conjugate gradient method for solving large linear systems using global reduction pipelining". In: *arXiv preprint arXiv:1905.06850* (2019).

[2] Mathias Malandain, Nicolas Maheu, and Vincent Moureau. "Optimization of the deflated conjugate gradient algorithm for the solving of elliptic equations on massively parallel machines". In: *Journal of Computational Physics* 238 (2013), pp. 32–47.

[3] Clive Temperton. "On the FACR (l) algorithm for the discrete Poisson equation". In: *Journal of Computational Physics* 34.3 (1980), pp. 314–329.

[4] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011.

[5] Jonathan Richard Shewchuk et al. "An introduction to the conjugate gradient method without the agonizing pain". In: (1994).

[6] Jim Demmel. "Solving the discrete Poisson equation using Jacobi, SOR, Conjugate Gradients, and the FFT". In: *Retrieved from Lecture Notes Online Website: http://www. eecs. berkeley. edu/˜ demmel/cs267/lecture24/lecture24. html* (1996).

[7] Rudi Helfenstein and Jonas Koko. "Parallel preconditioned conjugate gradient algorithm on GPU". In: *Journal of Computational and Applied Mathematics* 236.15 (2012), pp. 3584–3590.

Peer Review (By Linjing Shen)

The report is nicely organized and easy to follow. It's got a good flow with sections that use visuals like images, tables, and pseudocode, making it a breeze to understand. This setup gives a clear picture of how the parallel conjugate gradient algorithm works, from distributing and swapping grid points among processes to assessing scalability through experiments. Plus, there's a solid list of references backing everything up, adding weight to the findings.

The code is also well-structured, with different parts neatly separated out. There are sections for MPI setup, memory stuff, and the CG iteration, all clearly defined. It's robust too, handling communication and load balancing smoothly. Using $MPI\_Sendrecv$ for exchanging boundary values avoids any deadlocks and keeps data in sync. The code dynamically determines neighbor process indices and employs $MPI\_Allreduce$ for global reduction operations, enhancing efficiency. Additionally, the use of functions to abstract matrix operations contributes to code cleanliness and maintainability. In the test after compiler acceleration, it is reasonable that the measured speedup can reach 1/2 of the ideal speedup and the measured efficiency can remain relatively stable within a certain range.

The experimental results are correct and presented comprehensively, covering process counts from 1 to 64, giving a good range for scalability testing.

Overall, it's a solid project. Everything including the `README.md`, the `Makefile`, the code, and the report looks polished. But if there's one thing to consider, it's adding more comments in the code, especially for trickier parts like matrix operations and MPI communication. That would make things even clearer and easier to work with.