MARCH 2022

A dissertation submitted in partial satisfaction of the requirements for
the degree of Doctor of Philosophy in Engineering

Parallelization of a Poisson Solver by Block Red-Black Ordering and Its Application to Particle-in-Cell Plasma Simulation

The University of Electro-Communications
Graduate School of Informatics and Engineering
Department of Computer and Network Engineering

Akemi Shioya

Parallelization of a Poisson Solver by Block Red-Black Ordering and Its Application to Particle-in-Cell Plasma Simulation

Committee Members:

Prof. Yusaku Yamamoto
Associate Prof. Tomoya Tatsuno
Prof. Nobito Yamamoto
Prof. Hidenori Ogata
Associate Prof. Tadashi Yamazaki
Tetsu Narumi

概要

ブロック赤黒順序付けによる Poisson ソルバーの並列化と Particle-in-Cell プラズマシミュレーションへの応用

プラズマ挙動を解析する Particle-In-Cell 法を製造装置解析に適用すると、Poisson 方程式の求解が、計算時間の多くを占める。この求解を Krylov 部分空間法で行うための修正不完全分解前処理は高い収束性を持ち、その並列化には高度な並列性と少ない同期点を持つブロック赤黒順序付けが使用できる。これらの組み合わせた時に発生するゼロピボットによる破綻の必要十分条件を示し、これを回避して強力な前処理とする方法を示した。我々はこの前処理の GPU 上への実装を試み、実装上の工夫とパラメータ最適値により、既存ライブラリよりも高速に精度の良い解を得た。さらに、ブロック赤黒順序付けの高い収束性の理由を示すために、収束性を調査した。

Abstract

Parallelization of a Poisson Solver by Block Red-Black Ordering and Its Application to Particle-in-Cell Plasma Simulation

The particle-in-cell method is a particle-based approach to the analysis of non-equilibrium plasma behavior under low-pressure conditions. In applications of the method to manufacturing equipment, a significant proportion of the required computational time is consumed by solving linear simultaneous equations with sparse coefficient matrices, which results from discretizing the Poisson equations by finite difference. Modified incomplete factorization with no fill-in is effective as a preconditioner for solution with the Krylov subspace method without requiring additional memory. Compensating for the discarded fill-in values of the diagonal elements results in faster convergence than the simple incomplete factorization preconditioner. The block red-black ordering can be used to parallelize this preconditioner with a large degree of parallelism and fewer synchronization points. The block red-black ordering leverages the fact that blocks of orthogonal grids can be painted in two colors to reduce the synchronization point to one. Moreover, it enables the degree of parallelism and convergence to be flexibly adjusted by varying the number of blocks.

Although the combination of modified incomplete factorization and block red-black ordering appears to produce a powerful and parallelizable preconditioner, in reality, it can cause a breakdown of factorization due to zero pivots. In this work, this phenomenon is analyzed to determine the necessary and sufficient conditions for the occurrence of zero-pivot in the case of orthogonal grids. We also prove theoretically and experimentally that adding perturbations to the diagonal elements or relaxing the compensation of the dropped fill-in are valid techniques to mitigate this issue. Numerical results show that the resulting preconditioner not only avoids the zero-pivot but also achieves a higher convergence rate and is applicable up to 10^3 degrees of parallelism.

Therefore, we implemented this preconditioner on a graphics processing unit (GPU) with a higher degree of parallelism. We realized coalesced memory access by optimizing the storage form of the matrix for block parallelization. To reduce the amount of data and utilize the single-precision performance of the GPU, we adopted a mixed-precision calculation in which some array data and operations were respectively stored and performed in single-precision, while nonetheless retaining an accuracy comparable to to comprehensive use of double-precision computation. Several numerical experiments were performed to determine the optimal values of adjustable parameters such as block partitioning conditions and parallel grain size. The results of performance comparisons using an NVIDIA Quadro GP100 GPU and an NVIDIA Tesla K40t GPU show that our solver is faster than existing libraries, even with conservative

implementations using OpenACC directives. The test problems for the series of numerical test results were obtained from potential calculations of a 3D magnetron sputter model using the particle-in-cell method with a 7-point stencil.

In addition, we investigated the convergence of block red-black ordering using the model problem, and identified the reason for the fast convergence of this ordering.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Purpose

Particle-In-Cell (PIC) is a numerical analysis method for non-equilibrium plasma in a rarefied gas [1, 2]. In the PIC method, the actual plasma behavior is reproduced by moving the simulation particles that represent charged particles, thereby enabling the simulation of non-equilibrium plasma, which is difficult to perform using ordinary continuum approximation. In PIC plasma simulation, the electric potential is obtained by solving the Poisson equation from the charge density distribution, where the charge of the simulated particle is assigned to each grid point. In such simulations, solving the Poisson equation is an important problem that sometimes accounts for the majority of calculation time.

The solution of the Poisson equation is reduced to the solution of linear system $A\boldsymbol{x} = \boldsymbol{b}$ with a large sparse matrix, as in various scientific and technical calculations. For solving linear simultaneous equations, Krylov subspace methods preconditioned with Incomplete LU (ILU) or Incomplete Cholesky (IC) factorization are widely used. LU factorization is the factorization of a matrix into the product of lower triangular matrix L and upper triangular matrix U. In Cholesky factorization, U is $L^\top$. In incomplete factorization, when an element that was zero in the original matrix becomes nonzero (fill-in), it is sometimes dropped and retained as zero. This suppresses the increase in the number of nonzero matrix elements but has a negative effect on convergence. The level at which fill-ins are allowed is indicated by the number in parentheses, and the level for dropping all fill-ins is 0, represented by (0). Modified ILU/IC (MILU/IC) is a variant of incomplete factorization that reduces the effect of this dropping and improves convergence. MILU(0)/IC(0) drops the elements at zero position in the original matrix at the time of factorization, and it subtracts the sum of the dropped elements in each row from the diagonal element of that row of U to compensate for the dropping. The MILU(0)/MIC(0) preconditioner is generally known to have a stronger convergence acceleration effect compared with ILU(0)/IC(0). In fact, the spectral condition number, which indicates the number of iterations required for convergence, is $O(h^{-1})$, where h is the grid size used for discretizing the model problem; meanwhile, in ILU(0)/IC(0), the condition number is $O(h^{-2})$ [3, 4, 5].

In recent years, with the emergence of high-degree parallelism of computers, speeding up computation through parallelization has become an important issue. However, it is well known that parallelization of ILU/IC preconditioner, which uses natural ordering of the grid points, is difficult. The incomplete factorization requires that, before processing the rows associated with

the node with a certain index number, the rows with smaller numbers connecting to the node should be processed. With the natural ordering of orthogonal structured grids, the processing of a row cannot begin until all rows associated with a set of nodes upwind in each axis have been processed; therefore, most of the factorization needs to be completed sequentially.

Reordering techniques are used to maximize the number of nodes for which no adjacent nodes have a smaller number than their own [6, 7, 8, 9, 10, 11, 12]. A technique used in combination with reordering is the coloring technique, in which nodes that can be computed at the same time are given the same color to parallelize computations that contain dependencies. For example, nodes that can be computed without waiting for other nodes to be computed are colored with color (1), and nodes that depend only on nodes with color (1) are colored with color (2). In this case, after the nodes with color (1) are computed in parallel, the processes are synchronized once and then the nodes with color (2) can be computed in parallel. A reordering method that uses this coloring technique is block red-black (BRB) ordering [13]. BRB ordering can control the degree of parallelism up to half the number of nodes in parallel, and the number of colors required is small; therefore, BRB is suitable for orthogonal structured grids, which has few synchronization points and can be expected to improve convergence by blocking. The ILU(0)/IC(0) preconditioner in combination with BRB ordering achieves high parallelism in various model and practical problems [14, 15], and its effectiveness in combination with several types of ILU preconditioners has also been verified [16].

As the MILU(0)/MIC(0) preconditioner has a high convergence acceleration effect, it is natural to parallelize them via reordering. However, it has been noted that MILU(0)/MIC(0) can result in factorization breakdowns by zero or very small pivots when combined with several ordering strategies. In particular, in the case of a self-adjoint difference operator discretized based on an evenly spaced grid, combining nodal red-black ordering or parallel wavefront ordering with MILU(0)/MIC(0) preconditioner results in a zero pivot or a pivot on the order of $O(h)$ [17]. This breakdown risk has not been investigated for BRB ordering.

In parallel computing, the use of accelerator devices in computers ensure a higher degree of parallelism is becoming more common to meet the expectations of higher speeds. With regard to parallelization of the PIC method, speeding up the computation using accelerator devices were considered [18, 19, 20]; and a comparison of computational speeds among them shows that GPUs have fast computational speeds [21]. Owing to the high computing performance of the GPU, speeding up the computation in combination with the improved MILU and BRB ordering should be possible. However, these preconditioning methods have been mainly evaluated in a multicore environment on the CPU, and few studies have been conducted on the GPU, which is a manycore device.

In this research, we will accelerate the plasma simulation using PIC method by parallelizing the solution of the Poisson equation, which accounts for the majority the computation time when the PIC method is applied to a manufacturing device. For the solver preconditioner, we use MILU(0), which has good convergence, and for parallelization we use BRB ordering, which can take advantage of the property that blocks made from orthogonal structure grid can be painted in two colors. To use them in combination to speed up computation, their characteristics will be investigated and GPUs will be optimized highly parallel environments.

We first analyze the combination of BRB ordering and the MILU/MIC preconditioner and show the risks when they are applied to the matrix resulting from the discretization of difference operators. In particular, if certain conditions of block division are met, a zero pivot occurs during factorization and an effective preconditioner cannot be obtained. We also show via theoretical

analysis and numerical experiments that this problem can be mitigated by the introduction of perturbations or relaxations. Subsequently, we conduct a GPU parallelization of the solution of simultaneous linear equations with coefficient matrices preconditioned by MILU(0) and BRB ordering with reduced breakdown issues. The implementation measures necessary to bring out the convergence behavior and high performance are examined under the condition of a large number of blocks and a high degree of parallelism. In numerical experiments, we demonstrate the convergence behavior and computation time to obtain a solution of Krylov subspace method solver applied to coefficient matrices with several preconditioners. The coefficient matrices are obtained from the Poisson equation for a 3D magnetron sputtering equipment simulation using PIC method while varying the matrix size. These test problems use a regular grid and a 7-point stencil, and the Poisson equation is discretized using the 3D finite difference method. In addition, we show the convergence of model problems reordered by BRB ordering using various degrees of parallelism.

1.2 Contributions

The contributions of this research are as follows.

BRB ordering breakdown analysis and workaround suggestions: We analyzed the combination of BRB ordering and the MILU(0)/MIC(0) preconditioner and demonstrated that if some block division condition is met when they are applied to the matrix resulting from the discretization of a difference operator, then zero pivots occur during the factorization and no effective preconditioner can be obtained. We also demonstrated that this problem can be alleviated by the introduction of perturbations or relaxations and that the improved method retains the convergence acceleration effect even at high parallelism levels. As a result, the MILU(0)/MIC(0) preconditioner and BRB ordering, which have excellent parallel acceleration, can be used together; the combination of these methods can now be regarded as an efficient, stable, and highly parallel preconditioner.

Development of BRB-ordered parallel solver for GPU: For parallel solvers running on GPU, we proposed a data storage format to favor the so-called coalesced access, which significantly improves the memory throughput. In addition, the calculation can be sped up while maintaining the overall accuracy by using the mixed precision of single precision and double precision for the substitution calculation, thereby taking advantage of the amount of single-precision floating-point data and the high single-precision arithmetic performance of the GPU. We optimized the block division conditions for the manycore configuration considering the block size in each direction and clarified the effect of specifying the parallel granularity on the parallel calculation speed. By using these in combination, a high-speed solver can be implemented in a maintainable manner using OpenACC directives.

Analysis of convergence of BRB ordering: We clarified the fact that the optimum value of the perturbation factor differs depending on the block division scheme of BRB ordering. Furthermore, we showed the sufficient condition for the order of the condition number to be $O(h^{-1})$ is satisfied in perturbed MILU(0) preconditioned coefficient matrix with a small number of blocks. We also showed that even when the number of blocks is large, features that have a

positive impact on convergence appear in the eigenvalue spectrum, revealing the reason for the superior convergence characteristics of BRB ordering.

1.3 Outline of the Thesis

The remainder of this thesis is organized as follows:

Chapter 2
Chapter 2 describes the outline and speedup of the PIC method, a plasma simulation method that incorporates the Poisson solver.

Chapter 3
The first half of Chapter 3 describes the Krylov subspace method used for the solver and the preconditioning method accelerating its convergence. In the second half, we describe parallel computing and an ordering method used to parallelize the preconditioner.

Chapter 4
In Chapter 4, we describe the mechanism by which zero pivot occurs in the combination of parallelization using BRB ordering and the MILU(0)/MIC(0) preconditioner, and we show that the introduction of perturbation or relaxation is effective in alleviating this problem. The contents of this chapter are based on our previous work [22].

Chapter 5
Chapter 5 describes GPU features and a parallelization framework. Then, relaxed MILU(0) preconditioned solver, parallelized by applying BRB ordering, was implemented on GPU. This chapter elaborates on the content of our work [23] and evaluates its effect on the overall PIC method.

Chapter 6
Chapter 6 experimentally analyzes the effect of BRB ordering on the convergence of the perturbed MILU(0)/MIC(0) preconditioner. The content of this chapter is based on our preliminary work [24].

Chapter 7
In Chapter 7, we present our conclusions.

Chapter 2

Plasma Simulation via the Particle-In-Cell Method

Plasma is the fourth phase of matter, whereas the other three are solids, liquids, and gases. Several charged particles of this state exist in the universe, and they are being utilized in industries to micromachine and etch silicon surfaces, gas lasers, and more. Plasma phenomena are complex, and simulations are being used to understand them. Gas phase simulation methods are primarily divided into two categories: those based on particle models that track the trajectory of a large number of individual particles, and those based on fluid models that solve a distribution function smoothed on a grid. A conventional particle-modeling method is the particle-in-cell (PIC) method, which is useful for simulating plasma flows in a low-pressure region. Here, we describe a PIC/Monte Carlo collision (MCC) method [1] that employs the MCC method to handle the collisions within a cell.

2.1 Particle-In-Cell/Monte Carlo Collision Method

The PIC method is a simulation model that considers the plasma to be a collection of charged particles, such as electrons and ions. Because it is difficult to account for all the charged particles and neutral gas molecules, calculations are executed by representing multiple charged particles with a simulated particle called a super-particle. The number of real particles represented by a super-particle is called the weight of the super-particle. The plasma behavior is calculated by tracing the behavior of these super-particles. Super-particles are charged particles with charge q, mass m, position $\boldsymbol{X}$, and velocity $\boldsymbol{V}$. It is assumed that the density and temperature distributions of the neutral gas molecules are not altered during the calculation.

Fig. 2.1 illustrates the basic workflow of the PIC/MCC method. The computational domain is divided into grids (field nodes) with small cells, and the super-particles are arranged according to the density and temperature at time $t = 0$ (Fig. 2.1, top left). The charges of the super-particles are distributed at their respective node points, and the charge allocation ρ at the node points is obtained (Fig. 2.1, upper center). The potential ϕ at the lattice node is calculated by the Poisson equation from the ρ of the node (Fig. 2.1, upper right), whereas the electric field $\boldsymbol{E}$ and the magnetic field $\boldsymbol{B}$ are calculated from the potential are interpolated at the particle position (Fig. 2.1, lower right). The movement of the particles is calculated using the Newtonian

equations of motion:

$$m\frac{\partial \boldsymbol{V}}{\partial t} = \boldsymbol{F}, \tag{2.1.1}$$

$$\frac{\partial \boldsymbol{X}}{\partial t} = \boldsymbol{V}, \tag{2.1.2}$$

$$\boldsymbol{F} = q(\boldsymbol{E} + \boldsymbol{V} \times \boldsymbol{B}). \tag{2.1.3}$$

Secondary electron, ion emissions, annihilation, reflection, or absorption is calculated if a particle reaches a wall boundary during its movement (Fig. 2.1, lower middle). After the calculation of the movement is completed, the reactions between the charged particle and the neutral particle are calculated by the MCC method from the reaction probability, according to the collision cross section (Fig. 2.1, bottom left). Once all the PIC calculations in that time step are completed, the step is increased and the whole process is repeated until the flow reaches a steady-state, or the prespecified time is reached.

The magnetic field is calculated as a static magnetic field based on the Biot-Savart law:

$$\boldsymbol{B} = \frac{\mu}{4\pi}\int_V d^3\boldsymbol{r}_\mathrm{m}\frac{\boldsymbol{j}_\mathrm{m}(\boldsymbol{r}_\mathrm{m}) \times (\boldsymbol{r} - \boldsymbol{r}_\mathrm{m})}{|\boldsymbol{r} - \boldsymbol{r}_\mathrm{m}|^3} \tag{2.1.4}$$

$$\boldsymbol{j}_\mathrm{m} = \frac{1}{\mu_0}(\nabla \times \boldsymbol{M}) \tag{2.1.5}$$

from the magnet arrangement and magnetic characteristics, where $\boldsymbol{r}$, $\boldsymbol{r}_\mathrm{m}$, μ_0, and $\boldsymbol{M}$, respectively, denote the position vector, position of the magnet, permeability of the vacuum, and magnetization of the external magnets with dimension equivalent to $\boldsymbol{B}$.

We use the implicit method based on the study conducted by Vahedi *et al.* [2] for particle motion. The new velocities $\boldsymbol{V}_n$ and positions $\boldsymbol{X}_n$ at time-step n of the super-particles are obtained by

$$\boldsymbol{V}_n = \boldsymbol{V}_{n-1} + \frac{q}{2m}(\boldsymbol{E}_{n-1} + \boldsymbol{V}_{n-1} \times \boldsymbol{B} + \boldsymbol{E}_n + \boldsymbol{V}_n \times \boldsymbol{B})dt, \tag{2.1.6}$$

$$\boldsymbol{X}_n = \boldsymbol{X}_{n-1} + \frac{1}{2}(\boldsymbol{V}_n + \boldsymbol{V}_{n-1})dt, \tag{2.1.7}$$

where dt is the time-step size. Because there are terms on the right-hand side with a time step n that are unknown, the particles are moved using the l-th predicted values $\boldsymbol{V}^l$ and $\boldsymbol{X}^l$ of predictor-corrector loop, while the corresponding electric field $\boldsymbol{E}_n^l$ is derived by solving the Poisson equation. The predicted values of $\boldsymbol{V}^{l+1}$ and $\boldsymbol{X}^{l+1}$ are obtained by

$$\boldsymbol{V}^{l+1} = \boldsymbol{V}_{n-1} + \frac{qdt}{2m}(\boldsymbol{E}_{n-1} + \boldsymbol{V}_{n-1} \times \boldsymbol{B}) + \frac{qdt}{2m}(\boldsymbol{E}_n^l + \boldsymbol{V}_n^l \times \boldsymbol{B}), \tag{2.1.8}$$

$$\boldsymbol{X}^{l+1} = \boldsymbol{X}_{n-1} + \boldsymbol{V}_{n-1}dt + \frac{qdt}{2m}(\boldsymbol{E}_{n-1} + \boldsymbol{V}_{n-1} \times \boldsymbol{B})\frac{dt}{2} + \frac{qdt}{2m}(\boldsymbol{E}_n^l + \boldsymbol{V}_n^l \times \boldsymbol{B})\frac{dt}{2}. \tag{2.1.9}$$

$\boldsymbol{X}_n$ is rewritten as

$$\boldsymbol{X}_n = \tilde{\boldsymbol{X}}_n + \delta\boldsymbol{X}_n \tag{2.1.10}$$

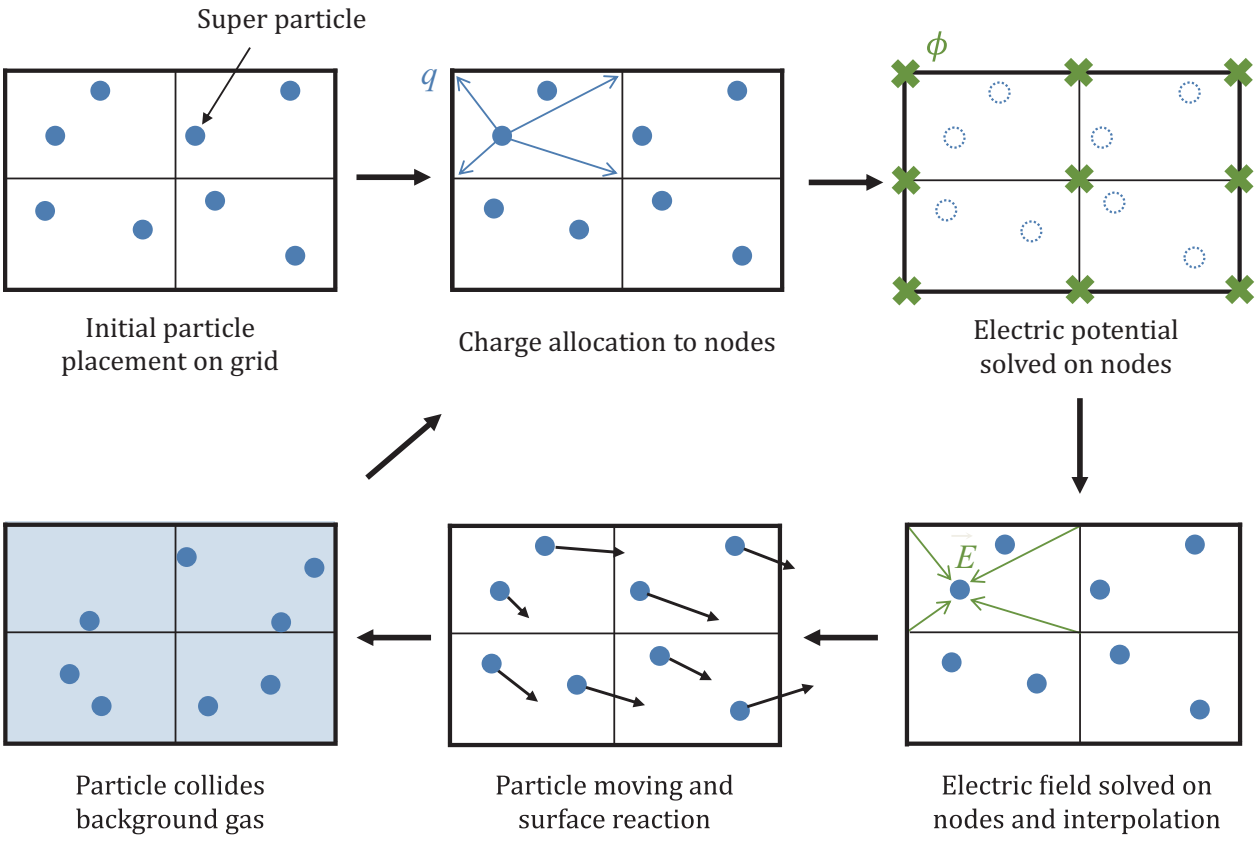

Figure 2.1: Basic flow of PIC/MCC method.

with $\tilde{\boldsymbol{X}}_n$ as the $\boldsymbol{X}$ derived only from time step $n-1$, and $\delta\boldsymbol{X}_n$ representing the part obtained from time step n. For simplicity, Eq. (2.1.6) is substituted into Eq. (2.1.7), then $\tilde{\boldsymbol{B}}$ is ignored and $\boldsymbol{E}(\boldsymbol{X})$ is regarded as the electric field of position $\boldsymbol{X}$; consequently, we obtain

$$\boldsymbol{X}_n = \boldsymbol{X}_{n-1} + \frac{1}{2}(\boldsymbol{V}_{n-1} + \frac{q}{2m}(\boldsymbol{E}_{n-1}(\boldsymbol{X}_{n-1}) + \boldsymbol{E}_n(X_n))dt + \boldsymbol{V}_{n-1})dt \tag{2.1.11}$$

$$= \boldsymbol{X}_{n-1} + \boldsymbol{X}_{n-1}dt + \left(\frac{dt}{2}\right)^2 \frac{q}{m}\boldsymbol{E}_{n-1}(\boldsymbol{X}_{n-1}) + \left(\frac{dt}{2}\right)^2 \frac{q}{m}\boldsymbol{E}_n(\boldsymbol{X}_n), \tag{2.1.12}$$

with the 1st – 3rd terms becoming $\tilde{\boldsymbol{X}}_n$, and the 4th term $\delta\boldsymbol{X}_n$. Using $\delta\boldsymbol{E}_n(\tilde{\boldsymbol{X}}_n)$, which is $\boldsymbol{E}_n$ at the position $\tilde{\boldsymbol{X}}_n$ as an estimated value of $\boldsymbol{E}_n(\boldsymbol{X}_n)$ yields

$$\boldsymbol{X}_n = \tilde{\boldsymbol{X}}_n + \left(\frac{dt}{2}\right)^2 \frac{q}{m}\boldsymbol{E}_n(\tilde{\boldsymbol{X}}_n). \tag{2.1.13}$$

Similarly, the charge density ρ becomes

$$\begin{aligned}
\rho_n &= \tilde{\rho}_n + \delta\rho_n \\
&= \tilde{\rho}_n - \nabla\cdot(\tilde{\rho}_n\delta\boldsymbol{X}_n) \\
&= \tilde{\rho}_n - \nabla\cdot\left[\tilde{\rho}_n\left(\frac{dt}{2}\right)^2\frac{q}{m}\boldsymbol{E}_n(\tilde{\boldsymbol{X}}_n)\right] \\
&= \tilde{\rho}_n - \nabla\cdot\left[\tilde{\rho}_n\left(\frac{dt}{2}\right)^2\frac{q}{m}(\nabla\phi_n)\right]
\end{aligned}$$

and the Poisson equation is rewritten as

$$\begin{aligned}
\frac{\partial^2\phi}{\partial\boldsymbol{X}^2} &= -\frac{\rho}{\epsilon} \\
&= -\frac{\tilde{\rho} - \nabla\cdot[\tilde{\rho}_n(\frac{dt}{2})^2\frac{q}{m}(\nabla\phi_n)]}{\epsilon},
\end{aligned}$$

where ϵ is the permittivity. Using the electrical susceptibility $\chi = (q(dt)^2/4m\epsilon)\rho$, the aforementioned implicit Poisson equation is written as

$$\nabla\cdot(\epsilon[1+\chi]\nabla\phi) = -\rho. \tag{2.1.14}$$

We solve this equation using the finite difference method. For simplicity, we present the formula with a uniform orthogonal grid for a planar two-dimensional coordinate system $(x-y)$. If the finite difference method is applied to the following two-dimensional implicit Poisson equation

$$\frac{\partial}{\partial x}\left(\epsilon[1+\chi]\frac{\partial}{\partial x}\right) + \frac{\partial}{\partial y}\left(\epsilon[1+\chi]\frac{\partial}{\partial y}\right) = -\rho(x,y), \tag{2.1.15}$$

it becomes

$$\begin{aligned}
&\frac{1}{\Delta x}\left(K_{i+1/2,j}\frac{\phi_{i+1,j}-\phi_{i,j}}{\Delta x} - K_{i-1/2,j}\frac{\phi_{i,j}-\phi_{i-1,j}}{\Delta x}\right) \\
&+ \frac{1}{\Delta y}\left(K_{i,j+1/2}\frac{\phi_{i,j+1}-\phi_{i,j}}{\Delta y} - K_{i,j-1/2}\frac{\phi_{i,j}-\phi_{i,j-1}}{\Delta y}\right) = -\rho_{i,j}.
\end{aligned}$$

Here, Δx and Δy represent the respective lattice widths in the x and y directions, the variables with subscripts i and j denote the values on the node, the variables with subscripts $i \pm 1/2$ and $j \pm 1/2$ are the values of the node midpoint, and $K_{i,j}$ represents the value of $\epsilon_{i,j}[1 + \chi_{i,j}]$. By arranging the terms of the above equation for each variable ϕ, we obtain

$$\frac{K_{i+1/2,j}}{\Delta x^2}\phi_{i+1,j} + \frac{K_{i-1/2,j}}{\Delta x^2}\phi_{i-1,j} + \frac{K_{i,j+1/2}}{\Delta y^2}\phi_{i,j+1} + \frac{K_{i,j-1/2}}{\Delta y^2}\phi_{i,j-1}$$
$$- \left(\frac{K_{i+1/2,j}}{\Delta x^2} + \frac{K_{i-1/2,j}}{\Delta x^2} + \frac{K_{i,j+1/2}}{\Delta y^2} + \frac{K_{i,j-1/2}}{\Delta y^2}\right)\phi_{i,j} = -\rho_{i,j}.$$

The boundary condition of the Poisson equation occurs when the outermost plasma flow region is in contact with a solid. The exact location where the potential is calculated in this region depends on the identity of the solid.

Dirichlet conditions are imposed on metallic surfaces. For example, if the known potential $\tilde{\phi}_{i-1,j}$ at node$i-1,j$ is defined,

$$\frac{K_{i+1/2,j}}{\Delta x^2}\phi_{i+1,j} + \frac{K_{i,j+1/2}}{\Delta y^2}\phi_{i,j+1} + \frac{K_{i,j-1/2}}{\Delta y^2}\phi_{i,j-1}$$
$$- \left(\frac{K_{i+1/2,j}}{\Delta x^2} + \frac{K_{i-1/2,j}}{\Delta x^2} + \frac{K_{i,j+1/2}}{\Delta y^2} + \frac{K_{i,j-1/2}}{\Delta y^2}\right)\phi_{i,j} = -\rho_{i,j} - \frac{K_{i-1/2,j}}{\Delta x^2}\tilde{\phi}_{i-1,j}.$$

The same is true when ϕ_{i+1}, j, $\phi_{i,j-1}$, and $\phi_{i,j+1}$ are defined.

Neumann conditions are imposed on non-metallic boundaries. For example, if $\partial\phi/\partial x = E_x(x, y)$ is applied to nodes i, j, and if node $i + 1, j$ is outside the computational domain, we assume an virtual potential $\phi_{i+1,j}$ at a position symmetrical to $i-1, j$ with respect to i, j. When the slope is approximated by a second-order central difference,

$$\phi_{i+1,j} = \phi_{i-1,j} - 2\Delta x E_x(x, y) \tag{2.1.16}$$

is obtained. Accordingly, assuming $K_{i+1/2,j} = K_{i-1/2,j}$ will result in

$$\left(\frac{K_{i+1/2,j}}{\Delta x^2} + \frac{K_{i-1/2,j}}{\Delta x^2}\right)\phi_{i-1,j} + \frac{K_{i,j+1/2}}{\Delta y^2}\phi_{i,j+1} + \frac{K_{i,j-1/2}}{\Delta y^2}\phi_{i,j-1}$$
$$- \left(\frac{K_{i+1/2,j}}{\Delta x^2} + \frac{K_{i-1/2,j}}{\Delta x^2} + \frac{K_{i,j+1/2}}{\Delta y^2} + \frac{K_{i,j-1/2}}{\Delta y^2}\right)\phi_{i,j} = -\rho_{i,j} + \frac{K_{i+1/2,j}}{\Delta} x^2(2\Delta x)E_x(x, y).$$

These equations can be easily extended to three dimensions.

The resulting coefficient matrix is a sparse matrix with up to five elements per row in two dimensions, and up to seven elements in three dimensions. In the case of unequally spaced nodes, the coefficient matrix becomes nonsymmetric.

We repeat the particle motion and Poisson equation solving until ϕ is converged by the predictor-corrector loop.

2.2 Application and Acceleration of the PIC method

The PIC method has been utilized to understand plasma phenomena, and its effectiveness has been examined by reproducing various phenomena and comparison with experimental values [1, 25]. Several methods for accelerating the PIC method have also been investigated

with the increase in computer performance [26, 18, 21]. Sparse grid combination [27] and load distribution [28] are some of the acceleration techniques, and recently, some studies have considered approaches that involve tiling on GPU [19] and particle partitioning by offloading computationally-intensive workloads such as particle motion and interpolation into available GPUs [20]. With the use of acceleration techniques, large-scale PIC simulations using a large number of node points and super-particles, such as fusion reactors [29] and cosmological systems [30, 31], have become feasible, in addition to the reproduction of laboratory phenomena.

We investigate a magnetron sputter model as an industrial application of the PIC method. Several cases of the PIC method for magnetron sputtering have been verified to reproduce experimental results [32, 33, 34]. Magnetron sputtering is a key application adopted in the development of industrial manufacturing equipment that utilizes this phenomenon because solutions can be obtained in a relatively stable manner via the PIC method. However, the models that can be analyzed are limited because the computational cost increases significantly with the increase in device size and plasma density. Therefore, there is a need for faster magnetron sputtering simulation via PIC method.

For a preliminary evaluation of the computational cost, we employed a test problem modeling a typical 3D DC magnetron sputtering apparatus, with its shape and size close to that of an actual device. Fig. 2.2 presents a schematic of the model. The static magnetic field illustrated in Fig. 2.3 was used as the magnetic field distribution at the grid point. Argon 1Pa was uniformly distributed as an initial ambient gas configuration. The electrode has secondary electron emission coefficients of 0.02 and -200 V. The sidewalls and substrate were grounded to 0 V. All their boundaries were metal surfaces treated as Dirichlet boundaries. In addition, all grid points were assumed to have a vacuum permittivity ϵ_0. The device size was 60 mm $\times$ 60 mm $\times$ 30 mm, as presented in Fig. 2.4, and a 1 mm uniform orthogonal mesh were formed.

The coefficient matrix of the Poisson equation for this model becomes a symmetric sparse matrix. It becomes nonsymmetric if a non-uniform orthogonal grid is used near the boundaries to increase the computational accuracy. To address this situation, our simulation code used a solution method and preconditioning that can treat nonsymmetric matrices. Specifically, we used the BiCGSTAB method known for its superior efficiency and stability (Section 3.1.4), with the MILU(0) preconditioner (Section 3.2.1), among nonsymmetric Krylov subspace solvers.

The implicit PIC method illustrated in Section 2.1 was implemented with Fortran90, and all floating-point data and operations in the code were in double precision; further, we performed MPI parallelization via the particle decomposition method. An Intel compiler and MPICH [35] were used. The computational environment was an NEC Express5800/R120b-1 with an Intel Xeon CPU X5675@3.06 GHz (6 cores, 12 threads) $\times$2.

Fig. 2.5 presents the proportion of the calculation time for 60,000 steps of PIC simulation taken by each subroutine when calculated across 8 parallel cores on a single node. The overall computational time was 30.5 h. The calculation time of the predictor-corrector loop enclosed by the dashed line in Fig. 2.5 was the largest, occupying 83% of the total time. Within the predictor-corrector loop, deriving the potential from the Poisson equation took the longest, taking up 57% of the total. Accordingly, some studies report that the computation of the potential in the PIC method is expensive [36, 27].

In this study, the potential was determined using an approach that accelerates the solution of simultaneous linear equations with Krylov subspace methods. The next several Chapters discuss the linear solver, and the results are applied to the PIC method in Chapter 5.

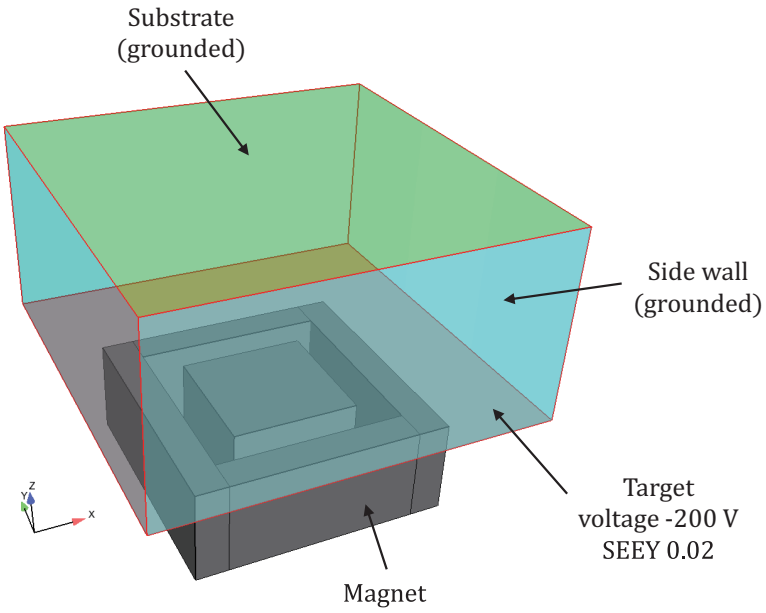

Figure 2.2: 3D magnetron sputter model.

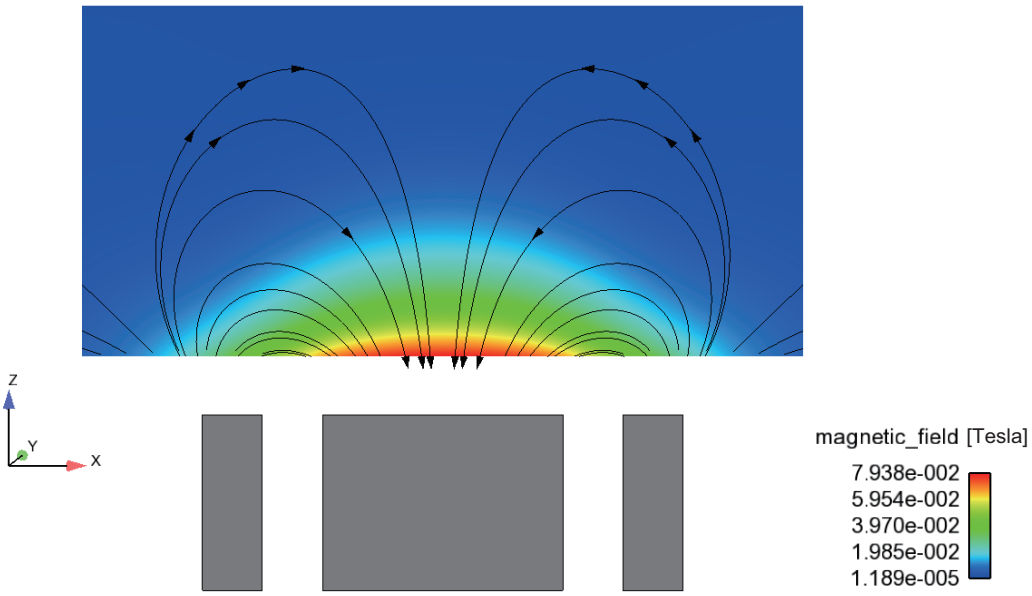

Figure 2.3: Magnetic field on midpoint cut plane in the y-axis direction.

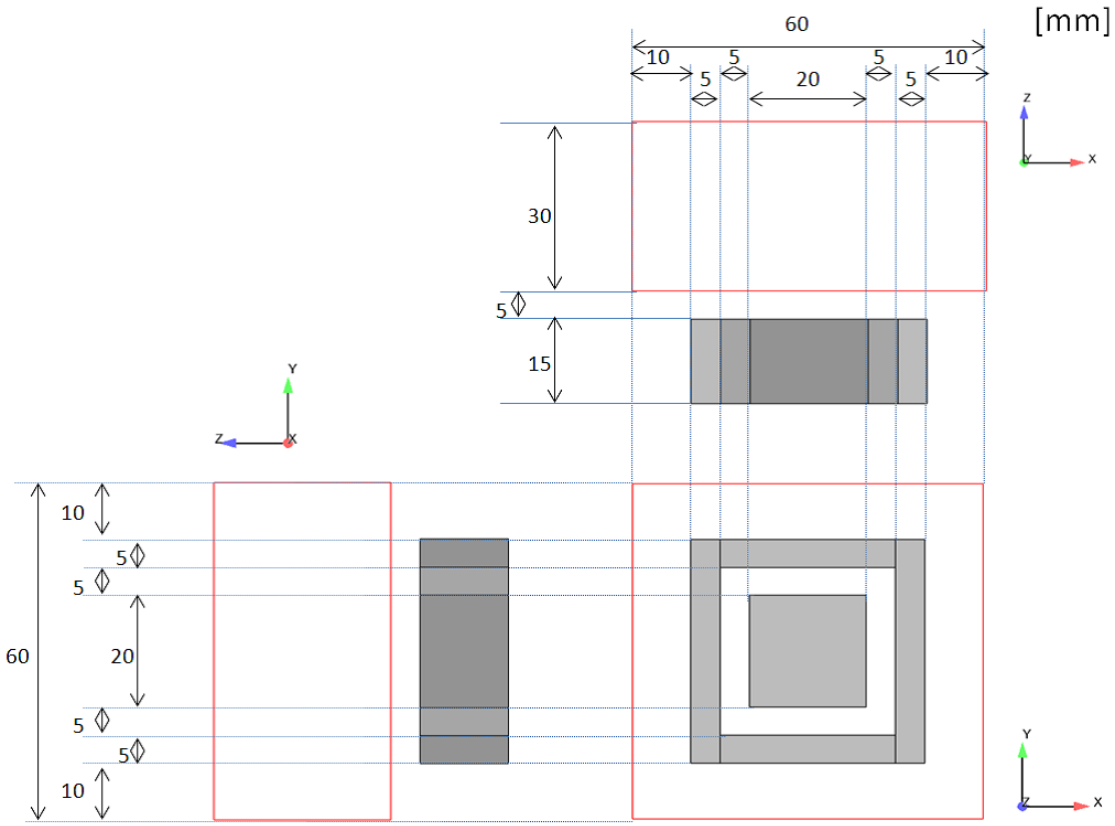

Figure 2.4: Geometry size of 3D magnetron sputter model.

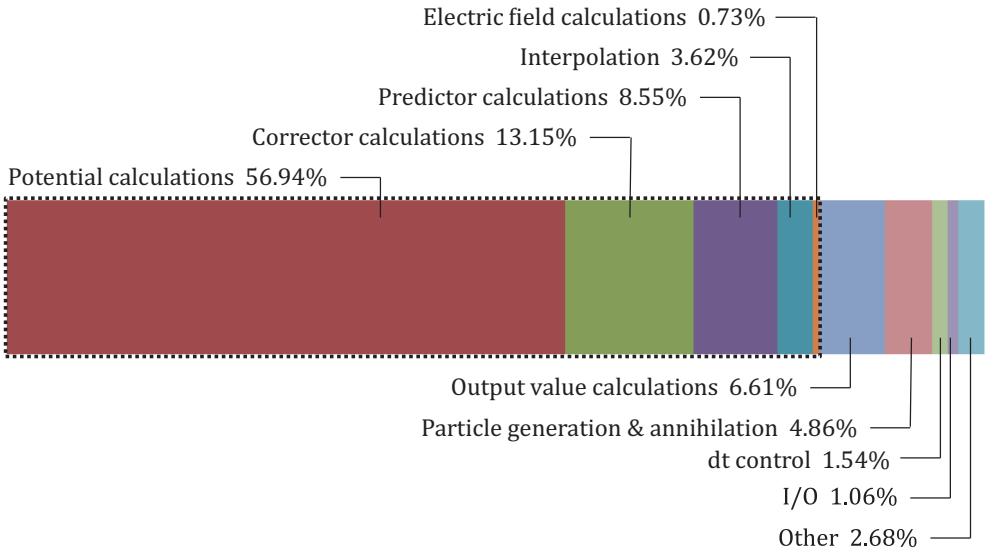

Figure 2.5: Ratio of computational time for each subroutine.

Chapter 3

Krylov Subspace Methods for Linear Simultaneous Equations and Preconditioning Techniques

Finding an approximate solution to partial differential equations (PDEs), which describe natural and engineering phenomena by discretizing them using the finite element method (FEM) or finite difference method (FDM), involves solving a large-scale simultaneous linear equation $A\boldsymbol{x} = \boldsymbol{b}$. This simultaneous linear equation has a sparse coefficient matrix A, which is large-scale and most of its elements are zero.

When the problem size is small, the direct solution method is effective. Meanwhile, when the scale of the problem is large, the iterative solution method, which can efficiently obtain the required accuracy in terms of storage capacity and computational complexity, is used. There are two types of iterative methods: the stationary method, which updates the approximation solution by a linear recurrence relation, and the nonstationary method, which is based on the generation of orthogonal vector sequences and orthogonal polynomials. The Krylov subspace method is a nonstationary method for creating orthogonal vector sequences. Conjugate gradient methods are applied to symmetric positive definite matrices, and derivatives exist for nonsymmetric matrices.

Accurately predicting the convergence of the iterative methods is difficult, but superficial error limits can often be obtained. The spectral condition number κ of a symmetric positive definite matrix A is given by $\kappa = \lambda_{\max}/\lambda_{\min}$, where $\lambda_{\max}$ and $\lambda_{\min}$ are the maximum and minimum eigenvalues of A, respectively. To find the upper bound of the difference between the exact solution $\boldsymbol{x}^*$ and the approximation solution $\boldsymbol{x}^{(n)}$, obtained at the n-th iteration of the conjugate gradient method for the linear equation $A\boldsymbol{x} = \boldsymbol{b}$, a bound on the A-norm of the error can be used for the initial approximation solution $\boldsymbol{x}^{(0)}$:

$$\sqrt{(\boldsymbol{x}^{(n)} - \boldsymbol{x}^*, A(\boldsymbol{x}^{(n)} - \boldsymbol{x}^*))} \leq 2\left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^n \sqrt{(\boldsymbol{x}^{(0)} - \boldsymbol{x}^*, A(\boldsymbol{x}^{(0)} - \boldsymbol{x}^*))} \tag{3.0.1}$$

It is clear from this equation that the number of iterations for the relative error to reach the small value of ε is roughly proportional to $\sqrt{\kappa}$. For an elliptical second-order PDE, the coefficient matrix A typically has the condition number $\kappa = O(h^{-2})$, where h is the mesh width used for discretization, and the number of iterations is expected to be $O(h^{-1})$ for the conjugate gradient method without preconditioning.

As the rate at which the iterative methods converge depends on the eigenvalue spectrum of the coefficient matrix, these methods often require a matrix called a preconditioner, which transforms the coefficient matrix into a matrix with a better spectrum. Using a good preconditioner improves the convergence of the iterative method and fully recovers the extra cost required to construct and apply the preconditioner. In fact, the iterative method may not converge without a preconditioner.

The effect of the preconditioner can be estimated by examining the proximity eigenvalues of the corresponding Lanczos process. Consider a case in which the eigenvalues at both ends of the minimum and maximum sides of the coefficient matrix $K^{-1}A$ are sufficiently separated. Then, the conjugate gradient method for solving a linear system, with $K^{-1}A$ as the coefficient matrix, eliminates the error components in the eigenvector direction corresponding to the eigenvalues at both ends, and then proceeds as if these eigenvalues are non-existent. Hence, the rate of convergence depends on a reduced system with a small spectral condition number [37, 38]. It is also known that convergence improves when eigenvalues form one or more clusters [39, 40].

3.1 Krylov Subspace Methods

In general, for the matrix $A \in \mathbb{R}^{m\times m}$ and vector $\boldsymbol{y} \in \mathbb{R}^m$, the vector sequence $\{\boldsymbol{y}, A\boldsymbol{y}, A^2\boldsymbol{y}, \ldots\}$ generated by multiplying $\boldsymbol{y}$ with the power of A, is called the Krylov sequence. The subspace $K_n(A;\boldsymbol{y})$, called the Krylov subspace, of order n is defined as the subspace spanned by the first n vectors:

$$K_n(A;\boldsymbol{y}) = \mathrm{span}(\boldsymbol{y}, A\boldsymbol{y}, \ldots, A^{n-1}\boldsymbol{y}). \tag{3.1.1}$$

In the Krylov subspace $K_n(A;\boldsymbol{r}^{(0)})$ formed from $\boldsymbol{r}^{(0)}$, with the initial approximation solution as $\boldsymbol{x}^{(0)}$, and the initial residual as $\boldsymbol{r}^{(0)} = \boldsymbol{b} - A\boldsymbol{x}^{(0)}$. An iterative solution method, called the Krylov subspace method, generates a sequence of approximation solutions $\boldsymbol{x}^{(n)}$ satisfying $\boldsymbol{x}^{(n)} - \boldsymbol{x}^{(0)} \in K_n(A;\boldsymbol{r}^{(0)})$.

As $K_1(A;\boldsymbol{r}^{(0)}) \subseteq K_2(A;\boldsymbol{r}^{(0)}) \subseteq \ldots \subseteq K_n(A;\boldsymbol{r}^{(0)})$, the approximation solution is updated whereas the space to be searched is expanded. Because the condition on the space alone does not determine the approximate solution $\boldsymbol{x}^{(n)}$, we have the freedom to choose any vector in the Krylov subspace $K_n(A;\boldsymbol{r}^{(0)})$ as $\boldsymbol{x}^{(n)}$. Three types of principles, namely, residual minimization, which chooses $\boldsymbol{x}^{(n)}$ in $K_n(A;\boldsymbol{r}^{(0)})$ so that the norm of the residual $\boldsymbol{r}^{(n)} = A\boldsymbol{x}^{(n)} - \boldsymbol{b}$ is minimized, the Ritz-Galerkin method, which defines $\boldsymbol{x}^{(n)}$ using the orthogonal condition

$$\boldsymbol{r}^{(n)} \perp K_n(A;\boldsymbol{r}^{(0)}), \tag{3.1.2}$$

and the Petrov-Galerkin method, which defines $\boldsymbol{x}^{(n)}$ using the orthogonal condition

$$\boldsymbol{r}^{(n)} \perp L_n \tag{3.1.3}$$

with some appropriate subspace sequence $L_1 \subseteq L_2 \subseteq \ldots \subseteq L_n$ of $\mathbb{R}^m$, are used as conditions imposed on the residual $\boldsymbol{r}^{(n)}$ corresponding to the approximate solution $\boldsymbol{x}^{(n)}$.

3.1.1 Generating an Orthonormal Basis for Krylov Subspaces

Consider expressing the approximation solution $\boldsymbol{x}^{(n)}$ as a linear combination of the basis vectors of the Krylov subspace $K_n(A;\boldsymbol{r}^{(0)})$. In this case, using an orthonormal basis is advantageous in terms of numerical stability and ease of theoretical derivation of the solution method.

Orthogonalization of the vector sequence $\{\boldsymbol{r}^{(0)}, A\boldsymbol{r}^{(0)}, \ldots, A^{n-1}\boldsymbol{r}^{(0)}\}$ using the Gram-Schmidt method can yield an orthonormal system $\boldsymbol{q}^{(1)}, \boldsymbol{q}^{(2)}, \ldots, \boldsymbol{q}^{(n)}$ of the Krylov subspace

$$K_n(A; \boldsymbol{r}^{(0)}) = \mathrm{span}(\boldsymbol{r}^{(0)}, A\boldsymbol{r}^{(0)}, \ldots, A^{n-1}\boldsymbol{r}^{(0)}), \tag{3.1.4}$$

which is called the Arnoldi process, as shown in Algorithm 1. In Algorithm 1, lines 3 to 5 represent the orthogonalization process, and lines 6 to 7 represent the normalization process.

Algorithm 1 Arnoldi process

1: $\boldsymbol{q}^{(1)} = \boldsymbol{r}^{(0)}/\|\boldsymbol{r}^{(0)}\|_2$
2: **for** $n = 1, 2, \ldots$ **do**
3: **for** $i = 1$ to n **do**
4: $h_{in} = (\boldsymbol{q}^{(i)}, A\boldsymbol{q}^{(n)})$
5: **end for**
6: $h_{n+1,n} = \|A\boldsymbol{q}^{(n)} - \Sigma_{i=1}^{n} h_{in}\boldsymbol{q}^{(i)}\|_2$
7: $\boldsymbol{q}^{(n+1)} = (A\boldsymbol{q}^{(n)} - \Sigma_{i=1}^{n} h_{in}\boldsymbol{q}^{(i)})/h_{n+1,n}$
8: **end for**

From the Arnoldi process, we obtain $\boldsymbol{q}^{(i)} = h_{1i}\boldsymbol{q}^{(1)} + h_{2i}\boldsymbol{q}^{(2)} + \ldots + h_{i+1,i}\boldsymbol{q}^{(i+1)}$. Therefore, if the $m \times n$ orthogonal matrix $Q_n = [\boldsymbol{q}^{(1)}|\boldsymbol{q}^{(2)}|\ldots|\boldsymbol{q}^{(n)}]$ and the $(n+1) \times n$ matrix

$$\tilde{H}_n = \begin{bmatrix} h_{11} & h_{12} & \ldots & h_{1n} \\ h_{21} & h_{22} & \ldots & h_{2n} \\ & \ddots & \ddots & \vdots \\ & & h_{n,n-1} & h_{nn} \\ & & & h_{n+1,n} \end{bmatrix} \tag{3.1.5}$$

are defined, $AQ_n = Q_{n+1}\tilde{H}_n$ holds. When $Q_n^\top$ is multiplied from the left, because the columns of Q_n are mutually orthogonal, the $n \times n$ Hessenberg matrix is obtained as follows:

$$H_n = Q_n^\top A Q_n = Q_n^\top Q_{n+1}\tilde{H}_n = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ & \ddots & \ddots & \vdots \\ & & 0 & 1 \\ & & & 0 \end{bmatrix} \tilde{H}_n = \begin{bmatrix} h_{11} & h_{12} & \ldots & h_{1n} \\ h_{21} & h_{22} & \ldots & h_{2n} \\ & \ddots & \ddots & \vdots \\ & & h_{n,n-1} & h_{nn} \end{bmatrix} \tag{3.1.6}$$

This can be seen as a matrix obtained by projecting A onto $\mathrm{span}(\boldsymbol{q}^{(1)}, \boldsymbol{q}^{(2)}, \ldots, \boldsymbol{q}^{(n)}) = K_n(A; \boldsymbol{r}^{(0)})$.

Next, considering the case where A is a symmetric matrix, H_n also becomes a symmetric matrix, and as $h_{in} = (A\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(n)}) = 0$ with $i \leq n-2$ from the orthogonality of $\boldsymbol{q}^{(1)}, \boldsymbol{q}^{(2)}, \ldots, \boldsymbol{q}^{(n)}$, H_n becomes a symmetric tridiagonal matrix. That is, $A\boldsymbol{q}^{(i)}$ must be orthogonalized only against $\boldsymbol{q}^{(i)}$ and $\boldsymbol{q}^{(i-1)}$. The Arnoldi process for symmetric matrices using this is shown in Algorithm 2, which is called the Lanczos process.

Using the Lanczos process, $\boldsymbol{q}^{(n+1)}$ can only be obtained from $\boldsymbol{q}^{(n)}$ and $\boldsymbol{q}^{(n-1)}$. That is, if A is a symmetric matrix, and $\boldsymbol{u}^{(n)} = U_n(A)\boldsymbol{u}^{(0)}$ generated by the n-th degree polynomial $U_n(z)$ satisfies the orthogonal condition, then there exist real numbers ξ_n, η_n, ζ_n and the three-term recurrence relation

$$U_{n+1}(z) = \xi_n z U_n(z) + \eta_n U_n(z) + \zeta_n U_{n-1}(z) \tag{3.1.7}$$

holds. This is called the Lanczos principle and is used to derive the conjugate gradient (CG) method shown in Section 3.1.2.

Algorithm 2 Lanczos process

1: $\boldsymbol{q}^{(0)} = \boldsymbol{0}, \boldsymbol{q}^{(1)} = \boldsymbol{r}^{(0)}/\|\boldsymbol{r}^{(0)}\|_2$
2: **for** $n = 1, 2, \ldots$ **do**
3: $\quad h_{n-1,n} = (\boldsymbol{q}^{(n-1)}, A\boldsymbol{q}^{(n)})$
4: $\quad h_{nn} = (\boldsymbol{q}^{(n)}, A\boldsymbol{q}^{(n)})$
5: $\quad h_{n+1,n} = \|A\boldsymbol{q}^{(n)} - h_{n-1,n}\boldsymbol{q}^{(n-1)} - h_{nn}\boldsymbol{q}^{(n)}\|_2$
6: $\quad \boldsymbol{q}^{(n+1)} = (A\boldsymbol{q}^{(n)} - h_{n-1,n}\boldsymbol{q}^{(n-1)} - h_{nn}\boldsymbol{q}^{(n)})/h_{n+1,n}$
7: **end for**

3.1.2 Conjugate Gradient (CG) Method

The CG method for symmetric matrices is derived from the Lanczos principle using the spatial conditions of the Krylov subspace and the orthogonal conditions (3.1.2) of the Ritz-Galerkin method. In the CG method, the residual $\boldsymbol{r}^{(n)} = \boldsymbol{b} - A\boldsymbol{x}^{(n)}$ corresponding to the approximation solution $\boldsymbol{x}^{(n)}$ at the n-th iteration is expressed as $\boldsymbol{r}^{(n)} = R_n(A)\boldsymbol{r}^{(0)}$ by the n-th degree polynomial $R_n(z)$. At this time,

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - A^{-1}(R_{n+1}(A) - R_n(A))\boldsymbol{r}^{(0)} \tag{3.1.8}$$

holds; thus, when setting

$$X_{n+1}(z) = (R_{n+1}(z) - R_n(z))/z, \tag{3.1.9}$$

it becomes

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - X_{n+1}(A)\boldsymbol{r}^{(0)}. \tag{3.1.10}$$

Using the Lanczos principle (3.1.11), $R_n(z)$ and $(R_{n+1}(z) - R_n(z))/z$ are obtained such that the residual $\boldsymbol{r}^{(n)} = R_n(A)\boldsymbol{r}^{(0)}$ becomes orthogonal. From $R_n(0) = 1$, as $\eta_n + \zeta_n = 1$, $R_{n+1}(z)$ becomes

$$R_{n+1}(z) = \xi_n z R_n(z) + (1 - \zeta_n)R_n(z) + \zeta_n R_{n-1}(z), \tag{3.1.11}$$

and by rewriting this into a simultaneous binary recurrence relation,

$$\frac{R_{n+1}(z) - R_n(z)}{\xi_n z} = R_n(z) - \frac{\zeta_n \xi_{n-1}}{\xi_n}\frac{R_n(z) + R_{n-1}(z)}{\xi_{n-1} z} \tag{3.1.12}$$

and setting

$$P_n(z) = \frac{R_{n+1}(z) - R_n(z)}{\xi_n z}, \quad \alpha_n = -\xi_n, \quad \beta_n = -\frac{\zeta_n \xi_{n-1}}{\xi_n}, \tag{3.1.13}$$

the recurrence relations

$$P_n(z) = R_n(z) + \beta_{n-1}P_{n-1}(z) \tag{3.1.14}$$

$$R_{n+1}(z) = R_n(z) + \alpha_n z P_n(z) \tag{3.1.15}$$

are obtained, and thus

$$\boldsymbol{p}^{(n)} = \boldsymbol{r}^{(n)} + \beta_{n-1}\boldsymbol{p}^{(n-1)} \tag{3.1.16}$$

$$\boldsymbol{r}^{(n+1)} = \boldsymbol{r}^{(n)} - \alpha_n A\boldsymbol{p}^{(n)}. \tag{3.1.17}$$

From the recurrence relation of $R_{n+1}(z)$ (3.1.15) and $(R_{n+1}(A)\boldsymbol{r}^{(0)}, P_n(A)\boldsymbol{r}^{(0)}) = 0$, the coefficient α_n becomes

$$\alpha_n = \frac{(R_n(A)\boldsymbol{r}^{(0)}, P_n(A)\boldsymbol{r}^{(0)})}{(AP_n(A)\boldsymbol{r}^{(0)}, P_n(A)\boldsymbol{r}^{(0)})} = \frac{(\boldsymbol{r}^{(n)}, \boldsymbol{p}^{(n)})}{(A\boldsymbol{p}^{(n)}, \boldsymbol{p}^{(n)})}. \tag{3.1.18}$$

Furthermore, from the recurrence relation of $P_n(z)$ (3.1.14) and $(P_n(A)\boldsymbol{r}^{(0)}, AP_{n-1}(A)\boldsymbol{r}^{(0)}) = 0$, the coefficient β_{n-1} becomes

$$\beta_{n-1} = -\frac{(R_n(A)\boldsymbol{r}^{(0)}, AP_{n-1}(A)\boldsymbol{r}^{(0)})}{(P_{n-1}(A)\boldsymbol{r}^{(0)}, AP_n(A)\boldsymbol{r}^{(0)})} = \frac{(\boldsymbol{r}^{(n)}, A\boldsymbol{p}^{(n-1)})}{(\boldsymbol{p}^{(n-1)}, A\boldsymbol{p}^{(n-1)})}. \tag{3.1.19}$$

By defining $K\boldsymbol{z}^{(n)} = \boldsymbol{r}^{(n)}$ for preconditioning, and $\boldsymbol{q}^{(n)} = A\boldsymbol{p}^{(n)}$ as the vector that holds the result of the matrix vector product, and setting $\rho_n = (\boldsymbol{r}^{(n)}, \boldsymbol{z}^{(n)})$, the algorithm of the CG method with preprocessing shown in Algorithm 3 is obtained.

Algorithm 3 CG with preconditioner K

Compute $\boldsymbol{r}^{(0)} = \boldsymbol{b} - A\boldsymbol{x}^{(0)}$ for some initial guess $\boldsymbol{x}^{(0)}$
for $n = 1, 2, \ldots,$ until convergence **do**
  solve $K\boldsymbol{z}^{(n-1)} = \boldsymbol{r}^{(n-1)}$
  $\rho_{n-1} = (\boldsymbol{r}^{(n-1)}, \boldsymbol{z}^{(n-1)})$
  if $n = 1$ **then**
    $\boldsymbol{p}^{(n)} = \boldsymbol{z}^{(n-1)}$
  else
    $\beta_{n-1} = \rho_{n-1}/\rho_{n-2}$
    $\boldsymbol{p}^{(n)} = \boldsymbol{z}^{(n-1)} + \beta_{n-1}\boldsymbol{p}^{(n-1)}$
  end if
  $\boldsymbol{q}^{(n)} = A\boldsymbol{p}^{(n)}$
  $\alpha_n = \rho_{n-1}/(\boldsymbol{p}^{(n)}, \boldsymbol{q}^{(n)})$
  $\boldsymbol{x}^{(n)} = \boldsymbol{x}^{(n-1)} + \alpha_n\boldsymbol{p}^{(n)}$
  $\boldsymbol{r}^{(n)} = \boldsymbol{r}^{(n-1)} - \alpha_n\boldsymbol{q}^{(n)}$
  if norm of $\boldsymbol{r}^{(n)}$ is small enough **then**
    stop
  end if
end for

3.1.3 BiConjugate Gradient (BiCG) Method

For a nonsymmetric matrix, a sequence of residuals that satisfy the orthogonal condition cannot be generated by the approach used in the CG method. The BiCG method is a Petrov-Galerkin method that considers the auxiliary equation $A^\top\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}}$ and the shadow residual $\tilde{\boldsymbol{r}} = \tilde{\boldsymbol{b}} - A^\top\tilde{\boldsymbol{x}}$, and has an orthogonal condition

$$\boldsymbol{r}^{(n)} \perp K_n(A^\top; \tilde{\boldsymbol{r}}^{(0)}) \tag{3.1.20}$$

which corresponds to setting $L_n = K_n(A^\top; \tilde{\boldsymbol{r}}^{(0)})$ in the orthogonal condition (3.1.3). Consider a new Krylov subspace

$$K_n(A^\top; \tilde{\boldsymbol{r}}^{(0)}) = \mathrm{span}(\tilde{\boldsymbol{r}}^{(0)}, (A^\top)\tilde{\boldsymbol{r}}^{(0)}, \ldots, (A^\top)^{n-1}\tilde{\boldsymbol{r}}^{(0)}), \tag{3.1.21}$$

which uses residual biorthogonalization instead of orthogonalization.

Here, we consider constructing a biorthogonal system of $\boldsymbol{r}^{(n)} = R_n(A)\boldsymbol{r}^{(0)}$ and $\tilde{\boldsymbol{r}}^{(n)} = R_n(A)\tilde{\boldsymbol{r}}^{(0)}$ from the n-th order residual polynomial $R_n(z)$ and the initial residual. When the biorthogonal condition is satisfied, $R_n(z)$ satisfies the three-term recurrence relation (3.1.11)

according to the generalized Lanczos principle. The recurrence relation of $R_{n+1}(z)$ (3.1.15), and the recurrence relation of $P_n(z)$ (3.1.14), are derived similarly to the CG method.

As the coefficients α_n and β_{n-1} are obtained by

$$\alpha_n = \frac{(R_n(A^\top)\tilde{\boldsymbol{r}}^{(0)}, R_n(A)\boldsymbol{r}^{(0)})}{(P_n(A^\top)\tilde{\boldsymbol{r}}^{(0)}, AP_n(A)\boldsymbol{r}^{(0)})} = \frac{(\tilde{\boldsymbol{r}}^{(n)}, \boldsymbol{r}^{(n)})}{(\tilde{\boldsymbol{p}}^{(n)}, A\boldsymbol{p}^{(n)})} \tag{3.1.22}$$

$$\beta_{n-1} = -\frac{(R_n(A^\top)\tilde{\boldsymbol{r}}^{(0)}, R_n(A)\boldsymbol{r}^{(0)})}{(R_{n-1}(A^\top)\tilde{\boldsymbol{r}}^{(0)}, R_{n-1}(A)\boldsymbol{r}^{(0)})} = \frac{(\tilde{\boldsymbol{r}}^{(n)}, \boldsymbol{r}^{(n)})}{(\tilde{\boldsymbol{r}}^{(n-1)}, \boldsymbol{r}^{(n-1)})} \tag{3.1.23}$$

from biorthogonality, the approximation solution $\boldsymbol{x}^{(n)}$ can be computed sequentially using

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + \alpha_n P_n(A)\boldsymbol{r}^{(0)}. \tag{3.1.24}$$

From the above relational expression, the BiCG method can be derived by creating the recurrence relations for

$$\boldsymbol{r}^{(n)} = R_n(A)\boldsymbol{r}^{(0)}, \tilde{\boldsymbol{r}}^{(n)} = R_n(A)\tilde{\boldsymbol{r}}^{(0)} \tag{3.1.25}$$

$$\boldsymbol{p}^{(n)} = P_n(A)\boldsymbol{r}^{(0)}, \tilde{\boldsymbol{p}}^{(n)} = P_n(A)\tilde{\boldsymbol{r}}^{(0)} \tag{3.1.26}$$

and $\boldsymbol{x}^{(n)}$.

By defining $K\boldsymbol{z}^{(n)} = \boldsymbol{r}^{(n)}$ and $K^\top\tilde{\boldsymbol{z}}^{(n)} = \tilde{\boldsymbol{r}}^{(n)}$ for preconditioning, and $\boldsymbol{q}^{(n)} = A\boldsymbol{p}^{(n)}$ and $\tilde{\boldsymbol{q}}^{(n)} = A^\top\tilde{\boldsymbol{p}}^{(n)}$, as the vectors that hold the result of the matrix vector product, and setting $\rho_n = (\tilde{\boldsymbol{r}}^{(n)}, \boldsymbol{z}^{(n)})$, the algorithm of the BiCG method, with the preconditioner shown in Algorithm 4 is obtained.

The BiCG method can be applied to nonsymmetric matrices, but it requires matrix-vector products of the transposed matrices, as shown in lines 5 and 16. Moreover, its convergence behavior is often irregular, and there is a risk of breakdown when the inner products calculated in lines 6 and 17 become close to 0.

3.1.4 BiConjugate Gradient Stabilized (BiCGSTAB) Method

There is a product-type iterative method that accelerates convergence using the n-th degree polynomial $U_n(z)$, while avoiding the computing operation with $A^\top$ of the BiCG method. The residual of the product-type iterative method is defined as $\boldsymbol{r}^{(n)} = U_n(A)\boldsymbol{r}^{(n)}_{\mathrm{BiCG}} = U_n(A)R_n(A)\boldsymbol{r}^{(0)}$. The conjugate gradient squared (CGS) method updates the residual vector as $\boldsymbol{r}^{(n)} = (R_n(A))^2\boldsymbol{r}^{(0)}$. As a result, the computing operation with the transposed matrix is avoided, but irregular convergence behavior appears, which is sometimes more severe than that of the BiCG method as the square of the residual polynomial of the BiCG method is used.

The BiCGSTAB method was developed to solve nonsymmetric simultaneous linear equations while avoiding the irregular convergence patterns that occur in the CGS method. The residual vector is updated as $\boldsymbol{r}^{(n)} = S_n(A)R_n(A)\boldsymbol{r}^{(0)}$. Here, the n-th order polynomial $S_n(z)$ is recursively defined at each step, with the aim of stabilizing the convergence behavior. Similarly, $\boldsymbol{p}$ is also updated by $\boldsymbol{p}^{(n)} = S_n(A)P_n(A)\boldsymbol{r}^{(0)}$. $S_n(z)$ is defined by a simple recursive formula:

$$S_{n+1}(z) = (1 - \omega_n)S_n(z) \tag{3.1.27}$$

Algorithm 4 BiCG with preconditioner K

Compute $\boldsymbol{r}^{(0)} = \boldsymbol{b} - A\boldsymbol{x}^{(0)}$ for some initial guess $\boldsymbol{x}^{(0)}$
$\tilde{\boldsymbol{r}}^{(0)} = \boldsymbol{r}^{(0)}$
for $n = 1, 2, \ldots,$ until convergence **do**
  solve $K\boldsymbol{z}^{(n-1)} = \boldsymbol{r}^{(n-1)}$
  solve $K^\top \tilde{\boldsymbol{z}}^{(n-1)} = \tilde{\boldsymbol{r}}^{(n-1)}$
  $\rho_{n-1} = (\tilde{\boldsymbol{r}}^{(n-1)}, \boldsymbol{z}^{(n-1)})$
  if $n = 1$ **then**
    $\boldsymbol{p}^{(n)} = \boldsymbol{z}^{(n-1)}$
    $\tilde{\boldsymbol{p}}^{(n)} = \tilde{\boldsymbol{z}}^{(n-1)}$
  else
    $\beta_{n-1} = \rho_{n-1}/\rho_{n-2}$
    $\boldsymbol{p}^{(n)} = \boldsymbol{z}^{(n-1)} + \beta_{n-1}\boldsymbol{p}^{(n-1)}$
    $\tilde{\boldsymbol{p}}^{(n)} = \tilde{\boldsymbol{z}}^{(n-1)} + \beta_{n-1}\tilde{\boldsymbol{p}}^{(n-1)}$
  end if
  $\boldsymbol{q}^{(n)} = A\boldsymbol{p}^{(n)}$
  $\tilde{\boldsymbol{q}}^{(n)} = A^\top \tilde{\boldsymbol{p}}^{(n)}$
  $\alpha_n = \rho_{n-1}/(\tilde{\boldsymbol{p}}^{(n)}, \boldsymbol{q}^{(n)})$
  $\boldsymbol{x}^{(n)} = \boldsymbol{x}^{(n-1)} + \alpha_n \boldsymbol{p}^{(n)}$
  $\boldsymbol{r}^{(n)} = \boldsymbol{r}^{(n-1)} - \alpha_n \boldsymbol{q}^{(n)}$
  $\tilde{\boldsymbol{r}}^{(n)} = \tilde{\boldsymbol{r}}^{(n-1)} - \alpha_n \tilde{\boldsymbol{q}}^{(n)}$
  if norm of $\boldsymbol{r}^{(n)}$ is small enough **then**
    stop
  end if
end for

using the free parameter ω_n. The coefficients α_n and β_n are rewritten as

$$\begin{aligned}\alpha_n &= \frac{(\tilde{\boldsymbol{r}}^{(n)}_{\mathrm{BiCG}}, \boldsymbol{r}^{(n)}_{\mathrm{BiCG}})}{(\tilde{\boldsymbol{p}}^{(n)}_{\mathrm{BiCG}}, A\boldsymbol{p}^{(n)}_{\mathrm{BiCG}})} = \frac{(\tilde{\boldsymbol{r}}^{(n)}_{\mathrm{BiCG}}, \boldsymbol{r}^{(n)}_{\mathrm{BiCG}})}{(\tilde{\boldsymbol{r}}^{(n)}_{\mathrm{BiCG}}, A\boldsymbol{p}^{(n)}_{\mathrm{BiCG}})} = \frac{(S_n(A^\top)\tilde{\boldsymbol{r}}^{(0)}, \boldsymbol{r}^{(n)}_{\mathrm{BiCG}})}{(S_n(A^\top)\tilde{\boldsymbol{r}}^{(0)}, A\boldsymbol{p}^{(n)}_{\mathrm{BiCG}})} \\ &= \frac{(\tilde{\boldsymbol{r}}^{(0)}, S_n(A)R_n(A)\boldsymbol{r}^{(0)})}{(\tilde{\boldsymbol{r}}^{(0)}, AS_n(A)P_n(A)\boldsymbol{r}^{(0)})} = \frac{(\tilde{\boldsymbol{r}}^{(0)}, \boldsymbol{r}^{(n)})}{(\tilde{\boldsymbol{r}}^{(0)}, A\boldsymbol{p}^{(n)})}\end{aligned} \tag{3.1.28}$$

$$\begin{aligned}\beta_n &= \frac{(\tilde{\boldsymbol{r}}^{(n+1)}_{\mathrm{BiCG}}, \boldsymbol{r}^{(n+1)}_{\mathrm{BiCG}})}{(\tilde{\boldsymbol{r}}^{(n)}_{\mathrm{BiCG}}, \boldsymbol{r}^{(n)}_{\mathrm{BiCG}})} = \frac{(S_{n+1}(A^\top)\tilde{\boldsymbol{r}}^{(0)}, \boldsymbol{r}^{(n+1)}_{\mathrm{BiCG}})}{(S_n(A^\top)\tilde{\boldsymbol{r}}^{(0)}, \boldsymbol{r}^{(n)}_{\mathrm{BiCG}})}\frac{\alpha_n}{\omega_n} \\ &= \frac{(\tilde{\boldsymbol{r}}^{(0)}, S_{n+1}(A)R_{n+1}(A)\boldsymbol{r}^{(0)})}{(\tilde{\boldsymbol{r}}^{(0)}, S_n(A)R_n(A)\boldsymbol{r}^{(0)})}\frac{\alpha_n}{\omega_n} = \frac{(\tilde{\boldsymbol{r}}^{(0)}, \boldsymbol{r}^{(n+1)})}{(\tilde{\boldsymbol{r}}^{(0)}, \boldsymbol{r}^{(n)})}\frac{\alpha_n}{\omega_n}\end{aligned} \tag{3.1.29}$$

using $S_n(A)$ instead of updating $\tilde{\boldsymbol{r}}^{(n)}$ associated with $A^\top$. The residual vector is updated as

$$\begin{aligned}\boldsymbol{r}^{(n+1)} &= S_{n+1}(A)R_{n+1}(A)\boldsymbol{r}^{(0)} = (1-\omega_n A)S_n(A)(R_n(A) - \alpha_n AP_n(A))\boldsymbol{r}^{(0)} \\ &= (1-\omega_n A)(\boldsymbol{r}^{(n)} - \alpha_n A p_n).\end{aligned} \tag{3.1.30}$$

ω_n is chosen to minimize the residual norm $\|\boldsymbol{r}^{(n+1)}\|_2$. When the auxiliary vector

$$\boldsymbol{s}^{(n+1)} = \boldsymbol{r}^{(n)} - \alpha_n A\boldsymbol{p}^{(n)} \tag{3.1.31}$$

is introduced, the residual vector becomes

$$\boldsymbol{r}^{(n+1)} = \boldsymbol{s}^{(n+1)} - \omega_n A\boldsymbol{s}^{(n+1)} \tag{3.1.32}$$

so the value of ω_n is given as

$$\omega_n = \frac{(A\boldsymbol{s}^{(n+1)}, \boldsymbol{s}^{(n+1)})}{(A\boldsymbol{s}^{(n+1)}, A\boldsymbol{s}^{(n+1)})}. \tag{3.1.33}$$

Using $\boldsymbol{s}^{(n+1)}$, $\boldsymbol{x}$ and $\boldsymbol{p}$ are updated as

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + \alpha_n\boldsymbol{p}^{(n)} + \omega_n(\boldsymbol{r}^{(n)} - \alpha_n A\boldsymbol{p}^{(n)}) = \boldsymbol{x}^{(n)} + \alpha_n\boldsymbol{p}^{(n)} + \alpha_n\boldsymbol{s}^{(n+1)} \tag{3.1.34}$$

$$\begin{aligned}\boldsymbol{p}^{(n+1)} &= S_{n+1}(A)P_{n+1}(A)\boldsymbol{r}^{(0)} = S_{n+1}(A)(R_{n+1}(A) + \beta_n P_n(A))\boldsymbol{r}^{(0)} \\ &= S_{n+1}(A)R_{n+1}(A)R_{n+1}(A) + \beta_n(1-\omega_n) + S_n(A)P_n(A)\boldsymbol{r}^{(0)} \\ &= r_{n+1} + \beta_n(\boldsymbol{p}^{(n)} - \omega_n A\boldsymbol{p}^{(n)}).\end{aligned} \tag{3.1.35}$$

By defining $K\hat{\boldsymbol{p}} = \boldsymbol{p}^{(n)}$ and $K\hat{\boldsymbol{s}} = \boldsymbol{s}^{(n)}$ for preprocessing, and $\boldsymbol{v}^{(n)} = A\hat{\boldsymbol{p}}$ and $\boldsymbol{t}^{(n)} = A\hat{\boldsymbol{s}}$ as the vectors that hold the result of the matrix vector product, and setting $\rho_n = (\tilde{\boldsymbol{r}}^{(0)}, \boldsymbol{r}^{(n)})$, the algorithm of the BiCGSTAB method with preprocessing, as shown in Algorithm 5, is obtained.

3.2 Incomplete Factorization Preconditioners

The convergence rate of the iterative method depends on the spectral characteristics of the coefficient matrix. Preconditioners are used to transform simultaneous linear equations into a problem with better spectral characteristics and the same solution. For example, if the

Algorithm 5 BiCGSTAB with preconditioner K

Compute $\boldsymbol{r}^{(0)} = \boldsymbol{b} - A\boldsymbol{x}^{(0)}$ for some initial guess $\boldsymbol{x}^{(0)}$
$\tilde{\boldsymbol{r}} = \boldsymbol{r}^{(0)}$
for $n = 1, 2, \ldots,$ until convergence **do**
  $\rho_{n-1} = (\tilde{\boldsymbol{r}}, \boldsymbol{r}^{(n-1)})$
  if $\rho_{n-1} = 0$ **then**
    method fails
  end if
  if $n = 1$ **then**
    $\boldsymbol{p}^{(n)} = \boldsymbol{r}^{(n-1)}$
  else
    $\beta_{n-1} = (\rho_{n-1}/\rho_{n-2})(\alpha_{n-1}/\omega_{n-1})$
    $\boldsymbol{p}^{(n)} = \boldsymbol{r}^{(n-1)} + \beta_{n-1}(\boldsymbol{p}^{(n-1)} - \omega_{n-1}\boldsymbol{v}^{(n-1)})$
  end if
  solve $K\hat{\boldsymbol{p}} = \boldsymbol{p}^{(n)}$
  $\boldsymbol{v}^{(n)} = A\hat{\boldsymbol{p}}$
  $\alpha_n = \rho_{n-1}/(\tilde{\boldsymbol{r}}, \boldsymbol{v}^{(n)})$
  $\boldsymbol{s}^{(n)} = \boldsymbol{r}^{(n-1)} - \alpha_n \boldsymbol{v}^{(n)}$
  if norm of $\boldsymbol{s}^{(n)}$ is small enough **then**
    $\boldsymbol{x}^{(n)} = \boldsymbol{x}^{(n-1)} + \alpha_n \hat{\boldsymbol{p}}$ and stop
  end if
  solve $K\hat{\boldsymbol{s}} = \boldsymbol{s}^{(n)}$
  $\boldsymbol{t}^{(n)} = A\hat{\boldsymbol{s}}$
  $\omega_n = (\boldsymbol{t}^{(n)}, \boldsymbol{s}^{(n)})/(\boldsymbol{t}^{(n)}, \boldsymbol{t}^{(n)})$
  $\boldsymbol{x}^{(n)} = \boldsymbol{x}^{(n-1)} + \alpha_n \hat{\boldsymbol{p}} + \omega_n \hat{\boldsymbol{s}}$
  $\boldsymbol{r}^{(n)} = \boldsymbol{s}^{(n)} - \omega_i \boldsymbol{t}^{(n)}$
  if norm of $\boldsymbol{r}^{(n)}$ is small enough **or** $\omega_n = 0$ **then**
    stop
  end if
end for

preconditioner K approximates the coefficient matrix A in any way, the equation $A\boldsymbol{x} = \boldsymbol{b}$ can be transformed into $K^{-1}A\boldsymbol{x} = K^{-1}\boldsymbol{b}$, with no change in the solution. Furthermore, compared with A, the coefficient matrix $K^{-1}A$ is expected to become closer to the identity matrix having the condition number 1. With this, it can be expected that the convergence of the Krylov subspace method will become faster.

A more accurate way to introduce the preconditioner is to divide the preconditioner into $K = K_1K_2$ and transform the original equation as $K_1^{-1}AK_2^{-1}(K_2\boldsymbol{x}) = K_1^{-1}\boldsymbol{b}$. Preconditioning with $K_1 = I$ is called right preconditioning, and the preconditioning with $K_2 = I$ is called left preconditioning. Algorithms 3 – 5 use the right preconditioning.

The conditions that need to be satisfied by the preconditioner K are :

- $K^{-1}A$ is as close to the identity matrix as possible.
- Generation of K is easy.
- Calculation of $K^{-1}\boldsymbol{x}$ is easy.

To satisfy these conditions, an incomplete factorization preconditioner is used.

Most of the preconditioners are given in a factorized form such as $K = LU$, where L is the lower triangular matrix and U is the upper triangular matrix. If the factorization process ignores the update of the fill element, that is, an element that was originally zero but becomes nonzero in the exact factorization, it is called an incomplete factorization. At this time, the effect of preprocessing is affected by the amount of K approximating A. If the original matrix is an M-matrix, it is guaranteed that incomplete factorization exists for many factorization strategies.

By performing $A = LU$ factorization, $A\boldsymbol{x} = \boldsymbol{b}$ is replaced by $LU\boldsymbol{x} = \boldsymbol{b}$, and the solution $\boldsymbol{x}$ is obtained by solving $L\boldsymbol{y} = \boldsymbol{b}$ and $U\boldsymbol{x} = \boldsymbol{y}$. The forward/ backward substitution method used to solve these triangular systems is called a triangular solver. In the 'solve' in the iterative calculation shown in Algorithms 3 – 5, which repeatedly finds $\boldsymbol{x}$ from different $\boldsymbol{b}$'s, L and U once found can be used repeatedly.

The algorithms for the LU factorization of matrix $A \in \mathbb{R}^{n\times n}$ include the outer-product form Gaussian method, the inner-product form Gaussian method, and the Crout method, all of which have different data reference regions and update regions. In this section, we first explain the LU factorization procedure based on the inner-product form Gaussian method, which is often used for sparse matrices.

The k-th stage operation of the factorization $a_{ij}^{(k+1)} = a_{ij}^{(k)} - (a_{ik}^{(k)}/a_{kk}^{(k)})a_{kj}^{(k)}$ uses the matrix

$$L_k = \begin{bmatrix} 1 & & & & & & & & \\ 0 & 1 & & & & & & & \\ \vdots & 0 & \ddots & & & & & & \\ & \vdots & \ddots & 1 & & & & & \\ & & & 0 & 1 & & & & \\ & & & \vdots & -l_{k+1,k} & 1 & & & \\ & & & & -l_{k+2,k} & 0 & \ddots & & \\ \vdots & \vdots & & \vdots & \vdots & \vdots & \ddots & 1 & \\ 0 & 0 & \cdots & 0 & -l_{n,k} & 0 & \cdots & 0 & 1 \end{bmatrix} \tag{3.2.36}$$

with $l_{ik} = a_{ik}^{(k)}/a_{kk}^{(k)}$, and can be written as $A^{(k+1)} = L_k A^{(k)}$, where $A^{(k)}$ is the coefficient matrix immediately before starting the elimination of the k-th stage. When this is repeated up to n stages, $A^{(n)}$ becomes the upper triangular matrix U. Thus, if the above procedure is repeatedly applied starting from $A^{(1)} = A$, $A^{(n)}$ becomes

$$U = A^{(n)} = L_{n-1}A^{(n-1)} = L_{n-1}L_{n-2}A^{(n-2)} = \cdots = L_{n-1}L_{n-2}\cdots L_1 A, \tag{3.2.37}$$

where the element u_{ij} of U is $a_{ij}^{(n)}$. In addition, because $L = L_{n-1}^{-1}L_{n-2}^{-1}\cdots L_1^{-1}$, $A = LU$ holds. Because L_k^{-1} is a matrix obtained by reversing the sign of the l_{ik} element of L_k,

$$L = \begin{bmatrix}
1 & & & & & & & & \\
l_{21} & 1 & & & & & & & \\
\vdots & l_{32} & \ddots & & & & & & \\
 & \vdots & \ddots & 1 & & & & & \\
 & & & l_{k,k-1} & 1 & & & & \\
 & & & \vdots & l_{k+1,k} & 1 & & & \\
 & & & & l_{k+2,k} & l_{k+2,k+1} & \ddots & & \\
\vdots & \vdots & & \vdots & \vdots & \vdots & \ddots & 1 & \\
l_{n1} & l_{n2} & \cdots & l_{n,k-1} & -l_{n,k} & l_{n,k+1} & \cdots & l_{n,n-1} & 1
\end{bmatrix}. \tag{3.2.38}$$

At this time, even if $a_{ij}^{(0)}$ of the original matrix A is at the zero position, the element l_{ij} or u_{ij} at the same position of L or U may not be zero.

In the substitution calculation to solve $LU\boldsymbol{x} = \boldsymbol{b}$, $L\boldsymbol{y} = \boldsymbol{b}$ is solved by forward substitution

$$y_i = \begin{cases} b_i & i = 1 \\ b_i - \sum_{j=1}^{i-1} l_{ij}y_j & i = 2, 3, \cdots, n \end{cases} \tag{3.2.39}$$

in order from $i = 1$ to n, and $U\boldsymbol{x} = \boldsymbol{y}$ is solved by backward substitution

$$x_i = \begin{cases} \frac{1}{u_{ii}}y_i & i = n \\ \frac{1}{u_{ii}}(y_i - \sum_{j=i+1}^{n} u_{ij}y_j) & i = n-1, n-2, \cdots, 1 \end{cases} \tag{3.2.40}$$

in order from $i = n$ to 1. The elements of $\boldsymbol{y}$ and $\boldsymbol{x}$ that appear on the right side must be updated, which can prevent the parallelization of substitution calculations. The method that can be used to avoid this and parallelize it will be described in Section 3.3.2.

If the original coefficient matrix is symmetric positive definite, the LU factorization becomes the Cholesky factorization. For the LU/Cholesky factorizations, we can consider an approximate factorization that discards all the fill-ins occurring at positions of zero elements in the original matrix. This is called the incomplete LU/Cholesky factorization with level(0) (ILU(0)/IC(0)). Algorithm 6 shows the ILU(0) factorization algorithm.

The order of the condition number of the ILU(0)/IC(0) preprocessed coefficient matrix is known to be $O(h^{-2})$, where h is the mesh size. This is the same as in the case of a non-preconditioned coefficient matrix, and an isolated minimum eigenvalue appears in the eigenvalue spectrum [41, 42].

Algorithm 6 ILU(0) factorization of an matrix $A \in \mathbb{R}^{n \times n}$

```
1: for i = 1 to n do
2:   for k = 1 to i − 1 if a_ik ≠ 0 do
3:     a_ik = a_ik / a_kk
4:     for j = k + 1 to n if a_kj ≠ 0 do
5:       if a_ij ≠ 0 then
6:         a_ij = a_ij − a_ik a_kj
7:       end if
8:     end for
9:   end for
10: end for
```

3.2.1 Modified Incomplete (MILU/MIC) Factorization

A modified incomplete factorization [43, 3, 5] can be used to improve the convergence acceleration effect in incomplete factorization. It accomplishes this by reducing the effect of fill-in drops during the factorization. In the ILU(0)/IC(0) factorization, the fill-in element at the zero-value position of the original matrix is discarded during the row update operation. In the MILU(0)/MIC(0) factorization, the sum of the values of the discarded elements is added to the diagonal elements of the row. With this compensation, the preconditioner has the same row sum as the original matrix; thus reducing the effect of rejection.

Algorithm 7 shows the MILU(0) factorization algorithm. It differs from the ILU(0) factorization algorithm owing to the addition of line 7, which holds the rejected fill-in value, and line 13, which compensates for the discarded elements. The preconditioner K is obtained from Algorithm 7, as follows: Let $\hat{A}$ be the matrix generated by Algorithm 7, and let $\hat{D}$, $\hat{L}$, and $\hat{U}$ be the diagonal, exact lower, and exact upper parts of the matrix $\hat{A}$, respectively. It then becomes $\hat{A} = \hat{L} + \hat{D} + \hat{U}$. At this time, K is factorized as $K = (\hat{L} + I)(\hat{D} + \hat{U})$, where I is the identity matrix. Moreover, when the original matrix A is symmetric, K becomes $(\hat{L} + I)\hat{D}(I + \hat{L}^\top)$.

Algorithm 7 MILU(0) factorization of matrix $A \in \mathbb{R}^{n \times n}$

```
1: for i = 1 to n do
2:   s = 0.0
3:   for k = 1 to i − 1 if a_ik ≠ 0 do
4:     a_ik = a_ik / a_kk
5:     for j = k + 1 to n if a_kj ≠ 0 do
6:       if a_ij = 0 then
7:         s = s + a_ik a_kj
8:       else
9:         a_ij = a_ij − a_ik a_kj
10:      end if
11:    end for
12:  end for
13:  a_ii = a_ii − s
14: end for
```

In the modified incomplete factorization, the diagonal element becomes close to zero due to

a compensation to the diagonal element, and there is a risk of breakdown due to division by the zero pivot. This becomes particularly problematic when ordering is applied for parallelization.

In the eigenvalue spectrum of a level-zero incompletely factorized matrix, isolated eigenvalues appear on the maximum side, and the minimum eigenvalue becomes 1 [41, 42]. One reason to consider modified incomplete factorization is the behavior of the spectral condition number of the preprocessed system, which is discussed in association with perturbation and relaxation.

3.2.2 Perturbed Modified Incomplete (PMILU/PMIC) Factorization

Variations of modified incomplete factorization with perturbation added to diagonal elements, are often treated as being indistinguishable from MILU/MIC, but they are distinguished here as perturbed modified incomplete factorization (PMILU/PMIC). In the PMILU/PMIC factorization, the diagonal elements of the coefficient matrix, before factorization are multiplied by $1+\zeta h^2$ with mesh size h and perturbation coefficient ζ. In other words, the MILU/PMIC factorization is applied to the $A+E$ matrix and not on the original matrix A, where $E = \zeta h^2 \mathrm{diag}(A)$.

Algorithm 8 shows the PMILU(0) factorization algorithm. A line perturbing the diagonal element before, factorization is inserted between lines 2 and 3 of the Algorithm 7.

Algorithm 8 PMILU(0) factorization of matrix $A \in \mathbb{R}^{n \times n}$

```
1:  for i = 1 to n do
2:    s = 0.0
3:    a_ii = a_ii(1 + ζh^2)
4:    for k = 1 to i − 1 if a_ik ≠ 0 do
5:      a_ik = a_ik/a_kk
6:      for j = k + 1 to n if a_kj ≠ 0 do
7:        if a_ij = 0 then
8:          s = s + a_ik a_kj
9:        else
10:         a_ij = a_ij − a_ik a_kj
11:       end if
12:     end for
13:   end for
14:   a_ii = a_ii − s
15: end for
```

It has been confirmed that the condition number of PMILU/PMIC preconditioned matrix is reduced from $O(h^{-2})$ to $O(h^{-1})$ in some problems [3, 4]. When $\zeta = 0$, the algorithm becomes the MILU(0)/MIC(0) method. Theoretical analysis suggests that a non-zero perturbation of A is required to achieve this order, but numerical experiments have pointed out that the condition number of $O(h^{-1})$ may be obtained even in the absence of perturbations [3]. In the eigenvalue spectrum of an incompletely factorized matrix with level 0, isolated eigenvalues appear on both sides of the minimum and maximum [42]. It is necessary to introduce perturbations that are small enough to bring the minimum eigenvalue closer to 1 and large enough to control the maximum eigenvalue. An estimation method [44] for the maximum and minimum eigenvalues including the internal distribution, was developed. The introduction of perturbations not only improves convergence, but also reduces the risk of zero pivot in MILU(0)/MIC(0) by increasing

the original diagonal elements. An explanation of the effect on the zero pivot is provided in Section 4.3.1.

3.2.3 Relaxed Modified Incomplete (RMILU/RMIC) Factorization

Relaxed modified incomplete factorization [23], also called the relaxed ILU/IC, relaxes the amount of compensation for fill-in, and reduces the effect on the superiority of diagonal elements. In the RMILU/RMIC factorization, the sum of the rejected fill-in is multiplied by a relaxation parameter $0 \leq \alpha \leq 1$, and then added to the diagonal elements.

Algorithm 9 shows the RMILU(0) factorization algorithm. It differs from the MILU(0) factorization algorithm in that the relaxation coefficient is multiplied by the compensation value in line 13.

Algorithm 9 RMILU(0) factorization of an matrix $A \in \mathbb{R}^{n \times n}$

```
1:  for i = 1 to n do
2:     s = 0.0
3:     for k = 1 to i − 1 if a_ik ≠ 0 do
4:        a_ik = a_ik/a_kk
5:        for j = k + 1 to n if a_kj ≠ 0 do
6:           if a_ij = 0 then
7:              s = s + a_ik a_kj
8:           else
9:              a_ij = a_ij − a_ik a_kj
10:          end if
11:       end for
12:    end for
13:    a_ii = a_ii − αs
14: end for
```

When $\alpha = 0$, the algorithm reduces to the ILU(0)/IC(0) method, and when $\alpha = 1$, the algorithm becomes the MILU(0)/MIC(0) method. An optimal value of α that depends on the mesh size h has been proposed [45, 42], whereas $\alpha = 0.95$ has also been proposed as an empirical recommended value [46].

As an improvement in the eigenvalue spectrum by relaxation, isolated eigenvalues appear on both sides of the minimum and maximum in the eigenvalue spectrum of the incompletely factorized matrix with level 0 [41, 42]. When a small number of eigenvalues are isolated near the maximum and minimum, fewer iterations are estimated compared with a uniform distribution [37]. Analysis of the eigenvalue distribution shows the equivalence between the perturbation and relaxation methods used to calculate the incomplete factorization [44]. The introduction of relaxation not only improves convergence, but also reduces the risk of zero pivot in MILU(0)/MIC(0) by reducing the fill-in subtracted from the diagonal elements. An explanation of the effect on the zero pivot is provided in Section 4.3.2.

3.3 Parallelization

In recent years, parallel computing has become widespread as a means of performing large-scale calculations. Therefore, adapting numerical calculation methods to multicore and manycore devices has become an important challenge. Several attempts have been made to implement iterative methods for large-scale simultaneous equations on various parallel computers. To achieve parallelization, alternative algorithms are being developed that extract as much parallelism as possible from standard algorithms and enhance parallelism by avoiding sequential processes. In addition, it is necessary to devise an implementation that matches the characteristics of the arithmetic unit and data communication in the machine that executes the calculation.

3.3.1 Parallel Architectures and Implementation Frameworks

Computer configurations are advancing from single-core to multicore, manycore processors, and devices to increase parallelism. The following parallelizations are widely used to increase the degree of parallelism of computers:

(a) Within a single compute node with multiple cores.
(b) Using accelerator devices.
(c) Among compute nodes with multiple nodes connected to each other.

Accelerator devices have long been used to accelerate computations in many scientific applications, and acceleration using GPUs is now widespread. Parallelization includes thread parallelization and process parallelization.

In thread parallelization, threads parallelize data on a common memory, which corresponds to (a) and (b). The process is started as a sequential operation, and when the computation reaches a loop with a large amount of calculation, parallel execution is performed by the threads. When this is completed, the operation returns to the sequential operation. Thread parallelization can be implemented by directives, and typical APIs that use directives include OpenMP and OpenACC [47]. The OpenACC is specialized for GPUs. The directives are inserted into the program code as a comment statement and specify the methods of parallelization, among others, to the compiler. If the compiler does not accept OpenACC directives, they remain as comment statements. In this way, directive-based parallelization can be implemented without leaving the original code, which has advantages for program maintainability and code portability. Declaring parallel sections based on directive syntax causes the compiler to properly generate and control threads. Parallelization is instructed by sandwiching the loop with the directives, and the parallelization method, private variables, reduction, etc., are specified as options. When using an accelerator device, directives are inserted to exchange data with the device.

Process parallelization corresponds to (c). As each process has its own data, parallel computation, wherein processes run on different nodes is possible. Multiple independent processes that execute the same program are generated, and operations are performed in parallel. Each process has a separate memory space, and the processes cooperate by exchanging messages. A library standard for message passing is the message passing interface (MPI) [48]. As the API is standardized, it can be executed on any machine where MPI is installed without changing the program. It can also be used for parallelization on a single node, but it also requires data distribution processing. System control and communication are performed by calling MPI functions.

Each process has an identification number called a rank, and the behavior for each rank can be defined.

Thread parallelism and process parallelism can be combined and implemented, and large-scale computations often use hybrid parallelization, where process parallelism is used to execute multiple thread-parallelized nodes and arithmetic accelerators [49, 50, 51, 20].

3.3.2 Parallelization of Preconditioning by Reordering

Owing to the limitations in parallelism, for example, during preconditioning, including sequential processes such as simple ILU factorization, methods to enhance parallelism have been developed. Major parallelization methods include partitioning the domain and computing each domain in parallel, and using graph theory algorithms to increase parallelism. Here, we describe a graph-theory-based parallelization method that involves the renumbering of problem variables.

In the forward/backward substitution calculation for incomplete factorization preprocessing, before calculating the row corresponding to a node, the row for the smaller/larger numbered nodes that are connected to the node must be calculated. If some nodes are numbered so that they are not adjacent to the nodes with smaller/larger numbers than themselves, they can be calculated without waiting for the other rows to be calculated. This means that the rows corresponding to a series of nodes starting from those nodes can be parallelized.

Such ordering methods include the multicolor method, the nested dissection method, and the Cuthill-Mckee method [11, 12]. The nodal multicolor method makes it possible to compute nodes with the same color in parallel, by coloring adjacent nodes with different colors and numbering the nodes with the same color in sequence [10]. If the nodes are colored by two colors, the nodal red-black (nodal RB) method is used. In the nested dissection method, nodes are divided into regions, and ordered from each region to its boundary [7]. If the region is divided in only one direction, it is a one-way dissection method [8]. In the Cuthill-Mckee method, the number of adjacent nodes is taken as the degree and leveled from the nodes with the lowest degree, after which the ordering is repeated in the order of the levels [6]. If we reorder it further in reverse, it is called the reverse Cuthill-Mckee method [9].

Generally, there is a trade-off between the degree of parallelism and the number of iterations for ordering strategies. Ordering affects the error matrix, causing clustering and clamping in the eigenvalue spectrum [52]. The deterioration of convergence may be improved by allowing a fill-in, but it has a negative impact on parallelism and data volume. The Frobenius norm of the remainder matrix $R = A - LU$ [53], and the graph information [54, 55, 15], are evaluated as indicators of the effect of ordering on convergence.

3.3.3 Block Red-Black (BRB) Ordering

In this study, to parallelize incomplete LU preprocessing, we use the block red-black (BRB) ordering [13], which partitions the nodes into blocks and applies red-black ordering to the blocks. When two or more colors are used for color coding, it is called block multicolor ordering, which has also been applied to irregular lattices to confirm its validity [56, 57].

The BRB method reduces the synchronization cost by reducing the number of colors compared to the block multicolor method while maintaining the convergence rate [14]. The number of blocks is twice that of parallel processes. The number of nodes in a block is, on average, the

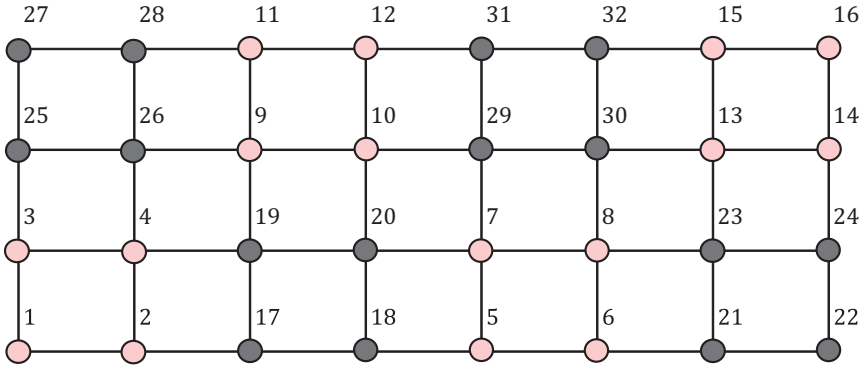

Figure 3.1: BRB coloring and ordering of a 8×4 grid.

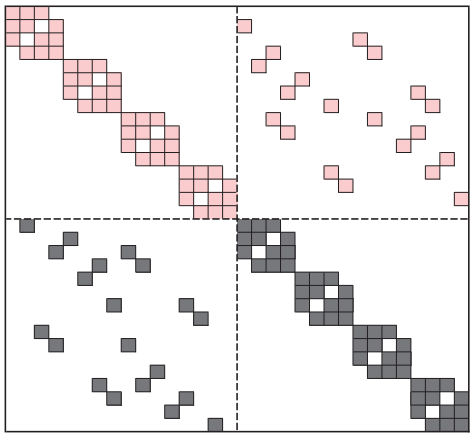

Figure 3.2: Sparse pattern of the matrix arising from BRB ordering in Fig. 3.1.

total number of nodes divided by the number of blocks. The tasks corresponding to the nodes contained in a block are processed by a single process or thread.

In BRB ordering, grid points are numbered using the following procedure:

Step 1. Divide the calculation region into blocks.

Step 2. Paint the blocks in red or black so that the adjacent blocks have different colors.

Step 3. Number the grid points in the red block. The numbering within each block is kept in the lexicographical order.

Step 4. After all the grid points in the red block have been numbered, the grid points in the black block are numbered in the same manner.

Fig. 3.1 shows an example of the BRB ordering of a two-dimensional 5-point finite difference grid. Here, the 4×2 blocks on the grid are constructed such that each block consists of a 2×2 subgrid of the entire 8×4 grid.

The nonzero pattern of the coefficient matrix arising from this new ordering is shown in Fig. 3.2. The diagonal blocks correspond to matrix elements within the same block, and the non-diagonal blocks, in the lower-left and upper-right, correspond to matrix elements between

two grid points with different colors. One grid point is adjacent to a grid point within the same block, and to grid points in a block with the other color. This eliminates dependency among blocks, with the same color and enables the forward and backward substitution process for each block with the same color to be performed in parallel. The forward substitution requires synchronization after all the red diagonal blocks have been processed, and the backward substitution requires synchronization after all the black diagonal blocks have been processed, each time.

It has been pointed out that the combined use of nodal RB ordering and MILU/MIC factorization creates a breakdown problem owing to the occurrence of zero pivot [17]. This is because the reduction of the diagonal elements due to the compensation of fill-in is of the same size as the diagonal elements themselves. Even in the case of BRB ordering, zero pivot occurs in the internal black blocks, where all the adjacent red blocks in the direction of decreasing coordinate values are internal blocks. The mechanism of the zero pivot and the detailed conditions for their occurrences are described in Section 4.2. As shown in Section 4.3, this problem can be avoided by introducing a perturbation or relaxation.

There is a trade-off between parallelism and convergence also in BRB ordering, but unlike nodal RB ordering, BRB ordering can adjust the block size to balance convergence and parallelism. In general, convergence indicators such as S.R.I. [15] and incompatible ratio [54] worsen, with an increasing number of blocks. However, for the combination of either the PMILU(0) or RMILU(0) factorization with BRB ordering, an increase in the number of blocks does not necessarily lead to worse convergence, as long as each block contains more than several tens of nodes. Figs. 4.4, 4.5 and 5.12 also show a near-flat trend in this range. This suggests the existence of a condition that allows for both high parallelism and fast convergence. The relationship between block partitioning and convergence rate is discussed in Section 5.3.4.2.

Chapter 4

Numerical Stability of MILU(0) Preconditioning Based on Block Red-Black Ordering

4.1 Introduction

Since MILU/MIC preconditioner has a high convergence acceleration effect, it is a natural idea to parallelize them through reordering. However, it has been noted that the combination of some reordering strategies and MILU/MIC preconditioning can lead to a factorization with zero or very small pivots. In particular, for matrices arising from the discretization of second-order linear differential operators on an equally spaced grid, it was proved that nodal red-black (nodal RB) ordering or multiple wave-front type ordering combined with MILU/MIC factorization results in zero pivot or $O(h)$ pivot, respectively [17]. Perturbation [4] or relaxation [41] is reported to alleviate the problem.

In this chapter we consider this danger under the following assumptions.

Assumption 1. *The matrix A is derived from a finite difference discretization of the second-order linear partial differential equation (PDE) on an equally spaced grid using the two-dimensional five-point or three-dimensional seven-point stencil. Thus, A has a symmetric sparse pattern.*

Assumption 2. *If i is an interior node of the grid, then the i-th row of A has zero row sum. In other words, the PDE does not have zeroth-order terms, and has the form:*

$$c_1\frac{\partial^2 u}{\partial x^2} + c_2\frac{\partial^2 u}{\partial x \partial y} + c_3\frac{\partial^2 u}{\partial y^2} + c_4\frac{\partial u}{\partial x} + c_5\frac{\partial u}{\partial y} = f, \tag{4.1.1}$$

where c_1, ..., c_5 and f are in general functions of x and y.

As useful tools for analyzing the properties of the preconditioning matrices generated by the MILU(0)/MIC(0) factorization, we define *influence range* [58, 17] introduced by Eijkhout and of *depth* used for the discussion of the influence range.

Definition 1. [58, 17] *For a matrix A that has a symmetric sparse pattern, the influence range for node i is defined as the set:*

$$\begin{aligned}\mathcal{I}(i) = \{j : \exists i_0, \cdots, i_p \ (j = i_0 < \cdots < i_p = i) \\ \text{and } \forall q \in \{1, 2, \ldots, p\} \quad (i_q \text{ and } i_{q-1} \text{ are connected.})\}.\end{aligned} \tag{4.1.2}$$

Furthermore, we define depth $\mathcal{D}(i)$ as the maximum length of the chain $i_0 < i_1 < \cdots < i_p = i$ that appears in the definition of $\mathcal{I}(i)$ for node i.

Since i_q and i_{q-1} are connected, $a_{i_q i_{q-1}} \neq 0$ and the i_{q-1}-th row is used to update the i_q-th row. Thus, $\mathcal{I}(i)$ is considered as the set of rows that directly or indirectly affect the i-th row during factorization. From the definition, it is easy to see that if $k \in \mathcal{I}(i)$, then $\mathcal{I}(k) \subseteq \mathcal{I}(i)$. If the i-th row is not updated from any row, then $\mathcal{I}(i) = \emptyset$ and only then $\mathcal{D}(i) = 0$. The concepts of the influence range and depth are useful for analyzing the generation and mitigation of zero pivot.

In this chapter we analyze the combination of block red-black (BRB) ordering and MILU/MIC factorization under these assumptions and definitions. Then, we show the dangers when applied to matrices resulting from the discretization of difference operators. Especially, we show that under certain conditions regarding the number of block partitions, zero pivot occurs during factorization and no effective preconditioner can be obtained. We also show that this problem can be mitigated by perturbation or relaxation, and that the resulting combination of the perturbed or relaxed preconditioner and BRB ordering is more effective than the combination of ILU(0) preconditioner and BRB ordering, as shown by the numerical results.

The remainder of this chapter consists of the following. In Section 4.2, we show how zero pivoting occurs in certain combinations of ordering strategies and MILU factorization. On this basis, we derive necessary and sufficient conditions for the occurrence of zero pivot in BRB ordering. We describe perturbation and relaxation methods that can solve this problem in Sections 4.3.1 and 4.3.2, respectively. The experimental results of the resulting preconditioner performance are presented in Section 4.4. Finally, Section 4.5 gives some summary.

4.2 Occurrence of Zero Pivot

The combination of MIC/MILU factorization with ordering methods such as nodal RB ordering or ordering by diagonal leads to the danger of zero pivots. The following conditions that produce a zero pivot have been proven by Eijkhout. Here, the condition is expressed by a generalized row sum. For a fixed vector $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$ independent of i, the row sum of the i-th row in A is defined as $\sum_{j=1}^{n} c_j a_{ij}$. Using generalized row sums for the modified incomplete factorization, line 7 of Algorithm 7 is rewritten as

$$s = s + a_{ik} a_{kj} (c_j / c_i). \tag{4.2.3}$$

The generalized row sum of row i after being updated by row k is always equal to the generalized row sum obtained without dropping any fill-in [17]. A generalized row sum is a row sum when $\boldsymbol{c} = (1, 1, \ldots, 1)$.

Theorem 1. (Lemma 4.1 of Eijkhout [17]) *For a symmetric sparse pattern matrix A, suppose that the following conditions hold for some i.*

(i) *Matrix has zero generalized row sum at row i,*
(ii) *Generalized row sum is zero for all nodes in the influence range of i, and*
(iii) *i is not included in the influence range of any other node.*

Then, the unperturbed modified incomplete decomposition either makes u_{ii} a zero pivot or generates a zero pivot in the influence range of i. When the matrix is an M-matrix, the latter is not possible to occur, and the condition is then a necessary condition to produce a zero pivot.

Here we give a proof of Theorem 1 using the notation we introduced in Definition 1.

Proof. If a zero pivot occurs in any row contained in $\mathcal{I}(i)$, the theorem is trivial. Therefore, in the following we assume that zero pivot does not occur in $\mathcal{I}(i)$. In that case, the rows in $\mathcal{I}(i)$ will be deleted first, since i is only updated by the rows in $\mathcal{I}(i)$. By changing the order of elimination, we can avoid breakdown in the elimination of rows $i' < i$ that do not belong to $\mathcal{I}(i)$, and the elimination of row i can be done without breakdown.

We prove that the generalized row sum of any row $k \in \mathcal{I}(i)$ becomes zero after the elimination is done. We use mathematical induction on $\mathcal{D}(k)$. First, consider the case where $\mathcal{D}(k) = 0$. In this case, $\mathcal{I}(k) = \emptyset$, so row k is not updated by any row. Therefore, its generalized row sum is zero due to condition (ii). Next, we assume that the proposition is true for all rows in $\mathcal{I}(i)$ with depth $< d$, where d is a positive integer. Now, take the row $k \in \mathcal{I}(i)$ with $\mathcal{D}(k) = d$. Then any row k' used to update row k exists in $\mathcal{I}(k) \subseteq \mathcal{I}(i)$. Moreover, $\mathcal{D}(k') < d$ because $\mathcal{D}(k) \geq d + 1$ otherwise. By assumption, for row k', the generalized row sum after elimination equals zero. Hence, the generalized row sum of row k, which is zero by condition (ii), is unchanged by the update by row k'. Since this is true for any row k' updating row k, the proposition is also true for $d + 1$. By induction, for any row $k \in \mathcal{I}(i)$, the generalized row sum after elimination equals zero.

Since row i is updated only from rows in $\mathcal{I}(i)$, the generalized row sum of row i is invariant after elimination. Therefore, the generalized row sum of row i is zero from condition (i). However, there is no non-zero element on the left side of the diagonal element in row i after elimination. Furthermore, condition (iii) implies that there are no non-zero elements on the right-hand side of the diagonal. Therefore, for row i, the diagonal element after elimination needs to be zero. For a proof of the necessary condition in the M-matrix case, see [17]. □

From Theorem 1, we show that the BRB ordered coefficient matrix has a zero pivot when certain conditions are satisfied. The following theorem covers the simplest case where the domain is rectangular in two dimensions or rectangular parallelepiped in three dimensions.

Theorem 2. *Let A be a matrix satisfying the Assumptions 1 and 2. Assume further that the region is rectangular or rectangular parallelepiped and that A is ordered by BRB. Then, if any one of the conditions:*

(a) *The first block is black and the number of blocks in each direction is greater than 4, or*
(b) *The first block is red and the number of blocks in each direction is greater than 4 and greater than 5 in at least one direction,*

is satisfied, an unperturbed MILU(0)/MIC(0) applied to A will produce a zero pivot.

Proof. First, assume that (a) is satisfied. We consider the 2D case, which is rectangular. Let N_x be the number of blocks in the x direction and N_y the number of blocks in the y direction, and define the block index (i_B, j_B), where $1 \leq i_B \leq N_x$, $1 \leq j_B \leq N_y$ and $N_x, N_y \leq 4$. As an example, consider the BRB coloring shown in Fig. 4.1.

Now consider the node i in the upper right corner of the black block $(3, 3)$. The node i is drawn by the filled circle in Fig. 4.1. Since node i is a part of the inner block and has zero row sum from 1, the condition (i) of Theorem 1 is satisfied.

Next, consider the influence range $\mathcal{I}(i)$ of i. In Fig. 4.1, $\mathcal{I}(i)$ is enclosed by a dashed line. The blocks adjacent to the black block $(3, 3)$ are red blocks $(2, 3)$, $(3, 2)$, $(4, 3)$ and $(3, 4)$. A

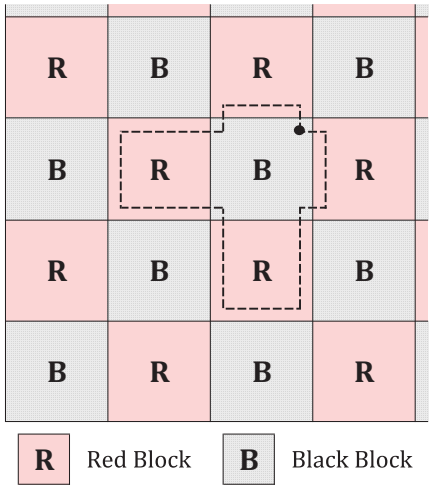

Figure 4.1: An example of BRB ordering. The node i (filled circle) where the zero pivot occurs and its influence range $\mathcal{I}(i)$ (dashed line).

node in a red block has a smaller number than a node in a black block, and is adjacent only to a node in a black block. Therefore, it is easy to show that $\mathcal{I}(i)$ is the set of nodes that belong to these red blocks and the block $(3,3)$ itself. Since the blocks $(2,3)$, $(3,2)$ and $(3,3)$ are inner blocks, all nodes which belong to them have zero row sum. By contrast, blocks $(4,3)$ and $(3,4)$ are boundary blocks. However, the nodes in the block $(4,3)$ belonging to $\mathcal{I}(i)$ are located only at the left end of the block, and the nodes in the block $(3,4)$ belonging to $\mathcal{I}(i)$ are located only at the bottom of the block. All of these nodes are internal nodes that maintain zero row sums. Therefore, the condition (ii) of Theorem 1 is satisfied.

Finally, i is not included in the influence range of any other node, because it has a number larger than any node it touches. Therefore, the condition (iii) of Theorem 1 is also satisfied.

The i-th pivot from Theorem 1 becomes zero pivot in MILU(0)/MIC(0) without perturbation of A. The same method can be used to prove the theorem for 3D in the case of a rectangular parallelepiped region.

Next, assume that (b) is satisfied. We consider the 2D case again. In this case, we can show that the conditions of Theorem 1 are satisfied in the same way as in (a), by focusing on the black block $(4,3)$ (when $N_x \geq 5$) or the black block $(3,4)$ (when $N_y \geq 5$). It is easy to extend this to the 3D case. □

Theorem 2 can be further generalized to irregular regions. That is, the theorem holds for the case where there is an interior black block such that all of the adjacent red blocks are themselves interior in the direction of decreasing x, y or z coordinates. In this case, the pivot corresponding to the largest node number i is zero in the modified incomplete factorization of A without perturbation.

4.3 Avoiding Zero Pivot

4.3.1 Mitigating the Problem by Introducing Perturbations

The zero pivot mentioned in Theorem 2 is caused by the fact that the total amount subtracted from the diagonal element by elimination and compensations for the fill-in is equal to of the diagonal element itself. It was reported in [17] that the introduction of perturbations to the diagonal elements solves the problem of pivots close to zero and improves the convergence behavior. The introduction of perturbation means that the diagonal elements of the coefficient matrix are multiplied by $1 + \zeta h^2$ prior to factorization, where h is the grid size and ζ is the perturbation coefficient. The algorithm for the perturbed MILU factorization with level 0 (PMILU(0)) is given in Algorithm 8. We apply the same approach to avoid zero pivot in BRB ordering as well.

As a preparation to prove the theorem concerning the effect of perturbation, we present the following lemma on updating of off-diagonal elements.

Lemma 1. *Let A be a matrix satisfying Assumption 1. Then, if the incomplete factorization with level 0, where all fill-ins are dropped, is applied to A, then the off-diagonal nonzero elements of A are not updated.*

Proof. This can be proved by contradiction. Assume that the elimination by the ℓ-th row updates the non-zero element a_{jk} ($j \neq k$). In this case, both $a_{j\ell}$ and $a_{\ell k}$ must be nonzero. The incomplete factorization with level 0 does not allow for fill-ins, which means that all of a_{jk}, $a_{j\ell}$, and $a_{\ell k}$ are nonzero elements in the original matrix. However, this is a condition for three nodes j, k, and ℓ where j is adjacent to k and ℓ, and k is adjacent to j and ℓ, which is impossible in a regular lattice. Therefore, the off-diagonal nonzero elements of A are kept at their original values during the factorization. □

The following theorem shows that perturbations introduced to the diagonal elements alleviate the problem of zero pivots, at least for certain types of matrices. This theorem is not restricted to matrices reordered by BRB ordering, but can also be applied when the domain is not rectangular or rectangular parallelepiped.

Theorem 3. *Let A be a matrix satisfying Assumption 1. Suppose that A is an M-matrix and is normalized to make the diagonal elements* 1*. Suppose further that row i satisfies conditions (i), (ii) and (iii) of Theorem 1 for the generalized row sum defined by a positive vector $\mathbf{c} = (c_1, c_2, \ldots, c_n)$. Then, if the PMILU(0)/MIC(0) factorization is applied to A, then the pivot satisfies $u_{kk} \geq \zeta h^2$ for all $k \in \mathcal{I}(i) \cup \{i\}$.*

Proof. Multiplying the diagonal element of A by $1 + \zeta h^2$ is the same as adding a perturbation of ζh^2 to the element after elimination. Let $\hat{a}_{kj}$ ($j \geq k$) be the element in the kth row before adding the perturbation after elimination, then $u_{kk} = \hat{a}_{kk} + \zeta h^2$ and $u_{kj} = \hat{a}_{kj}$ ($j > k$). Also note that from Lemma 1, the nonzero elements on the off-diagonal of A are not updated and remain negative during the factorization.

Here, we show by induction that for $k \in \mathcal{I}(i) \cup \{i\}$, $u_{kk} \geq \zeta h^2$ and $\sum_{j=1}^{n} c_j u_{kj} > 0$. First, consider the case where $\mathcal{D}(k) = 0$. In this case, since $\mathcal{I}(k) = \emptyset$, the k row is not updated from any of the rows. Therefore, from condition (ii) of Theorem 1,

$$\sum_{j=1}^{n} c_j \hat{a}_{kj} = \sum_{j=1}^{n} c_j a_{kj} = 0. \tag{4.3.4}$$

By introducing perturbations,

$$u_{kk} = \hat{a}_{kk} + \zeta h^2 = a_{kk} + \zeta h^2 \geq \zeta h^2. \tag{4.3.5}$$

It is also clear that

$$\sum_{j=1}^{n} c_j u_{kj} = \sum_{j=1}^{n} c_j a_{kj} + \zeta h^2 = \zeta h^2 > 0. \tag{4.3.6}$$

Next, assume that this claim holds for all rows of $\mathcal{I}(i)$ when depth is less than some positive integer d. Now, choose a row of $k \in \mathcal{I}(i) \cup \{i\}$ with $\mathcal{D}(k) = d$. It can be seen from conditions (i) and (ii) of Theorem 1 that the initial value of its generalized row sum is 0. Let k' be an arbitrary row used to eliminate k rows. Then, $k' \in \mathcal{I}(k) \subseteq \mathcal{I}(i)$ and $\mathcal{D}(k') < d$. Therefore, by hypothesis, $u_{k'k'} \geq \zeta h^2 > 0$. Since $a_{kk'} < 0$, the multiplier used in elimination $-a_{kk'}/u_{k'k'}$ is positive. From the combination of this fact and the hypothesis $\sum_{j=1}^{n} c_j u_{k'j} > 0$, the generalized row sum of the k-th row is positive after the elimination by the k'-th row. Therefore, after that of the k-th row is finished, $\sum_{j=1}^{n} c_j \hat{a}_{kj} > 0$ and

$$u_{kk} = \hat{a}_{kk} + \zeta h^2 > -\sum_{j \neq k} \frac{c_j}{c_k} \hat{a}_{kj} + \zeta h^2 = -\sum_{j \neq k} \frac{c_j}{c_k} a_{kj} + \zeta h^2 \geq \zeta h^2. \tag{4.3.7}$$

This completes the induction and proves the theorem. □

This theorem shows that with the introduction of perturbations, the pivot that becomes zero in the modified incomplete factorization without perturbations becomes at least ζh^2 away from zero.

4.3.2 Mitigating the Problem by Relaxing Compensation

One way to avoid the possibility of zero pivot other than perturbation is the relaxed modified incomplete factorization, which is used to improve the convergence of incomplete factorization preconditioner. The algorithm for the relaxed MILU factorization with level 0 (RMILU(0)) is shown in Algorithm 9. In the RMILU(0), the fill-in element generated during the factorization is multiplied by a relaxation parameter $0 \leq \alpha \leq 1$ before being subtracted from the diagonal element. This method is expected to alleviate the problem of zero pivot generation by making the compensation for fill-in smaller.

Before presenting the theorem, we introduce some notation.

Definition 2. *The set of column indices of the non-zero elements on the left side of the diagonal of the k-th row is defined as $\mathcal{L}(k)$. Then, $\mathcal{L}(k)$ is the set of indices of the row that updates k-th row. Similarly, the set of column indices of the nonzero elements on the right side of the diagonal is defined as $\mathcal{R}(k)$. Then, $\mathcal{R}(k)$ is the set of row indices that are updated by the k-th row from the symmetric sparse pattern of A. Also, define r_k as the row sum of the k-th row after elimination. Furthermore, we define $S_{\ell k}$ by*

$$s_{lk} = \frac{1}{u_{ll}} \left\{ r_l + (1-\alpha) \sum_{j \in \mathcal{R}(l) \setminus \{k\}} (-a_{lj}) \right\}. \tag{4.3.8}$$

The following theorem shows that relaxation of the compensation reduces the problem of zero pivots for at least certain types of matrices. Since the analysis in this case is more difficult than that for perturbations, we consider the restricted condition that the number of nonzero elements in each row in $\mathcal{I}(i) \cup \{i\}$ is the same and $\boldsymbol{c} = (1, 1, \ldots, 1)$.

Theorem 4. *Let A be an M-matrix satisfying Assumption 1. Now, suppose that conditions (i), (ii) and (iii) of Theorem 1 are satisfied for the row sum. We also assume that every row of $\mathcal{I}(i) \cup \{i\}$ has p nonzero elements on the off-diagonal, where p is a positive integer, $\mu = \min_{k \in \mathcal{I}(i) \cup \{i\}, \ell \neq k, a_{k\ell} \neq 0} |a_{k\ell}|$ and $\omega = \max_{k \in \mathcal{I}(i) \cup \{i\}, \ell \neq k} |a_{k\ell}|$. Assume further that μ and ω satisfy condition*

$$\omega \leq \min \left\{ \frac{p-2}{p-1} - (p-3)\mu - (1-\alpha),\ (p-1)\mu - (1-\alpha) \right\}, \tag{4.3.9}$$

where α $(0 \leq \alpha \leq 1)$ is the relaxation parameter. Then, applying the MILU(0)/MIC(0) factorization relaxed by the parameter α to A, the pivot satisfies $u_{kk} \geq (1-\alpha)(p-1)\mu^2$ for all $k \in \mathcal{I}(i) \cup \{i\}$.

Proof. As shown in Lemma 1, the off-diagonal nonzero elements of A are not updated by the factorization and are kept negative. Also, in the case of $k \in \mathcal{I}(i) \cup \{i\}$, from the zero row sum property, $\sum_{\ell \neq k}(-a_{k\ell}) = a_{kk} = 1$. Therefore, $0 < \mu \leq \omega \leq 1$.

There are no non-zero elements on the left side of the diagonal after elimination, and thus

$$r_k = u_{kk} + \sum_{l \in \mathcal{R}(k)} a_{k\ell} \tag{4.3.10}$$

If $k \in \mathcal{I}(i) \cup \{i\}$, then

$$\sum_{\ell \in \mathcal{L}(k)} a_{k\ell} + a_{kk} + \sum_{l \in \mathcal{R}(k)} a_{k\ell} = \sum_{\ell \in \mathcal{L}(k)} a_{k\ell} + 1 + \sum_{l \in \mathcal{R}(k)} a_{k\ell} = 0 \tag{4.3.11}$$

by the zero row sum property. Combining (4.3.10) and (4.3.11) yields

$$u_{kk} = a_{kk} + \sum_{l \in \mathcal{L}(k)} a_{k\ell} + r_k. \tag{4.3.12}$$

Assume that $k \in \mathcal{I}(i) \cup \{i\}$ and $\mathcal{L}(k) \neq \emptyset$. Consider the change that elimination by the row $\ell \in \mathcal{L}(k)$ makes to the row sum of the k-th row. This elimination sets $a_{k\ell}$ to zero and leaves the other off-diagonal elements intact. Thus, the increase in the contribution from the non-diagonal elements to the row sum is $-a_{k\ell}$. The diagonal element a_{kk} is increased by $-a_{k\ell}a_{\ell k}/u_{\ell\ell}$ by elimination. The total fill-in to be added to the diagonal is $-\alpha(a_{k\ell}/u_{\ell\ell}) \sum_{j \in \mathcal{R}(\ell) \setminus \{k\}} a_{\ell j}$. The change in the row sum by all these contributions combined is

$$\begin{aligned}
&-a_{k\ell} - \frac{a_{k\ell}a_{\ell k}}{u_{\ell\ell}} - \alpha \frac{a_{k\ell}}{u_{\ell\ell}} \sum_{j \in \mathcal{R}(\ell) \setminus \{k\}} a_{\ell j} \\
&= -\frac{a_{k\ell}}{u_{\ell\ell}} \left\{ u_{\ell\ell} + a_{\ell k} + \sum_{j \in \mathcal{R}(\ell) \setminus \{k\}} a_{\ell j} + (1-\alpha) \sum_{j \in \mathcal{R}(\ell) \setminus \{k\}} (-a_{\ell j}) \right\} \\
&= -\frac{a_{k\ell}}{u_{\ell\ell}} \left\{ r_\ell + (1-\alpha) \sum_{j \in \mathcal{R}(\ell) \setminus \{k\}} (-a_{\ell j}) \right\}
\end{aligned} \tag{4.3.13}$$

where we used Eq. (4.3.10) in the last equation. Then, the first row sum of k-th rows is zero, and thus the final row sum considering contributions from all $\ell \in \mathcal{L}(k)$ can be written as

$$r_k = \sum_{l \in \mathcal{L}(k)} (-a_{k\ell}) s_{\ell k} \tag{4.3.14}$$

using $s_{\ell k}$. Eqs. (4.3.8), (4.3.12) and (4.3.14) represent the propagation of the row sum change during the PMILU(0)/PMIC(0) factorization.

To prove the theorem, for any $k \in \mathcal{I}(i)$ and $m \in \mathcal{R}(k)$, the claim

$$s_{km} \geq (1-\alpha)(p-1)\mu \tag{4.3.15}$$

is shown by induction.

First, consider the case where $\mathcal{D}(k) = 0$. In this case, since $\mathcal{I}(k) = \emptyset$, the row k is not modified from any row. Therefore, from condition (ii) of Theorem 1, we have $u_{kk} = a_{kk} = 1$ and $r_k = 0$. In addition, because row k is unchanged from any row, there are no nonzero elements on the left side of the diagonal. That is, all of the p non-diagonal nonzero elements are on the right side of the diagonal. Therefore, from Eq. (4.3.8), s_{km} is bounded below as

$$s_{km} = (1-\alpha) \sum_{j \in \mathcal{R}(k) \setminus \{m\}} (-a_{kj}) \geq (1-\alpha)(p-1)\mu \tag{4.3.16}$$

and the claim (4.3.15) holds.

Next, we assume that Eq. (4.3.15) holds for all rows of $\mathcal{I}(i)$ with the depth less than a positive integer d.

Now, choose a row $k \in \mathcal{I}(i)$ with $\mathcal{D}(k) = d$. Then the row ℓ used to eliminate row k satisfies $\mathcal{D}(\ell) < d$, so s_{ellk} from the hypothesis. Therefore, the row sum of row k after elimination is evaluated as

$$r_k = \sum_{l \in \mathcal{L}(k)} (-a_{k\ell}) s_{\ell k} \geq (1-\alpha)(p-1)\mu \sum_{l \in \mathcal{L}(k)} (-a_{k\ell}). \tag{4.3.17}$$

Meanwhile, substituting Eq. (4.3.12) into Eq. (4.3.8), the definition of s_{km}, we get

$$s_{km} = \frac{(1-\alpha)\sum_{j \in \mathcal{R}(k) \setminus \{m\}} (-a_{kj}) + r_k}{a_{kk} + \sum_{l \in \mathcal{L}(k)} a_{kl} + r_k}. \tag{4.3.18}$$

We show that this right-hand side is bounded below. Let $|\mathcal{L}(k)| = q$ and $|\mathcal{R}(k)| = p - q$. Because row k is updated from at least one row, $q \geq 1$. Furthermore, in order to be consistent with the assumption $k \in \mathcal{I}(i)$, it must hold that $q \leq p-1$, that is, $\mathcal{R}(k) \neq \emptyset$.

Note that the first term in the numerator of (4.3.18) is the sum of the first and second terms in the denominator, which can be bounded as in

$$(1-\alpha) \sum_{j \in \mathcal{R}(k) \setminus \{m\}} (-a_{kj}) \leq \sum_{j \in \mathcal{R}(k)} (-a_{k\ell}) = a_{kk} + \sum_{j \in \mathcal{L}(k)} a_{k\ell}, \tag{4.3.19}$$

where we are using the zero row sum property for the last equality.

Next, consider the general inequality

$$\frac{b+\epsilon}{a+\epsilon} \geq \frac{b+\delta}{a+\delta}, \tag{4.3.20}$$

where the four positive numbers a, b, ϵ and δ are $a \geq b$ and $\epsilon \geq \delta$. By applying this to Eq. (4.3.18) with a as the right-hand side of Eq. (4.3.19), b as the left-hand side of Eq. (4.3.19), $\epsilon = r_k$, and δ as the right-hand side of Eq. (4.3.17),

$$s_{km} \geq \frac{(1-\alpha)\sum_{j\in\mathcal{R}(k)\setminus\{m\}}(-a_{kj}) + (1-\alpha)(p-1)\mu\sum_{l\in\mathcal{L}(k)}(-a_{k\ell})}{a_{kk} + \sum_{l\in\mathcal{L}(k)} a_{k\ell} + (1-\alpha)(p-1)\mu\sum_{l\in\mathcal{L}(k)}(-a_{k\ell})} \tag{4.3.21}$$

is obtained. The numerator is rewritten as

$$\begin{aligned}
&(1-\alpha)(p-1)\mu\Bigg[\sum_{j\in\mathcal{L}(k)}(-a_{kj}) + \sum_{j\in\mathcal{R}(k)\setminus\{m\}}(-a_{kj}) \\
&\qquad\qquad + \left\{\frac{1}{(p-1)\mu} - 1\right\}\sum_{j\in\mathcal{R}(k)\setminus\{m\}}(-a_{kj})\Bigg] \\
&= (1-\alpha)(p-1)\mu\left[1 + a_{km} + \left\{\frac{1}{(p-1)\mu} - 1\right\}\sum_{j\in\mathcal{R}(k)\setminus\{m\}}(-a_{kj})\right] \\
&\geq (1-\alpha)(p-1)\mu\left[1 - \omega + \left\{\frac{1}{(p-1)\mu} - 1\right\}(p-q-1)\mu\right]
\end{aligned} \tag{4.3.22}$$

using the zero row sum property for the first equation. The denominator is expressed as

$$a_{kk} + \sum_{l\in\mathcal{L}(k)} a_{kl} + (1-\alpha)(p-1)\mu\sum_{l\in\mathcal{L}(k)}(-a_{kl}) \leq 1 - q\mu + (1-\alpha), \tag{4.3.23}$$

where $(p-1)\mu \leq 1$ and $\sum_{l\in\mathcal{L}(k)}(-a_{kl}) \leq 1$ were used. Substituting (4.3.22) and (4.3.23) into (4.3.21), we get

$$s_{km} \geq (1-\alpha)(p-1)\mu \cdot \frac{1 - \omega + \left\{\frac{1}{(p-1)\mu} - 1\right\}(p-q-1)\mu}{1 - q\mu + (1-\alpha)}. \tag{4.3.24}$$

Since the denominator of the right-hand side is positive, we can see that the claim (4.3.15) holds if

$$-\omega + \left\{\frac{1}{(p-1)\mu} - 1\right\}(p-q-1)\mu \geq -q\mu + (1-\alpha) \tag{4.3.25}$$

or

$$\omega \leq \left(2\mu - \frac{1}{p-1}\right)q + 1 - (p-1)\mu - (1-\alpha). \tag{4.3.26}$$

The minimum value of the linear function of q on the right hand side in the interval $1 \leq q \leq p-1$ is

$$\begin{cases}
(p-1)\mu - (1-\alpha) & \text{at} \quad q = p-1 \quad \text{when} \quad \mu \leq \frac{1}{2(p-1)} \\
\frac{p-2}{p-1} - (p-3)\mu - (1-\alpha) & \text{at} \quad q = 1 \quad\;\; \text{when} \quad \mu > \frac{1}{2(p-1)}.
\end{cases} \tag{4.3.27}$$

In both cases, the minimum value is greater than or equal to ω according to the condition (4.3.9), and (4.3.15) holds for k. With the induction thus completed, (4.3.15) holds for any $k \in \mathcal{I}(i)$ and $m \in \mathcal{R}(k)$.

Now, since the claim (4.3.15) holds, the pivot u_{kk} of $k \in \mathcal{I}(i) \cup \{i\}$ is bounded from below. If $\mathcal{D}(k) = 0$, then row k is not eliminated from any row, so

$$u_{kk} = 1 > (1-\alpha)(p-1)\mu^2 \tag{4.3.28}$$

from the scaling assumption.

Next, we assume that $k \in \mathcal{I}(i)$ and $\mathcal{D}(k) > 0$. Then there are one or more nonzero elements on the left side of the diagonal. Therefore, from Eqs. (4.3.14) and (4.3.15),

$$r_k = \sum_{l \in \mathcal{L}(k)} (-a_{k\ell}) s_{\ell k} \geq \mu \cdot (1-\alpha)(p-1)\mu \geq (1-\alpha)(p-1)\mu^2. \tag{4.3.29}$$

Combing this with Eq. (4.3.10), we get

$$u_{kk} = r_k - \sum_{l \in \mathcal{R}(k)} a_{k\ell} \geq r_k \geq (1-\alpha)(p-1)\mu^2. \tag{4.3.30}$$

Finally, consider the case of $k = i$. Then there are no nonzero elements on the right side of the diagonal, so $\sum_{l \in \mathcal{L}(i)} (-a_{i\ell}) = a_{ii} = 1$. Therefore,

$$r_i = \sum_{l \in \mathcal{L}(i)} (-a_{i\ell}) s_{\ell i} \geq \left\{ \sum_{l \in \mathcal{L}(i)} (-a_{i\ell}) \right\} \cdot (1-\alpha)(p-1)\mu = (1-\alpha)(p-1)\mu, \tag{4.3.31}$$

and from Eq. (4.3.10),

$$u_{ii} = r_i \geq (1-\alpha)(p-1)\mu \geq (1-\alpha)(p-1)\mu^2. \tag{4.3.32}$$

□

While Theorem 4 places some restrictions on A, it covers the case where all nodes of $\mathcal{I}(i) \cup \{i\}$ are internal nodes, which we dealt with in Theorem 2.

In the case of the Poisson equation discretized with a 2D 5-point stencil, since $p = 4$ and $\mu = \omega = 1/4$, the inequality (4.3.9) holds at $\alpha \geq 5/6$. Also, in the 3D 7-point stencil case, since $p = 6$ and $\mu = \omega = 1/6$, the inequality holds in $\alpha \geq 13/15$. This inequality holds even for nonsymmetric matrices when the magnitudes of all non-diagonal non-zero elements fall within some narrow range.

4.4 Numerical Experiment

4.4.1 Test Problems and Computational Environment

We evaluated the performance of perturbed or relaxed MILU(0) preconditioner parallelized with BRB ordering. As test problems, we used the coefficient matrix and right-hand side vector for the Poisson equation (2.1.14) used to obtain the potential ϕ within the 3D magnetron sputter simulation shown in Fig. 2.2. At all boundaries, Dirichlet boundary conditions are imposed on ϕ. The analysis domain was the 60 mm ×60 mm ×30 mm parallelepiped shown in Fig. 2.4, and the equations were discretized using the 7-point finite difference method with a uniform orthogonal mesh of 1 mm or 0.5 mm.

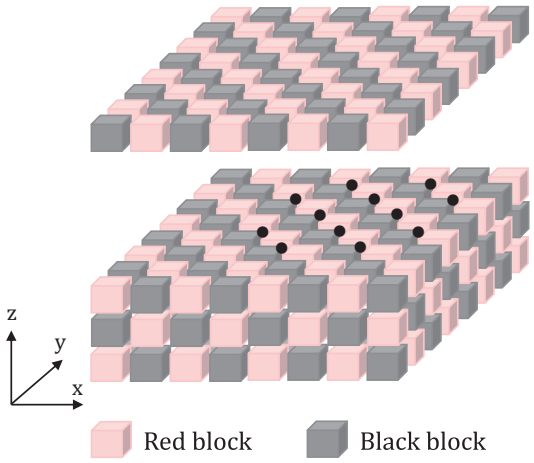

Figure 4.2: Zero pivot occurrence node (filled circles) in grid ordered by BRB ordering with $8 \times 8 \times 4$ blocks.

The number of unknowns is $59 \times 59 \times 29 = 100,949$ and $119 \times 119 \times 59 = 835,499$ for a grid size of 1 mm and 0.5 mm, respectively. The coefficient matrices and right-hand side vectors for the test problems were obtained from the 1,000th time step of the PIC simulation.

The preconditioned BiCGSTAB algorithm used to solve the linear equations is shown in Algorithm 5. The coefficient matrix and right-hand side vectors were reordered by BRB ordering with the first block in red. In the algorithm, forward/backward substitution, represented as 'solve', was parallelized on a block-by-block basis, while inner product, AXPY ($\alpha x + y$), sparse matrix-vector product, and other vector operations were parallelized on an element-by-element basis. The OpenMP `loop` directive was used for these parallelizations. The convergence criterion for the BiCGSTAB method is $\|\boldsymbol{r}_k\|_2 / \|A\boldsymbol{x}_0 - \boldsymbol{b}\|_2 < 10^{-8}$, where $\boldsymbol{r}_k$ and $\boldsymbol{x}_0$ are the residual vector at the k-th iteration and the initial estimated solution $\boldsymbol{x}_0 = \boldsymbol{0}$, respectively.

The numerical experiments were performed on a single node of Fujitsu PRIMEHPC FX10 with a SPARC64 IXfx@1.650 GHz (16 cores, 16 threads). The code was implemented using Fortran90 and OpenMP, and compiled using Fujitsu Fortran 1.2.1. Floating point data and operations were made to have double-precision.

4.4.2 Convergence Behavior

MILU(0)/MIC(0) factorization generates a zero pivot if the condition of Theorem 2 is satisfied. In this test problem, Assumptions 1 and 2 in Theorem 2 are satisfied, and the color of the first block is red. Therefore, zero pivot occurs if condition (b), the number of blocks in each direction is at least 4 and at least 5 in at least one direction, is satisfied. An example of $8 \times 8 \times 4$ blocks division that satisfies this condition is shown in Fig. 4.2. In this figure, the nodes where zero pivot is predicted to occur by Theorem 2 are shown as filled circles. These are the nodes with the largest node numbers in the internal black block, where all directions with decreasing node numbers are adjacent to the internal red block. In this experiment, we confirmed that the convergence speed actually deteriorated in the block division condition where these nodes occur.

The relationship between the number of iterations of MILU(0) preconditioned BiCGSTAB

Table 4.1: Number of blocks N and number of block divisions Nx, Ny, Nz in each axis direction.

N	N_x	N_y	N_z	Zero pivot condition
1	1	1	1	-
2	2	1	1	-
4	2	2	1	-
8	2	2	2	-
16	4	4	1	-
32	4	4	2	-
64	8	8	1	-
128	8	8	2	-
256	8	8	4	true
512	16	16	2	-
1024	16	16	4	true
2048	32	32	2	-
4096	32	32	4	true
8192	32	32	8	true
16384	32	32	16	true
32768	64	64	8	true
65536	64	64	16	true
131072	64	64	32	true

and the number of blocks N is shown in Fig. 4.3. The number of blocks N_x, N_y and N_z in each axial direction for the block N and whether they satisfy the zero pivot condition are shown in Table 4.1. In Fig. 4.3, the left end corresponds to natural ordering, where $N = 1$, and the right end corresponds to nodal RB ordering, where N is equal to the number of nodes. The values of N, for which no data is shown, correspond to the case where the calculation was terminated because the number of iterations exceeded 100,000. This graph shows that the number of iterations increases significantly in N where the conditions of Theorem 4.1 shown in Table 4.1 are satisfied. By contrast, for N where the condition was not satisfied, the number of iterations was suppressed even if N was large.

Figs. 4.4 and 4.5 show the convergence of two improved MILU(0) preconditioners, namely PMILU(0) and RMILU(0), respectively. For PMILU(0) or RMILU(0) with $\alpha \geq 13/15$, it is guaranteed by Theorem 3 or Theorem 4, respectively, that zero pivots will not occur in this problem. Both of these preconditioners converged in fewer iterations than ILU(0). As with the ILU(0) preconditioner, the number of iterations with these improved MILU(0) preconditioners increases with the number of blocks. When $N \leq 500$, $\zeta h^2 = 0.01$ or $\alpha = 0.95$ converges the fastest. As N increases, the difference due to these parameters becomes smaller. For the problem sizes of $59 \times 59 \times 29$ grid, with more than 10,000 blocks, and $119 \times 119 \times 59$ grid, with more than 100,000 blocks, the number of iterations of the improved MILU(0) is about the same as that of ILU(0). In summary, for these problem sizes, the improved MILU(0) is more effective than ILU(0) for block sizes up to a few thousand, and its number of iterations increases relatively gently with the number of blocks.

The convergence behavior at other steps of the PIC simulation is shown in Appendix A.

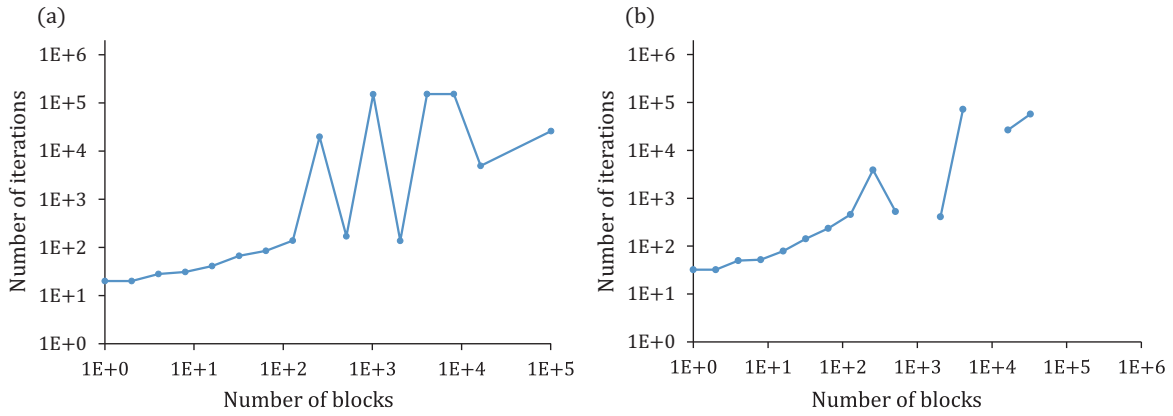

Figure 4.3: Number of blocks N and number of iterations required to converge with MILU(0) preconditioner. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

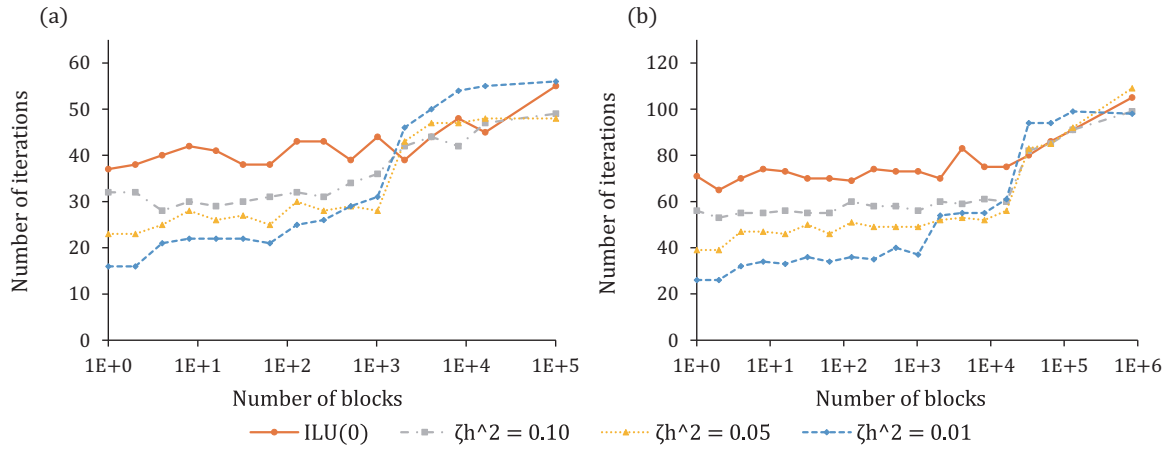

Figure 4.4: Number of blocks N and number of iterations required to converge with PMILU(0) preconditioner. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

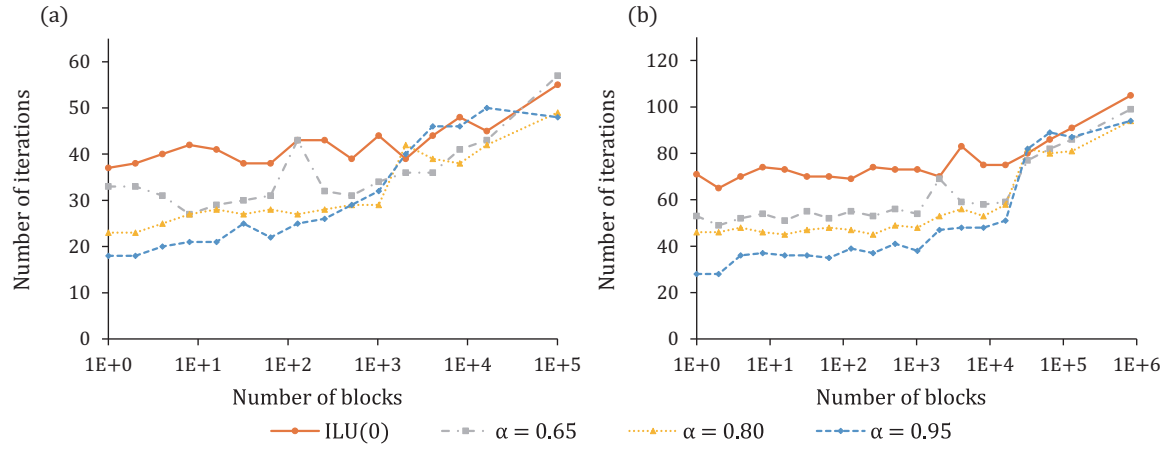

Figure 4.5: Number of blocks N and number of iterations with RMILU(0) preconditioner. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

4.4.3 Parallel Performance

Tables 4.2 and 4.3 show the parallel computation times for the problem sizes of $59 \times 59 \times 29$ grid and $119 \times 119 \times 59$ grid, respectively. Time [s] in the table includes ordering and factorization time. We used $\zeta h^2 = 0.01$ for the perturbation and $\alpha = 0.95$ for the relaxation. These are the parameters that achieve the minimum number of iterations when $N \leq 500$.

Note that Tables 4.2 and 4.3 are the results of parallelizing the operations on vectors, while Figs. 4.4 and 4.5 are the result of performing all operations sequentially. This causes slight differences in the results of the inner product calculation in the BiCGSTAB routine, and there are differences in the number of iterations required for convergence even under conditions with the same block partitioning and parameters.

For all problem sizes and block partitioning conditions corresponding to parallelization up to 16 threads considered here, the number of iterations was smaller for PMILU(0) and RMILU(0) than for ILU(0). The corresponding execution time was also shorter for PMILU(0) and RMILU(0) than for ILU(0). However, speed-up of PMILU(0) and RMILU(0) when the number of threads is increased is smaller that of than ILU(0). The reasons for this may be that the increase in the number of iterations due to the increase in the number of threads is larger for PMILU(0) and RMILU(0) than for ILU(0), and that the ratio of the sequential processing part such as ordering to the total time is larger because the total execution time for PMILU(0) and RMILU(0) is shorter.

4.5 Conclusion

In this chapter, we pointed out that zero pivots may occur in the combination of BRB ordering and unperturbed modified incomplete factorization, and showed that this problem may be mitigated by introducing perturbation or relaxation.

A necessary and sufficient condition for the occurrence of zero pivot in BRB ordering was derived based on Eijkhout's previous research. The condition is that there is an interior black block such that all adjacent red blocks are interior in the direction of decreasing x, y, z coordinates. Then the i-th pivot corresponding to the largest node number i in the block will be zero in the modified incomplete factorization without perturbation of the coefficient matrix.

To alleviate this problem, the introduction of perturbations to the diagonal elements or the relaxation of the compensation for the dropped fill-in can be effective. We have shown theorems showing that introducing perturbations or relaxations can resolve the zero-pivot problem and keep the pivot above a certain threshold, at least for a certain class of matrices.

The experimental results show that the convergence actually deteriorates for unperturbed MILU(0) with blocks that satisfy the condition for generating zero pivots, and convergence improves with the introduction of perturbations and relaxations. The number of iterations required for convergence has a trend of increasing with the number of blocks. However, up to a number of blocks on the order of 10^3 for problem sizes with $10^5 - 10^6$ unknowns, high convergence rate is achieved by combining the improved modified incomplete factorization with BRB ordering. These parallelized preconditioners are considered to be a promising choice for preconditioning in massively parallel environments.

Table 4.2: Number of threads and computation time for a problem size of $59 \times 59 \times 29$ grid ($h = 0.001$).

Thread	N	N_x	N_y	N_z	ILU(0)		
					Iteration	Time [s]	Speed-up
1	1	1	1	1	38	3.265	1.00
2	4	2	2	1	38	1.737	1.90
4	8	2	2	2	38	0.908	3.60
8	16	4	4	1	41	0.522	6.25
16	32	4	4	2	40	0.306	10.57
Thread	N	N_x	N_y	N_z	PMILU(0) ($\zeta h^2 = 0.01$)		
					Iteration	Time [s]	Speed-up
1	1	1	1	1	17	1.513	1.00
2	4	2	2	1	21	0.998	1.52
4	8	2	2	2	22	0.557	2.72
8	16	4	4	1	21	0.301	5.03
16	32	4	4	2	24	0.216	7.00
Thread	N	N_x	N_y	N_z	RMILU(0) ($\alpha = 0.95$)		
					Iteration	Time [s]	Speed-up
1	1	1	1	1	16	1.386	1.00
2	4	2	2	1	20	0.956	1.45
4	8	2	2	2	21	0.525	2.64
8	16	4	4	1	21	0.301	4.61
16	32	4	4	2	22	0.207	6.70

Table 4.3: Number of threads and computation time for a problem size of $119 \times 119 \times 59$ grid ($h = 0.0005$).

Thread	N	N_x	N_y	N_z	ILU(0)		
					Iteration	Time [s]	Speed-up
1	1	1	1	1	69	96.705	1.00
2	4	2	2	1	74	53.223	1.82
4	8	2	2	2	70	26.036	3.71
8	16	4	4	1	71	14.168	6.83
16	32	4	4	2	75	9.452	10.23
Thread	N	N_x	N_y	N_z	PMILU(0) ($\zeta h^2 = 0.01$)		
					Iteration	Time [s]	Speed-up
1	1	1	1	1	28	40.111	1.00
2	4	2	2	1	36	26.070	1.54
4	8	2	2	2	37	13.919	2.88
8	16	4	4	1	36	7.414	5.42
16	32	4	4	2	36	4.800	8.36
Thread	N	N_x	N_y	N_z	RMILU(0) ($\alpha = 0.95$)		
					Iteration	Time [s]	Speed-up
1	1	1	1	1	26	37.294	1.00
2	4	2	2	1	32	23.259	1.45
4	8	2	2	2	34	13.020	2.86
8	16	4	4	1	33	6.834	5.46
16	32	4	4	2	36	4.828	7.72

Chapter 5

GPU Acceleration of MILU(0) Parallelized by Block Red-Black Ordering

5.1 Introduction

In recent years, the use of accelerator devices for high-speed parallel computation has increased, and among them, parallelization strategies to take advantage of the high parallel computing power of GPUs have been extensively studied. In order to exploit the high-speed computational performance of GPUs, it is necessary to develop a highly parallel algorithm that can use a large number of threads, as well as to design an implementation that effectively utilizes the memory hierarchy. The development of APIs and libraries to assist GPU programming has reduced the cost of implementation for GPUs. Programming models widely used in GPU computing include compute unified device architecture (CUDA) [59] for NVIDIA GPUs and open computing language (OpenCL) [60] for cross-platform support. For easier implementation of GPU parallelism, there are also directive-based programming approaches, such as OpenACC [47] and OpenMP, which cover major platforms. OpenACC, which is one of the automation technologies for parallelization, enables the user to expand existing programs to use GPUs by merely inserting directives. In the field of GPU parallelization, OpenACC is ahead of OpenMP. The drawback of directive based parallelization is that maximum efficiency is not guaranteed without tuning [61]. However, since changes to the original program are kept to a minimum, it is suitable for maintaining programs over time and running them on a variety of computers.

As an application to matrix calculations, the performance improvement of GPU-parallelized computation by optimizing data layout and pipeline processing has been studied [62, 63]. In particular, the study of GPU implementations of sparse matrix solvers such as CG methods started from early on and their effectiveness has been established [64, 65]. Level scheduling and self-scheduling have been studied as common methods for parallel computation on GPUs of sequential operations in linear solvers such as incomplete factorization and forward/backward substitutions [66, 67, 68, 69]. Two approaches to level scheduling are available. The first is the self-scheduling algorithm, which executes a task when the corresponding dependent task is completed [68]. The second is an algorithm that creates a graph of task dependencies by analysis, and then applies parallel processing based on the graph [66]. The level scheduling

algorithm is effective when the dependency among the elements is small [69]. In the method of parallel processing of independent sets extracted by graph coloring [70], the dependencies between elements are represented by a graph. Nodes in the graph are colored to be different from neighboring nodes that have dependencies, and nodes with the same color are processed in parallel. Because the graph coloring problem is NP-complete, lower cost and sufficiently efficient approximation algorithms have been studied for GPU implementations [71, 70]. Alternatively, instead of sequential substitutions, methods that solve the triangular systems by iterative solvers [72], or that creates an explicit inverse of the preconditioner and calculating the product [73, 74] has also been proposed. Similarly, domain decomposition methods for improving parallelism are also used in combination with these approaches [75, 76]. Some of these results are provided as libraries for basic matrix operations [77, 78] and for calling linear solvers and preconditioning as functions [79, 80, 81, 82].

The combination of RMILU(0) preconditioner and BRB ordering with high convergence acceleration effect and a large degree of parallelism can be a viable alternative to the GPU acceleration methods mentioned above. Although the recent research has proposed a many-core implementation that combines ILU(0) preconditioner with block multicolor ordering [83], this method has so far been evaluated mainly on CPUs. Furthermore, for the combination of RMILU(0) and BRB ordering, even when the number of blocks is on the order of 10^3, which is necessary to take advantage of GPU parallelism, increasing the number of blocks does not necessarily lead to slower convergence if the problem size is large enough (see Figs. 4.5 and 5.12), and GPU parallelization can be expected to speed up the solution process.

Therefore, in this chapter, we evaluate the performance of this preconditioner on GPUs with high computing power. The solver targets 3D finite difference calculations using a Cartesian grid and a 7-point stencil. In addition, OpenACC, a directive based parallelization framework, is used for GPU parallelization to achieve high maintainability. In addition, several optimization strategies were explored, such as taking advantage of GPU's high single-precision floating point performance and improving memory access patterns. As a test problem, we used the Poisson equation (2.1.14) arising from PIC plasma simulations to investigate the effects of preconditioner and its optimization on GPU parallelization.

The rest of this chapter is organized as follows. Section 5.2 explains the basics of the CUDA architecture and OpenACC. In Section 5.3, we describe the details of the implementation of GPU parallelization using OpenACC and the optimization strategy. In Section 5.4, we validate the performance of our implementation and compare it with some existing methods from major libraries. Finally, we conclude this chapter with Section 5.5.

In this chapter, we use the empirically recommended value of $\alpha = 0.95$ [46] as the relaxation parameter α used in RMILU(0).

5.2 Fundamentals of GPU Computing

5.2.1 GPU Features and CUDA

GPU's superior processing power and memory bandwidth increase the execution speed of programs. In most cases, GPU communicates with CPU via Peripheral Component Interconnect Express (PCIe) and acts as a co-processor for host CPU. NVIDIA GPUs of the Pascal generation and later, including the Quadro GP100 used in this experiment, support NVLink, which is faster than PCIe. NVLink enables interconnections between GPUs and NVLink-enabled CPUs,

such as IBM Power microprocessors, and between GPUs and GPUs. As a general processing flow, first the input data is copied from CPU memory to GPU memory, then it is processed on GPU, and then the result of the parallel processing on GPUs is copied to CPU memory.

NVIDIA GPUs have multiple streaming multiprocessors (SMs) that run multiple threads in parallel. Each SM contains the CUDA core as the execution unit, scheduler, shared memory, and L1, constant, and texture caches.

When a kernel function is called from the host side, the control is transferred to the GPU device side. The parallel granularity in the logical components of CUDA consists of grids, thread blocks, and threads, in order of increasing size, and kernel functions are executed by threads in the grid. Threads in the grid are organized into thread blocks. Threads in a thread block have the same behavior on the same SM.

The hardware components of the device, SM, and CUDA core correspond to the logical components of the grid, thread block, and thread program, respectively. When the kernel grid is started, the thread blocks that make up the block are allocated to the SM. Each SM divides the allocated thread block into warps of 32 thread units and allocates hardware resources to them. Warp is the SM's execution unit. Warp threads, which have the address of current instruction and the state of the register, executes the instruction on their data. The number of warps running simultaneously is limited by the amount of resources available in the SM.

Two GPUs with different characteristics shown in Table 5.1 are used to compare the impact of their performance on computation speed. Both Quadro GP100 and Tesla K40t support double-precision computation, with the GP100 in particular offering enhanced double-precision performance and improved data operations. Each SM of Quadro GP100 has 64 single-precision CUDA cores and 32 double-precision CUDA cores. Hence, double-precision operations have 1/2 the performance of single-precision operations. Meanwhile, each SM of the Tesla K40t has 192 single-precision CUDA cores and 64 double-precision CUDA cores. Hence, the performance ratio is 1/3. The GP100 has 3584 CUDA cores that perform basic operations grouped into 56 SMs, while the K40t has 2880 CUDA cores grouped into 15 SMs. The GP100 is a configuration that contains more SMs with fewer cores than the K40t. As a result, the number of register entries and amount of shared memory that can be used by a single CUDA core is larger, and performance is improved by reducing the number of cases where instructions cannot be executed due to resource limitations shared by SMs. In addition, the GP100 SM has an improved warp scheduler and a higher clock rate. Due to the combined effect, GP100 achieves performance of 10.3 TFLOPS in single-precision and 5.2 TFLOPS in double-precision, whereas the performance of K40t is 4.29 TFLOPS in single-precision and 1.43 TFLOPS in double-precision.

Factors that limit the performance of the kernel include computing resources, memory bandwidth, and memory instruction latency. To improve these, the following tuning is effective.

Degree of parallelism: In order to maintain sufficient parallelism for each SM, the number of blocks in the kernel and the number of threads per block can be adjusted. In order to improve device utilization, it is effective to use multiple kernel calls on a single device, such as multiple grid-level parallelization.

Global memory access: To improve the performance of GPU parallel computing, optimization of memory transactions is essential. On this account, the two important concepts are memory alignment and coalescing. First, an aligned memory access occurs when the starting address of a device memory transaction is a multiple of the cache line granularity used for that transaction. Second, coalesced memory access occurs when 32 consecutive

Table 5.1: GPUs used in this study.

Property	Quadro GP100	Tesla K40t
Compute capability	6.0	3.5
Total amount of global memory	16278 MB	11520 MB
No. of CUDA cores	3584	2880
FP32 CUDA cores/GPU	3584	2880
FP64 CUDA cores/GPU	1792	960
Total amount of shared memory/block	49152 bytes	49152 bytes
Total No. of registers available/block	65,536	65,536
Warp size	32	32
Max No. of threads/multiprocessor	2,048	2,048
Max No. of threads/block	1,024	1,024
Max size of a thread block (x, y, z)	(1024, 1024, 64)	(1024, 1024, 64)
GPU max clock rate	1.44 GHz	0.75 GHz
Peak FP64 performance (board)	5.2 TFOPS	1.43 TFLOPS
Peak FP32 performance (board)	10.3 TFLOPS	4.29 TFLOPS
Memory clock rate	715 Mhz	3004 Mhz
Memory bus width	4096-bit	384-bit
L2 Cache Size	4194304 bytes	1572864 bytes
Total amount of constant memory	65536 bytes	65536 bytes
Max memory pitch	2147483647 bytes	2147483647 bytes
Max memory bandwidth	717 GB/s	288 GB/s
No. of SM	56	15
FP32 CUDA cores/SM	64	192
FP64 CUDA cores/SM	32	64

threads in a warp access consecutive memory locations. If these two conditions are met, the global memory access can be executed with as few transactions as possible. Thus, the memory bandwidth will be fully utilized. Furthermore, loop unrolling can be used to increase the number of independent memory operations performed to increase the parallelism of memory accesses.

Shared memory: Shared memory can be used as a programmable cache to avoid bank conflicts. It supports on-chip data reuse and improves the global memory access pattern, thereby reducing the required global memory bandwidth.

Register: The register is a resource divided among the active warps of the SM. Also, the smaller the number of registers used in the kernel, the larger the number of thread blocks allocated to each SM, which can increase the number of parallel threads per SM and improve occupancy and performance.

5.2.2 OpenACC

OpenACC [47] is a high-level programming model that absorbs barriers caused by the environment. OpenACC [47] offloads the region of code specified by the directive to an accelerator device such as GPUs. OpenACC frees the programmer from the task of managing communication and specifying parallelization details, and allows the programmer to parallelize the program with less effort than CUDA. The program is easy to maintain because the original code can be kept intact, and for this reason, it is very powerful for parallelizing the large programs on GPUs. While OpenACC is more versatile, certain environment-specific features, such as the shared memory of NVIDIA GPUs, can be difficult to use with OpenACC.

Unlike the case where parallel processing is performed on shared data on the same memory from the same CPU, when accelerator devices are used, processing related to data management is required. The declaration of variables on GPUs and the copying of data between host devices and GPUs are indicated by the data directive. Parallelization of loops or groups of loops can be specified by inserting a directive into the part to be parallelized, as in OpenMP. In OpenACC, parallelization is indicated by the accelerator compute directive. If the user wants to set any additional options for the directive, the clause list is appended after the directive.

To transfer data from CPUs to GPUs, insert the `data copyin` directive specifying the array or variable to be transferred, at the point of transfer. To transfer data from GPUs to CPUs, insert the `data copyout` directive in the same way. In addition to these clauses, `create` clause, which directs data generation on GPU, `delete` clause, which directs data deletion on GPUs, and other clauses are used together in data directive. `data` directive for dynamically allocated data can be omitted by activating unified memory. For loops that use data on GPUs, it is possible to specify the existence of data by a clause in the accelerate compute directive, and to specify what to do if the data does not exist. However, the control of data placement in GPUs is automatically generated by OpenACC.

The parallelization of the computation is specified by the accelerate compute directive. The clause set in the accelerate compute directive can be used to specify asynchronous processing up to the synchronization point. Conditional branching based on the device specified conditions and specification of parallel granularity are also supported. There are three levels of parallel granularity that can be specified in OpenACC, namely, `gang`, `worker`, and `vector`, from the coarser level to the finer level. The largest granularity of `gang` corresponds to a thread block in

CUDA and consists of one or more execution threads. For each stream processor, one `gang` is scheduled at a time. `worker` corresponds to a warp in CUDA of `gang` and is composed of one or more `vector` elements with a fixed vector width. `vector` elements correspond to threads in CUDA contained in a single execution stream, and the operations in the unit `vector` are SIMD and vector operations contained in `worker`. The major difference between the OpenACC and CUDA thread models is that in OpenACC, the concept of `worker` (warp) is directly provided in the programming model, while CUDA programmers do not need to be aware of warp.

`kernels` directive and `parallel` directive are accelerate compute directives. The `kernels` area is executed in parallel within the `gang` unit and the `parallel` area is executed in parallel in multiple `gang` units. In the `kernels` clause, the `loop` directive and the parameters that specify parallel granularity, `gang`, `worker`, and `vector`, can be omitted. In that case, the size of each of the three layers is automatically selected so that all layers are used as much as possible to parallelize the loop. The user can also specify the size by placing numbers in brackets after the gang, worker, and vector respectively. A proper choice of these parameters will improve the performance. In the accelerator compute directives, the data used in computation is specified by a clause: `present` clause is inserted to use data already presented in GPUs. `present` clause can be omitted when transferring data via data directive within the same subroutine or unified memory is enabled. `loop` directive placed after the accelerate compute directive is used to explicitly indicate the independence of the loop, the sequential processing, and the settings of `gang`, `worker`, and `vector`.

5.3 GPU Implementation of Preconditioned Iterative Solver

In this section, we implement our BiCGSTAB solver, which combines BRB ordering and RMILU(0) preconditioner, on GPU.

5.3.1 The original code

Our code consists of the reordering of the coefficient matrix, the solution and the right-hand side vectors by BRB ordering, the RMILU(0) factorization of the coefficient matrix, and the BiCGSTAB solver.

The preconditioned BiCGSTAB (PBiCGSTAB) solver is the main part of this program, and its algorithm is shown in Algorithm 5. Most of the computational load of this algorithm is accounted for by the forward/backward substitutions shown as 'solve' in lines 14 and 21, and the sparse matrix-vector product (SpMV) shown in lines 15 and 22.

As an example of substitution calculation, the loop structure for forward substitution of the red block is shown in Algorithm 10. This corresponds to the computation for node i in the red block in the forward substitution (3.2.39).

The LU-factorized elements of the $n \times n$ sparse matrix A used in the substitution calculation are stored in the modified sparse row storage (MSR) format [11]. The array alu stores the diagonal elements of the row corresponding to the index in its first n positions. Subsequent positions store the value of non-diagonal nonzero elements. The array jlu stores an index, which is the starting position in common to alu and jlu of the row corresponding to the index, in its first n positions. Subsequent positions store the column indices of non-diagonal nonzero elements. The array ju stores the starting index of the upper triangular part.

Algorithm 10 Forward substitution of red blocks

```
1: for iblock = 1 to redblocknum do
2:   for i = istart(iblock) to iend(iblock) do
3:     s = y(i)
4:     for k = jlu(i) to ju(i) − 1 do
5:       s = s − alu(k) ∗ x(jlu(k))
6:     end for
7:     x(i) = s
8:   end for
9: end for
```

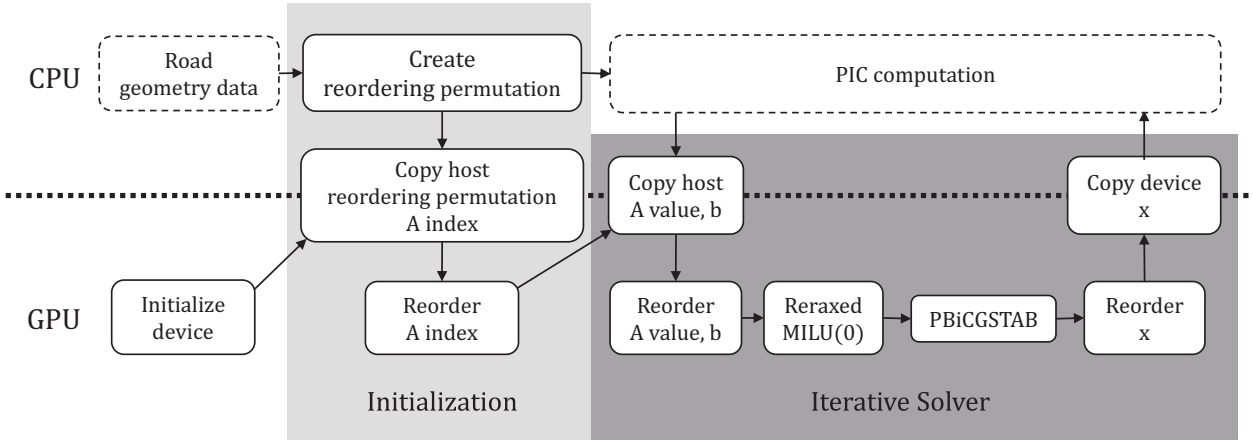

Figure 5.1: CPU and GPU tasks and data flow.

In the loop statement, *redblocknum* contains the number of red blocks. *istart*(*iblock*) contains the index of the first row of the (*iblock*)-th red block, and *iend*(*iblock*) contains the index of the last row of the (*iblock*)-th red block. Since each red block can be computed independently, the red block *iblock* can be computed in parallel if the outermost loop is parallelized. In forward substitution, after all the red block rows have been processed, synchronization is required before processing of the black block rows can begin. With the results of the red block, each black block can be computed independently, so parallel computation can be done for these blocks as well as for the red block. In backward substitution, as in forward substitution, the black block can be computed in parallel, and then, after synchronization, the red block can be computed in parallel.

There is no sequential nature in the SpMV process, and each element of the resulting vector can be calculated in parallel, just like vector operations such as the AXPY operation.

5.3.2 Method of GPU Implementation by OpenACC Directives

The tasks and data flows performed by CPU and GPU are shown in Fig. 5.1.

The index array of the coefficient matrix and the array of reordering permutation generated by the CPU are both sent to the GPU. Reordering using these arrays is executed on the GPU. The array of values in the coefficient matrix and the array of right-hand side vectors are generated each time a potential is needed by the PIC method, and sent to the GPU to be

reordered. For data transfer between the host CPU and the device GPU, the `data` directive was used. The parallel computation directives for the loops in RMILU(0) (Algorithm 9) and in PBiCGSTAB (Algorithm 5) used the `present` clause to tell them to use the data that has already been transferred. Listing 5.1 shows a naïve implementation of the forward substitution of red blocks (Algorithm 10) using OpenACC. Just by inserting a few lines of directives beginning with '`!$acc`' before and after the loop, OpenACC will parallelize the loop on the GPU. In the substitution calculation, the `loop` directive with the `independent` clause was inserted above the block loop to indicate parallel computation of blocks with the same color. The size of the respective parallel granularity was also added to the `loop` directive by the `gang`, `worker`, and `vector` clauses. The option whether to use the L1 cache or not and the register size are specified in the compile options.

Listing 5.1: OpenACC implementation of Algorithm 10

```
!$acc kernels present(istart,iend,jlu,ju,alu,x,y) async(0)
!$acc loop independent gang vector(128)
do iblock = 1, redblocknum
  do i = istart(iblock), iend(iblock)
    s = y(i)
    do k = jlu(i), ju(i)-1
      s = s - alu(k)*x(jlu(k))
    enddo
    x(i) = s
  enddo
enddo
!$acc end kernels
!$acc wait(0)
```

The obtained solutions are reordered into the natural ordering and then sent to the CPU by the `data copyout` directive.

However, it is difficult to fully exploit the computational power of GPUs with a naïve implementation that just inserts simple directives. In the next section, we explain the problems of the naïve implementation and propose possible solutions.

5.3.3 GPU Implementation: Issues and Solutions

In the following, we list the challenges of GPU implementations along with the problems of the naïve implementation.

- **Ensuring Sufficient Parallelism to Run Many GPU Threads**
  In BRB ordering, the number of blocks that have the same color, or in other words, one-half of the total number of blocks, is the degree of parallelism. Since the GP100 has 3584 cores and the K40t has 2880 cores, the total number of red and black blocks required to use all the cores is at least 7168 or 5760. The BRB ordering has a potential to parallelize the RMILU(0) factorization using the order of 10^3 blocks for the model size targeted in this study without significantly reducing the convergence speed, as shown in Fig. 5.12.
- **Realizing Coalesced Memory Access in the Forward/Backward Substitution**
  In the coefficient matrix sorted by the BRB method, the matrix elements are stored as shown in Fig. 5.2 if the data is stored in row or column wise order. In block-parallelized

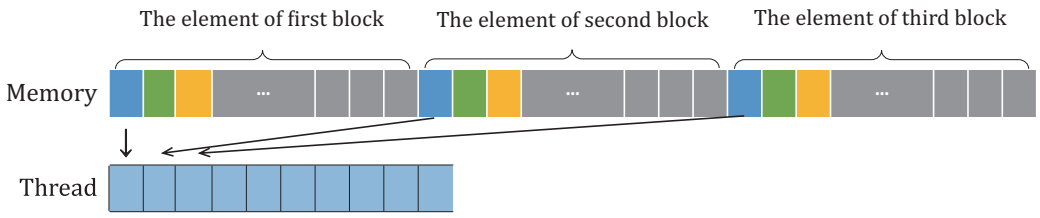

Figure 5.2: Naïve data storage for non-coalesced memory access.

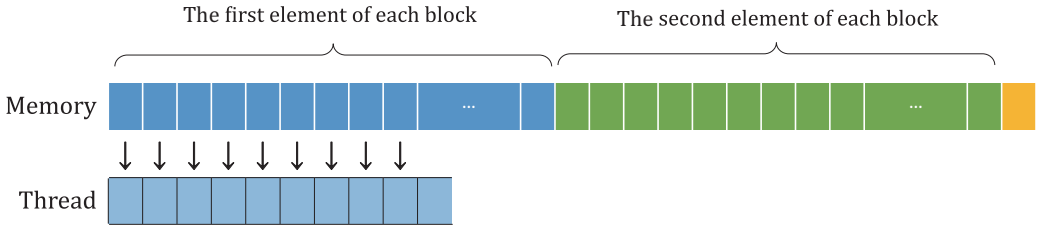

Figure 5.3: Improved data storage for coalesced memory access.

substitution computation, when one thread in a warp accesses the i-th element of a block, the adjacent thread accesses the i-th element of the adjacent same-color block. These elements are as far apart in memory as the number of elements in the block, and thus reloading occurs. As a result, coalesced memory access, where neighboring threads access neighboring memory, cannot be achieved, and performance is degraded.
To solve this problem, we changed the way the matrix elements are stored, as shown in Fig. 5.3. The data is stored in such a way that the first element of all blocks with the same color is first stored in a contiguous region, and then the second element of each block is stored in the subsequent region. Similarly, the i-th element of all blocks of the same color is stored in a contiguous region. This allows 32 consecutive threads in the warp to access contiguous memory locations and fully utilize the global memory bandwidth through coalesced memory access. The solution vector and the right-hand side vector used for the substitution calculation were also stored in a contiguous region with the i-th element of all blocks of the same color so that coalesced memory access is achieved when blocks of the same color are calculated in parallel. To ensure that operations on these input and output vectors, other than the assignment calculations in PBiCGSTAB, are coalesced memory accesses, the data for other vectors and coefficient matrix are stored in the same order, and the vector operations and SpMVs are block-parallelized so that one thread is responsible for one block. This reordering is implemented in 'Reorder A index' and 'Reorder A value, b', shown in Fig. 5.1.

- **Increasing the Number of Independent Memory Operations**
  To keep the GPU pipeline running for as much time as possible, we unrolled the innermost loop of the forward/backward substitutions and SpMV to increase the number of independent memory operations. In BRB ordering, the nodes in a block are numbered in the natural order, so the number of nonzero elements in one row of the L and U matrices can be easily estimated. Since a 7-point stencil is assumed in this study, the maximum number of nonzero off-diagonal elements in the L matrix used the forward substitution is 3 for the row corresponding to the node in the red block and 6 for the row corresponding

to the node in the black block. Therefore, we unrolled the loops by fixing the length of the innermost loop to 3 and 6 for the red and black node rows, respectively. Zero elements were padded in rows where the number of elements was less than this fixed length. The innermost loop of the backward substitution using the U matrix was similarly unrolled with the black and red node rows fixed at 3 and 6, respectively. In SpMV, the innermost loop length was set to 7, with zero padding where necessary. Our implementation strategy can also be applied to structural grid with more complex stencils, e.g., 27-point stencils. In that case, the innermost loop length can still be estimated using the same method.

- **Exploiting the Single-Precision Performance of GPU**
  To reduce the amount of data and to take advantage of the high single-precision performance of GPUs, we use single-precision floating point in part of BiCGSTAB method. When using right preconditioning (lines 14-15 and 21-22 of Algorithm 5) as a preconditioning for Krylov subspace method, it is known that the accuracy of the computed solution does not degrade even when preconditioning is done in single-precision because the relationship between the computed solution and the residuals is preserved [84]. Therefore, in this implementation, L and U matrices are stored as single-precision floating point arrays, and RMILU(0) factorization and the substitution calculation using them are performed in single-precision. The vectors $\hat{\boldsymbol{p}}$ and $\hat{\boldsymbol{s}}$, which are the outputs of the substitution calculations in the BiCGSTAB routine, are also stored as single-precision floating point arrays. For all other floating point data and operations, double-precision is used.
  Preconditioning using single-precision is equivalent to using slightly different preconditioning matrices at each iterative step. When the preconditioner is variable in this way, it is called flexible Krylov subspace method, which in a narrow sense uses an algorithm modified to compensate for errors due to variable preconditioner, and in a broad sense includes combinations with standard algorithms [86, 87]. The former correction is made to satisfy local conjugacy in CG method [85], for example, but it is not possible to make a correction that generally preserves the two conditions of bi-conjugacy in BiCGSTAB method. Therefore, we use the standard BiCGSTAB algorithm in our implementation.
  Fig. 5.4 shows the convergence history of the relative residual norm implemented with this mixed-precision in BiCGSTAB. The implemented mixed-precision version showed almost the same convergence behavior as the version that used double-precision for all floating point data and operations. Analysis of the variation of the residual vector of BiCGSTAB due to changes in the preconditioning matrix shows that the former can be bounded by a constant multiple of the latter [87]. Since the variation of the preconditioning matrix due to the use of single-precision is approximately $O(\sqrt{\epsilon})$, where ϵ is the rounding unit of double-precision, this would justify the use of the standard BiCGSTAB method.

5.3.4 GPU Implementation: Improvements

5.3.4.1 Structure of the Code

The following is the list of tasks performed by our code, which employs the improvements described in the previous section. The data and processing flow shown in Fig. 5.1 will remain unchanged except for the fact that the data layout will change as the order for coalesced memory access is sorted during reordering.

Initialization

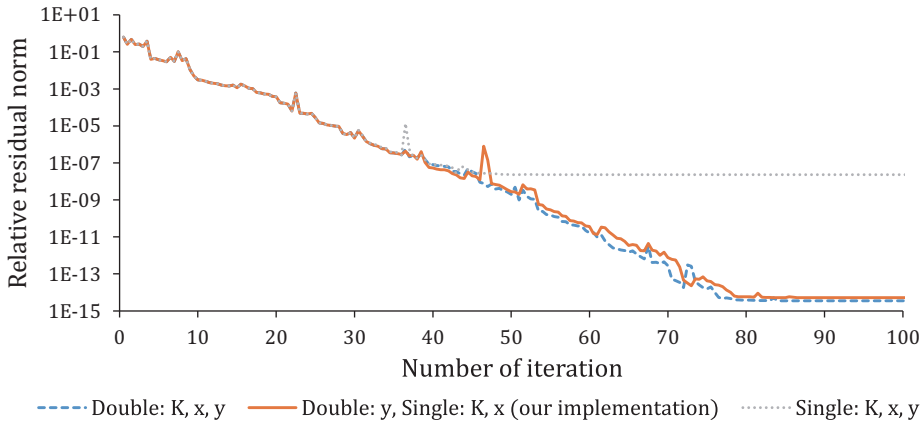

Figure 5.4: Convergence history of BiCGSTAB with mixed-precision $K\boldsymbol{x} = \boldsymbol{y}$ for test problem at PIC 10,000th step on (a) $59 \times 59 \times 29$ grid.

- **Create reordering permutation**
  Create a permutation array to be used for array reordering. This permutation array is used to reorder the coefficient matrices and right-hand side vectors of the lexicographic order stored in CSR format, and to achieve coalesced memory access during the block parallel computation with BRB ordering.
- **Copy reordering permutation and A index from host**
  These arrays, generated on the CPU, are sent to the GPU via the `data copyin` directive.
- **Reorder A index**
  The index array of A is reordered on the GPU using the permutation array.

Iterative solver

- **Copy A value and $\boldsymbol{b}$ from host**
  The array of values of A and $\boldsymbol{b}$ generated by the PIC simulation running on the CPU is transferred to the GPU via the `data copyin` directive.
- **Reorder A value and $\boldsymbol{b}$**
  The values of A and $\boldsymbol{b}$ are reordered using the permutation array on the GPU. For incomplete factorization, the lower triangular part, upper triangular part, and diagonal part of the coefficient matrix are allocated to separate single-precision floating point arrays. These data are stored in the rearranged position, while being converted to single-precision. The values of the coefficient matrices used in the SpMV of lines 15 and 22 of Algorithm 5 are stored in double-precision in the rearranged position of the array of double-precision floating point numbers.
- **Relaxed MILU(0)**
  Execute the RMILU(0) factorization shown in Algorithm 9. Since the matrix is structured as shown in Fig. 3.2, the computation of blocks with the same color can be performed in parallel. The results of factorization are overwritten on the single-precision arrays of the lower triangular part, upper triangular part, and diagonal part, respectively.
- **PBiCGSTAB**
  Listing 5.1 is rewritten to Listing 5.2 to support the modified data layout for coalesced

access when block elements are computed in parallel, and to unroll the innermost loop. Vector operations in PBiCGSTAB, such as SpMV, inner product, and AXPY, are also computed in parallel on a block-by-block basis in the same way as substitutions to achieve coalesced memory access. The implementation of SpMV is shown in Listing 5.3. Listing 5.4 shows an OpenACC implementation of the vector inner product for $\boldsymbol{s}$-norm of line 18 in Algorithm 5 as an example of basic vector operation.

Listing 5.2: OpenACC implementation of Algorithm 10

```
!$acc kernels present(istart,iend,jl,alr,x,y,jlcolr) async(0)
!$acc loop independent gang(1024) worker(2) vector(16)
do iblock = 1, lastblock
  x(iblock) = y(iblock)
  rownum = nblock * (iend(iblock) -istart(iblock)) + iblock
  do i = (iblock + nblock), rownum, blocknum
    s = y(i)
    k = jl(i)
    s = s-alr(k)              *x(jlcolr(k))
    s = s-alr(k+tempnblock)   *x(jlcolr(k+tempnblock))
    s = s-alr(k+2*tempnblock)*x(jlcolr(k+2*tempnblock))
    x(i) = s
  enddo
enddo
!$acc end kernels
!$acc wait(0)
```

Listing 5.3: OpenACC implementation of SpMV

```
!$acc kernels present(istart,iend,a,ja,x,y,ia) async(0)
!$acc loop independent gang worker(2) vector(16)
do iblock = 1, nblock
  rownum = nblock * (iend(iblock) -istart(iblock)) + iblock
  do i = iblock, rownum, blocknum
    k = ia(i)
    s =     a(k)           * x(ja(k))
    s = s + a(k+nblock)    * x(ja(k+nblock))
    s = s + a(k+2*nblock)  * x(ja(k+2*nblock))
    s = s + a(k+3*nblock)  * x(ja(k+3*nblock))
    s = s + a(k+4*nblock)  * x(ja(k+4*nblock))
    s = s + a(k+5*nblock)  * x(ja(k+5*nblock))
    s = s + a(k+6*nblock)  * x(ja(k+6*nblock))
    y(i) = s
  enddo
enddo
!$acc end kernels
!$acc wait(0)
```

Listing 5.4: OpenACC implementation of dot product

```
!$acc kernels loop independent gang vector(64) reduction(+:snrm)
do iblock = 1, nblock
  do i = iblock, rownum + iblock, blocknum
    snrm = snrm + s(i) * s(i)
  enddo
enddo
!$acc end kernels
```

- **Reorder $\boldsymbol{x}$**
  The array of solution vector $\boldsymbol{x}$ after convergence is reordered into a natural ordering using the permutation array.
- **Copy $\boldsymbol{x}$ from device**
  The reordered array of solution vector $\boldsymbol{x}$ is sent from the GPU to the CPU via the `data copyout` directive.

While BRB ordering is based on known structural grid, in this chapter we optimize the implementation based on common sparse matrix formats as shown in Listings 5.2 and 5.3. There have been studies to accelerate the matrix computation using such grid information on GPUs [88, 89, 90]. An example of improving the proposed implementation that uses information about the structure of the grid to avoid indirect references is provided in Appendix C.

5.3.4.2 Parameter Optimization

The code and calculation method have adjustable parameters that affect the calculation speed. In the following, we discuss their optimization to improve performance. The test problems and execution environments shown in Section 5.4.1 are used to optimize the parameters.

The Number of Blocks and the Parallel Granularity

The number of blocks is chosen to balance the generally conflicting requirements of being small enough not to adversely affect convergence speed, but large enough to take advantage of GPU's many cores. The relationship between the number of iterations and the number of blocks for PBiCGSTAB method is shown in Fig. 5.5. Since we are evaluating all possible block division patterns when the number of grids is (a) $59 \times 59 \times 29$, we have multiple plots with the same number of blocks. In other words, even given the same number of blocks, there are several choices on how to divide each axial direction of x, y, z, and the different orderings resulting from them affect the number of iterations required for convergence.

To estimate the effect of reordering on convergence rate, we can use the Frobenius norm of the remainder matrix $R = A - LU$ [53], the simple remainder index (S.R.I.) [15], or the incompatibility ratio [54]. In S.R.I., the average of I_{rsl} of all nodes is calculated as S.R.I., given the number $I_{rsl} \equiv_\ell C_2 = \ell(\ell-1)/2$ of each node, where ℓ is the number of neighboring nodes with larger node numbers. The I_{rsl} of a node is uniquely determined from the number of adjacent nodes whose node number is greater than itself. This indicator is based on the fact that neighboring nodes with higher numbers contribute to the update of R. Thus, the S.R.I. shows how many times R has been updated as a whole, and shows the same trend as the R norm. Incompatibility ratio is the percentage of incompatible nodes where both upwind and downwind neighbors in one or more directions have a node number greater than their own. This

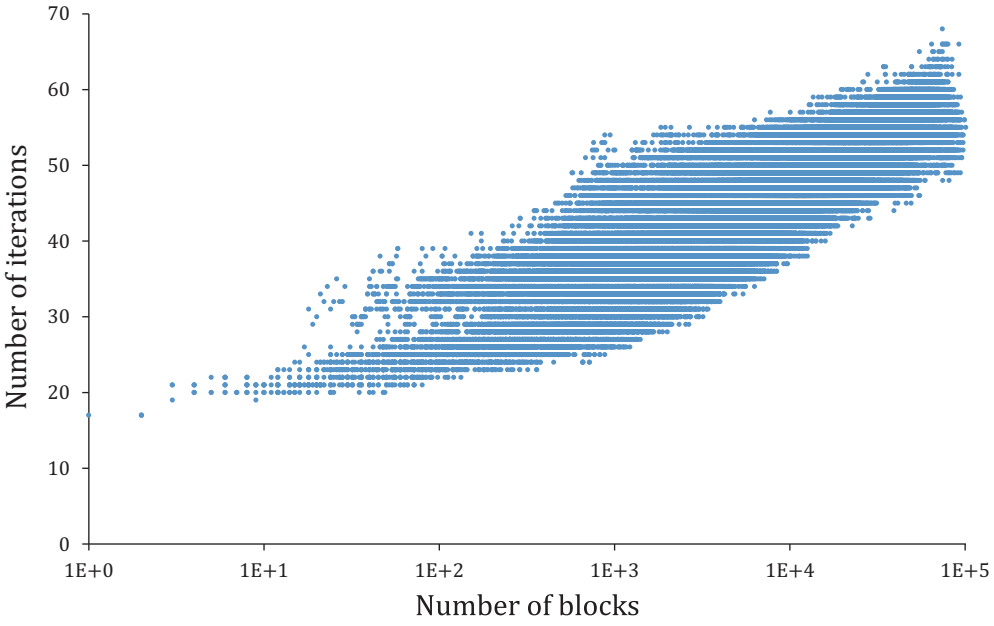

Figure 5.5: Number of blocks and number of iterations required for convergence of test problem at PIC 10,000th step on (a) $59 \times 59 \times 29$ grid.

indicator is also related to the degree of updating R. As with the R norm, it is known that orderings with small values of S.R.I. or incompatibility ratio tend to converge quickly.

The relationship between the number of iterations for PBiCGSTAB method and S.R.I. or incompatibility ratio is shown in Fig. 5.6 and Fig. 5.7, respectively. Here, we use BRB ordering and RMILU(0) factorization as preconditioning and evaluate all block division patterns as in Fig. 5.5. Although there is variation in the number of iterations for the same value of the index, the smaller the index, the better the convergence tends to be. In the case of BRB ordering over a 3D Cartesian structure lattice, nodes with large I_{rsl} are those located in the three planes with the smallest x, y and z coordinates in each red block, and these nodes are incompatible nodes (unless these planes are the boundaries of the computational domain).

Thus, the smallest S.R.I. and incompatibility ratio are obtained when $x = y = z = \sqrt[3]{n/(2p)}$, where n is the number of nodes and p is the number of parallel threads. The corresponding method of block division is the condition that each block is as close to a cube as possible.

Fig. 5.8 shows the relationship between the calculation time and the number of blocks in PBiCGSTAB. Here, the block division condition was determined using the method described above.

In order to reduce the computation time, it is necessary to select the smallest possible number of blocks that satisfies the following two conditions:

(1) The total number of blocks is a multiplier of 32, where 32 is the number of threads per warp, the execution unit of CUDA.
(2) The number of blocks per SM is greater than or equal to 32×6.

However, when the problem size is small, selecting the number of blocks to meet these requirements may result in a very small block size. This can increase the number of iterations required

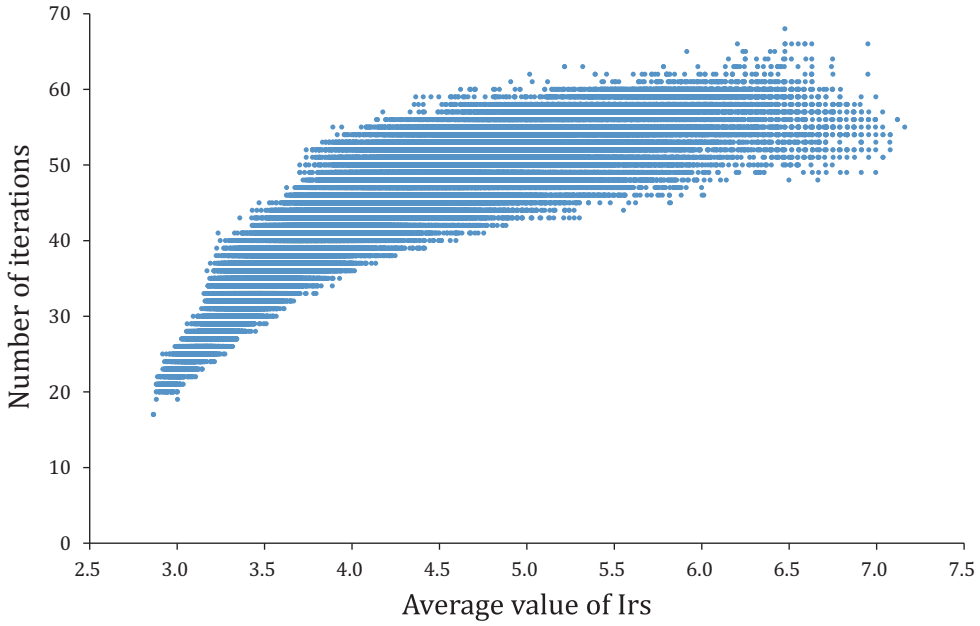

Figure 5.6: S.R.I. and number of iterations required for convergence of test problem at PIC 10,000th step on (a) $59 \times 59 \times 29$ grid.

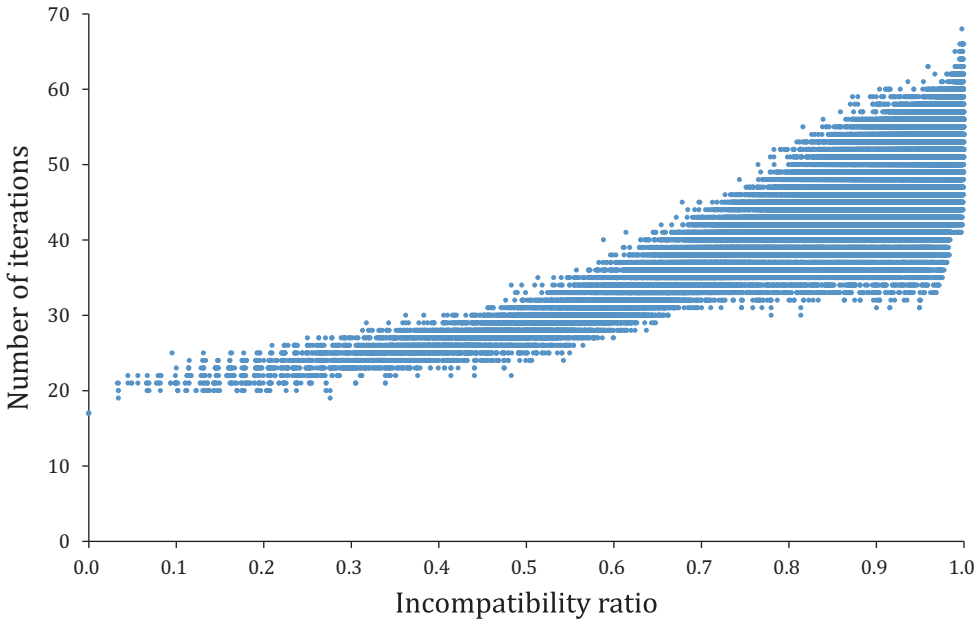

Figure 5.7: Incompatibility ratio and number of iterations required for convergence of test problem at PIC 10,000th step on (a) $59 \times 59 \times 29$ grid.

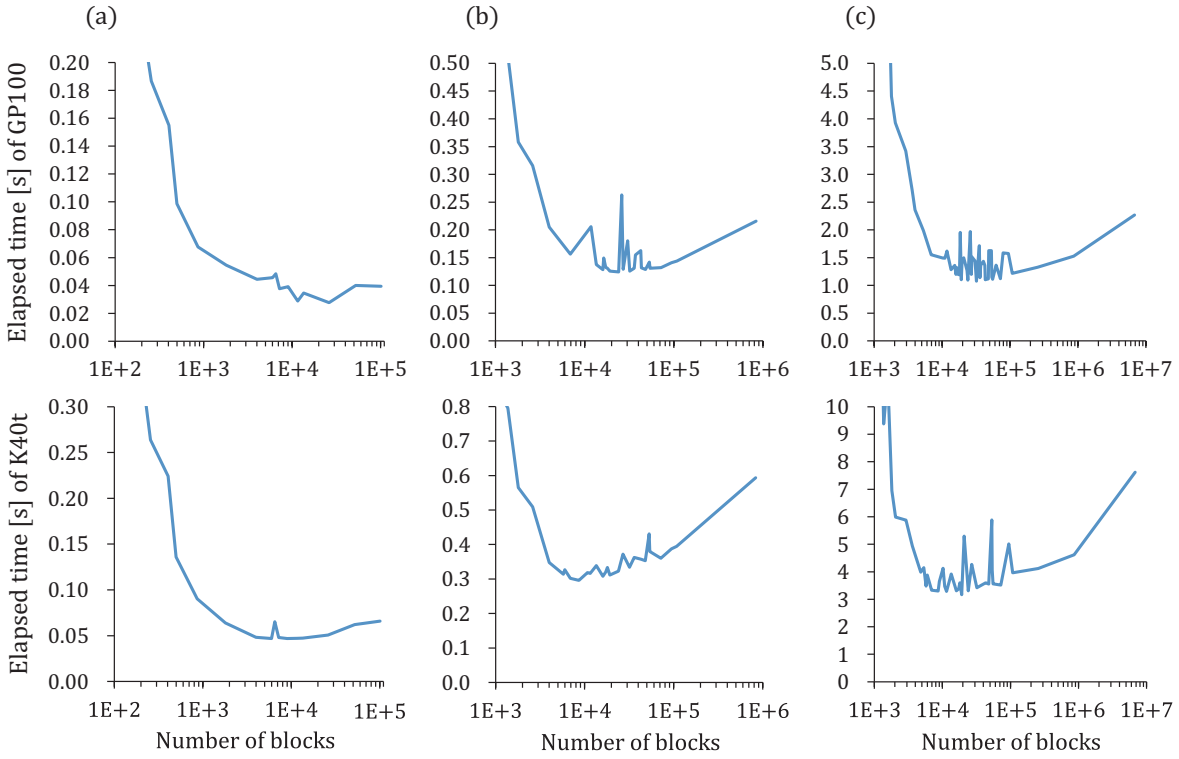

Figure 5.8: Computation time as a function of the number of blocks (proposed implementation).

Table 5.2: Number of blocks with optimized partitioning conditions.

		GP100	K40t
(a)	$59 \times 59 \times 29$ grid	26100 blocks	6000 blocks
(b)	$119 \times 119 \times 59$ grid	32000 blocks	8640 blocks
(c)	$239 \times 239 \times 119$ grid	32000 blocks	19200 blocks

for convergence, resulting in a larger computation time. In this case, choosing a larger block size that does not satisfy (1) and (2), or in other words, a smaller number of blocks, can reduce the overall execution time.

Table 5.2 shows the block partitioning condition that results in the fastest solution with these conditions taken into account. Thus, in the following experiments, we used 26100 blocks for (a) $59 \times 59 \times 29$ grid, 32000 blocks for (b) $119 \times 119 \times 59$ grid and (c) $239 \times 239 \times 119$ grid on GP100. Also, we used 6000 blocks for (a) $59 \times 59 \times 29$ grid, 8640 blocks for (b) $119 \times 119 \times 59$ grid and 19200 blocks for (c) $239 \times 239 \times 119$ grid on K40t.

Parallelization granularities `gang`, `worker` and `vector`

`gang`, `worker` and `vector` are the parallelization granularities that can be specified in OpenACC, corresponding to thread blocks, warps and threads in CUDA, respectively (see Section 5.2.2). Figs. 5.9 – 5.11 show the relationship between the average calculation time of

the loops of red block forward substitution (Listing 5.2), SpMV (Listing 5.3) and dot product (Listing 5.4) and the values of `gang`, `worker` and `vector`, given as clauses for the `kernels` directive.

Here, we use the block division condition shown in Table 5.2.

In most cases where `worker` × `vector` was the same, the computation time was almost the same. When 1 was specified for `worker` or `vector` different results were obtained from other conditions where `worker` × `vector` was the same. This often happens, especially when the vector of K40t is set to 1.

For all problem sizes and GPUs, the computation time will be longer if the value of `gang` × `worker` × `vector` is too small to use all cores. The computation time also increases when the value of `gang` × `worker` × `vector` is very large, which is often the case when the problem size is small. The optimal condition lies somewhere in between, and its location depends on the problem size, the number of blocks, and the operations performed in the loop. Even if the value of `gang` × `worker` × `vector` is appropriate, an inappropriate combination will increase the computation time. If the values of `gang` and `worker` are omitted, the result is almost always the same as if the good condition was set. However, for loops with a lot of data loading and random access, omission is not a good choice, and the performance peaks lie on the side where `gang` × `worker` × `vector` is small.

If the `parallel` directive is used, the result will be the same as any other condition where `worker` × `vector` is equal even if either `worker` or `vector` is set to 1. Except for this point, the computation times for `parallel` and `kernels`, where `gang`, `worker`, and `vector` are the same, are almost the same.

The values of `gang`, `worker`, and `vector`, which gave good results for the main parallel operations on GP100 and K40t are shown in Table 5.3. No significant speedup was observed by using L1 cache or limiting the number of registers.

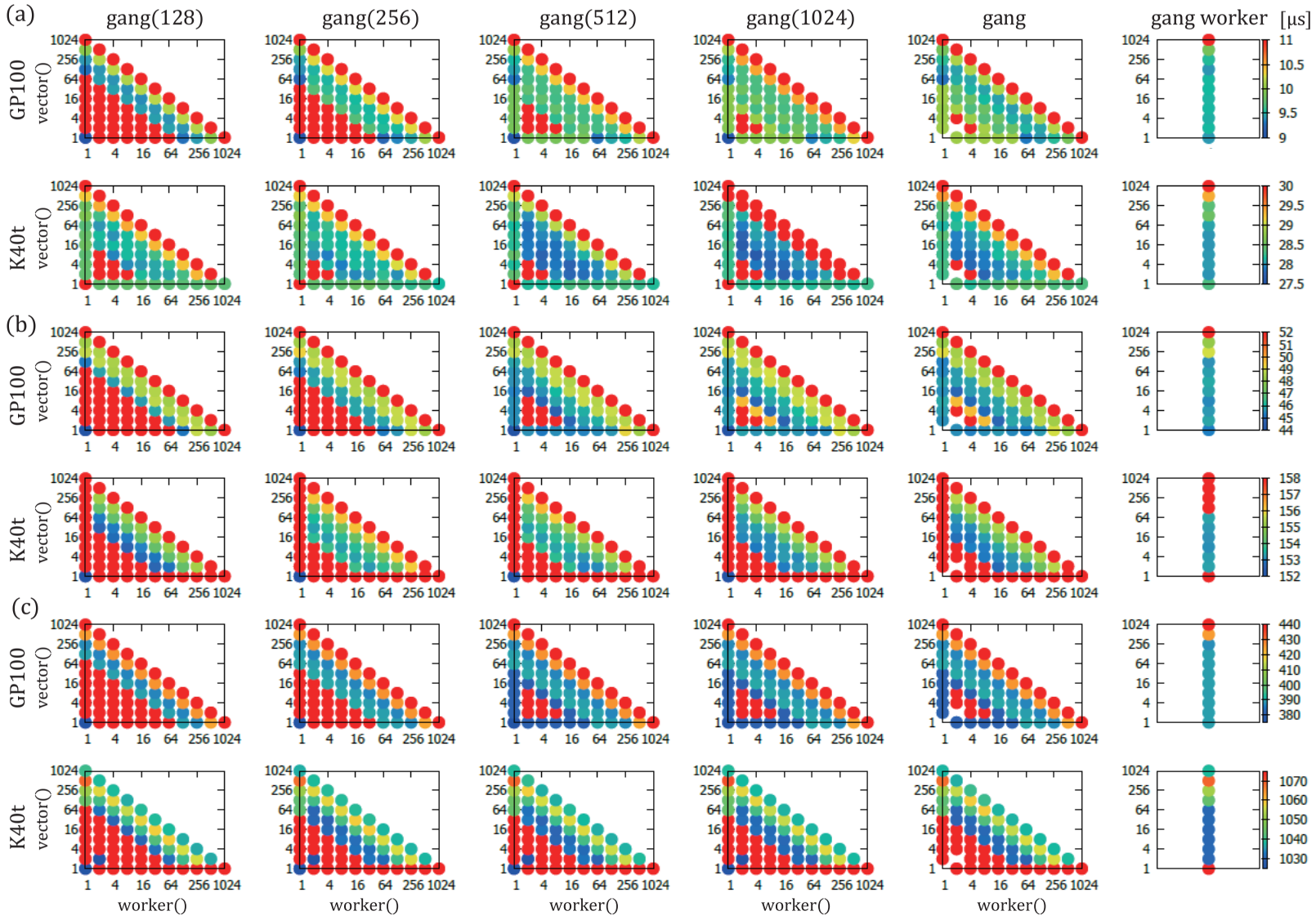

Figure 5.9: `gang`, `worker`, and `vector` values and average computation time for the red block forward substitution.

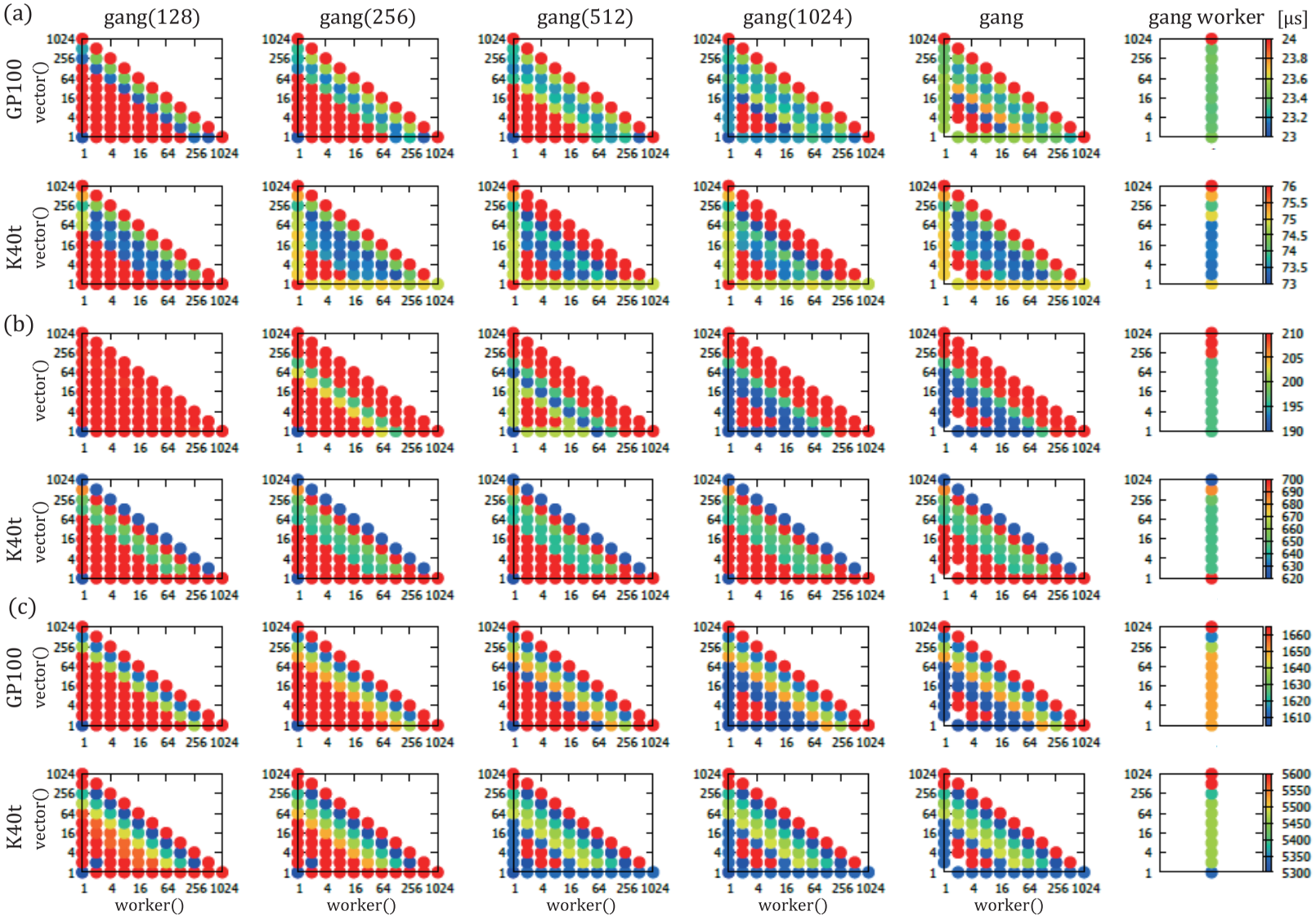

Figure 5.10: `gang`, `worker`, and `vector` values and average computation time for SpMV.

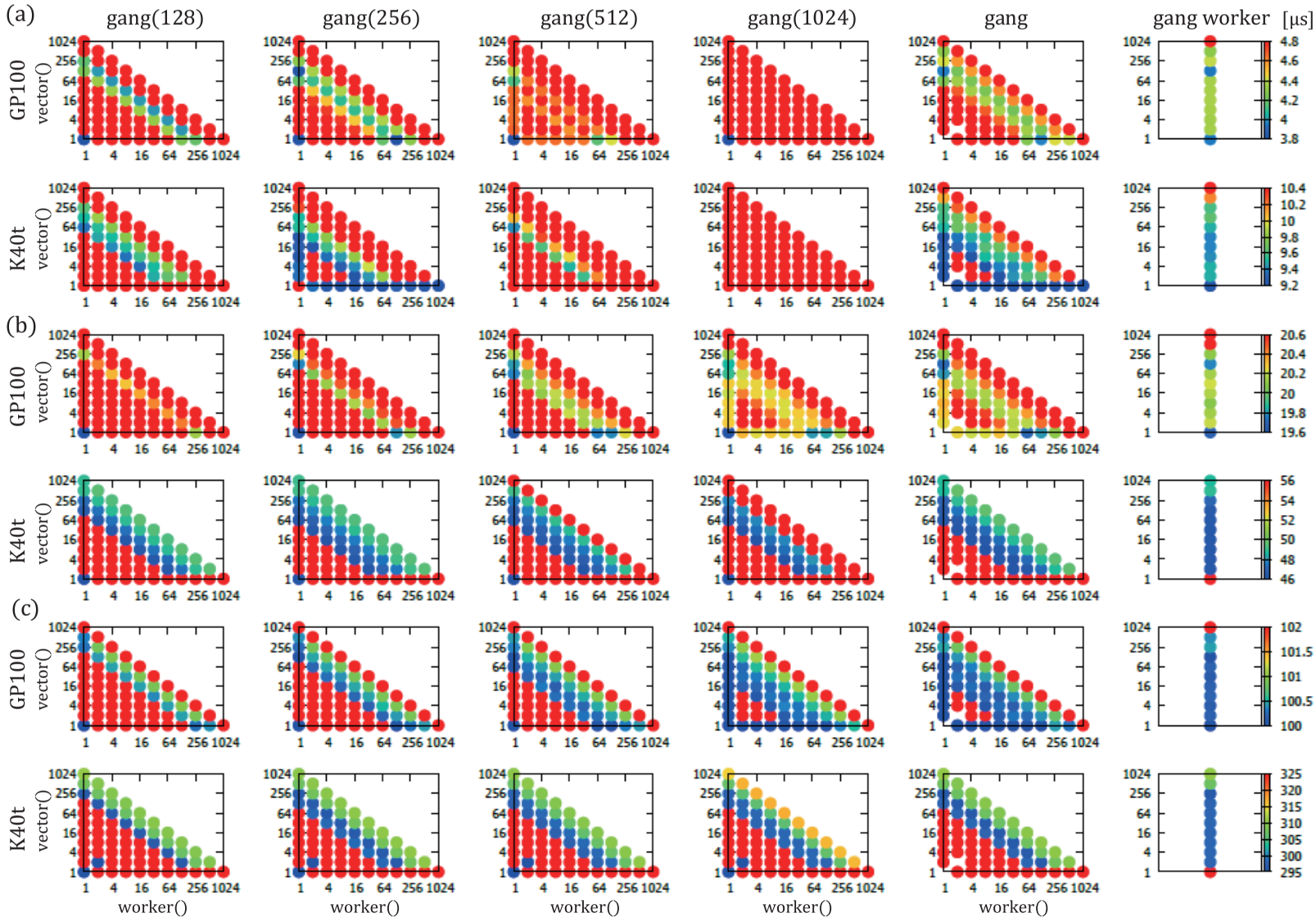

Figure 5.11: `gang`, `worker`, and `vector` values and average computation time for dot product.

Table 5.3: `gang`, `worker` and `vector` specified in each loop.

	GP100			K40t		
	gang	worker	vector	gang	worker	vector
Substitution	1024	2	16		2	64
SpMV		2	16		2	256
Dot product		1	64		2	64
Several AXPY/loop	512	1	64	512	1	64
1 AXPY/loop	512	1	64	1024	2	64
Creating $\tilde{r}$		2	16		16	8
Ordering input vectors	256	1	2	128	128	1
Ordering output vector			512		1	256
MILU(0)		16	1	128	8	16
Ordering A values	51 2	8	4	128	256	1
Ordering LU values		16	1	128	16	2
Preparing for ordering A	128	1	32	64	1	8
Preparing for ordering L	128	1	8	128	8	2
Preparing for ordering U	128	1	8	32	1	64
Preparing for ordering vectors	512	4	4	128	1	8

5.4 Performance Test

5.4.1 Test Problems and Computational Environment

We evaluated the performance of our implementation of the PBiCGSTAB solver combined with RMILU(0) preconditioner and BRB ordering on the GPU. As test problems, we used the coefficient matrix and right-hand side vector for the Poisson equation (2.1.14) used to obtain the potential ϕ within the 3D magnetron sputter simulation shown in Fig. 2.2. The shape and size of the computational domain to be simulated are shown in Fig. 2.4. For the discretization of the Poisson equation (2.1.14), a seven-point finite difference method with a uniform orthogonal grid was used. Dirichlet boundary conditions of -200 V for the target and 0 V for the substrate and sidewall were imposed on ϕ as shown in Fig. 2.2.

The number of unknowns is (a) $59 \times 59 \times 29 = 100,949$, (b) $119 \times 119 \times 59 = 835,499$, and (c) $239 \times 239 \times 119 = 6,797,399$ for grid sizes of 1, 0.5, and 0.25 mm, respectively. (a), (b), and (c) in the figures shown in this chapter represent these sizes of the test problems. The coefficient matrices and right-hand side vectors for the test problems were obtained from the 10,000th time step of the PIC simulation.

The RMILU(0) preconditioner, parallelized by BRB ordering, was optimized as described in Section 5.3.4. A relaxation parameter $\alpha = 0.95$ was used for the RMILU(0) factorization. The convergence criterion for the PBiCGSTAB method, unless otherwise specified, is that the relative residual norm $\|\boldsymbol{r}_k\|_2/\|A\boldsymbol{x}_0 - \boldsymbol{b}\|_2 < 10^{-8}$, where $\boldsymbol{r}_k$ and $\boldsymbol{x}_0$ are the residual vector at the k-th iteration and the initial estimated solution $\boldsymbol{x}_0 = \boldsymbol{0}$, respectively. After the convergence criterion using the working vector norm in line 18 or 26 of Algorithm 5 is satisfied and the iteration is stopped, convergence is confirmed by the true relative residual norm $\|A\boldsymbol{x}_k - \boldsymbol{b}\|_2/\|A\boldsymbol{x}_0 - \boldsymbol{b}\|_2$ calculated from the obtained solution $\boldsymbol{x}_k$. The program is implemented by Fortran90. Float-

Table 5.4: Environments used in the performance test.

	GDEP MAS-i7WF	SGI ICE XA
GPU	Quadro GP100 (Table 5.1, right)	Tesla K40t (Table 5.1, left)
CPU	Core i7 6800K	Xeon E5-2680v3 12C
CPU Speed	3.4 GHz	2.5 GHz
CPU cores/threads	6/12	12/24
Compiler	PGI Fortran 17.4	PGI Fortran 16.1
CUDA versiton	9.1	8.0

ing point data and operations were made to have double-precision, except for those marked as single-precision in the previous section.

The two different computing environments employed in the experiments are shown in Table 5.4. Data transfer between the CPU and the GPU was done via PCIe. The program was compiled using the -O3 optimization setting for both compilers.

5.4.2 Performance Test Results

5.4.2.1 Comparison with Naïve Implementation

We compare the performance of implementations, namely, the one before introducing the improvements shown in Sections 5.3.3 and 5.3.4 and the other after introducing them. The implementation before introducing them is called naïve, and after introducing them is called improved. Figs. 5.12 and 5.13 show the number of iterations and computation time, respectively, of the PBiCGSTAB iterative loop with the naïve implementation. The horizontal axis of the graph shows the number of blocks, and the right end of each graph is the number of nodes. Therefore, the rightmost point is the result of nodal RB ordering. Since the number of iterations required for convergence under the same conditions is the same for GP100 and K40t, Fig. 5.12 shows the number of iterations common to the two environments. The 'Elapsed time' in Fig. 5.12 is the time to compute the iterative loop of PBiCGSTAB. The data transfer time between the CPU and GPU is included, while the time taken for ordering and PMILU(0) factorization is not.

In BRB ordering, the number of same-color blocks, or one-half of the number of blocks, becomes the parallelism, so the computation time becomes smaller as the number of blocks increases until the number of blocks becomes large enough to use up the number of threads on the GPU. In a naïve implementation, the computation time is locally minimized when the number of blocks is close to the number of CUDA cores $\times 2$, due to the competition between the overhead of non-contiguous memory accesses caused by accessing elements stored farther away, which is greater with fewer blocks as described in the previous section, and the convergence acceleration effect of blocking, which is greater with fewer blocks. Overall, the execution time is minimized in the nodal RB with the largest number of blocks, which has the lowest overhead of non-contiguous accesses. Thus, the naïve implementation does not take advantage of the convergence acceleration effect of the blocking of BRB ordering.

By contrast, in the improved implementation, the overhead of non-contiguous access was eliminated by changing the storage format of the matrix. This minimized the execution time

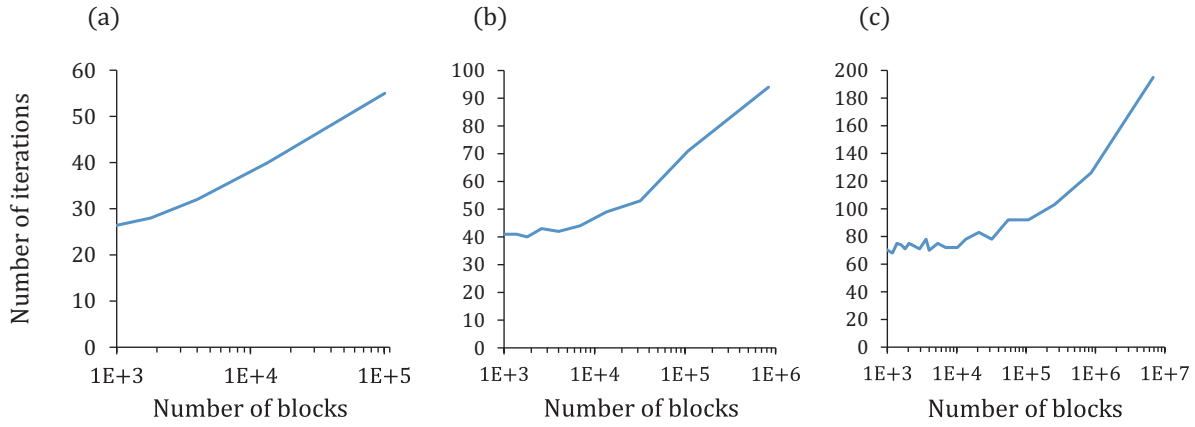

Figure 5.12: Number of blocks and the number of iterations required for PBiCGSTAB to converge.

in the region of smaller number of blocks, as shown in Fig. 5.8, and allowed us to fully exploit the convergence-accelerating effect of blocking.

Fig. 5.14 shows how much the modified implementation improves the computation time of each part of the PBiCGSTAB iterative loop compared to the naïve implementation. Here, the number of blocks is determined to be the number of CUDA cores $\times 2$, which is the local minimum value in Fig. 5.13. The computation time for forward and backward substitutions, which was long in naïve implementations, has been reduced by reducing the overhead of data access. The vector operations contained in SpMV and others had no data access problems even with a naïve implementation, but the change of the vectors $\hat{\boldsymbol{p}}$ and $\hat{\boldsymbol{s}}$ as input to SpMV to single-precision reduced the amount of data loading and calculation time in SpMV.

In the following sections, we use the block division condition that proved to be a good condition in Section 5.3.4.2 (Table 5.2). The peak performance ratio of the improved implementation using this block partitioning condition is described in Appendix B.

Even if the block partitioning is simply chosen, high performance can be obtained as shown in Fig. 5.15. For the conditions chosen here, the total number of blocks is the number of CUDA cores $\times 4$ and the number of blocks in each direction is close to $\sqrt[3]{\text{the total number of blocks}}$. The results shown in this figure, as well as the results in the following sections, include the computation time for ordering and PMILU(0) factorization, in addition to the iterative loop shown in Fig. 5.13.

5.4.2.2 Effect of Using Mixed-Precision

The difference between the convergence histories of the mixed-precision and double-precision solvers shown in Fig. 5.4 increases after the relative residual norm reaches 10^{-8}, which is used as our criterion. For comparison of the mixed-precision and double-precision solvers, two convergence criteria for the relative residual norm were used: less than 10^{-8}, which is shown in Section 5.4.1 as the common condition, and less than 10^{-13} as the high precision condition. The latter convergence condition was chosen because the relative residual norm when using only double-precision is no smaller than about 10^{-14}, as seen in Fig. 5.4.

The result is shown in Fig. 5.16. For problems other than the smallest size, the mixed-

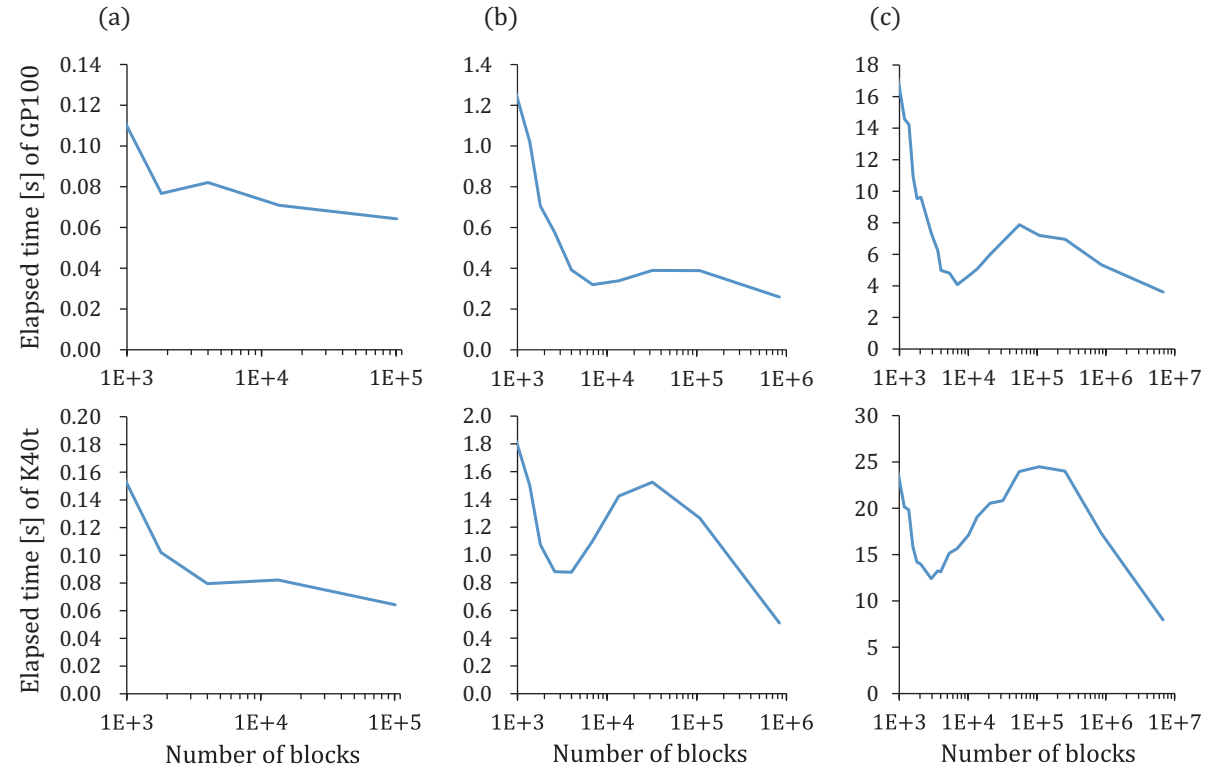

Figure 5.13: Number of blocks and computation time for PBiCGSTAB iterative loop (naïve implementation).

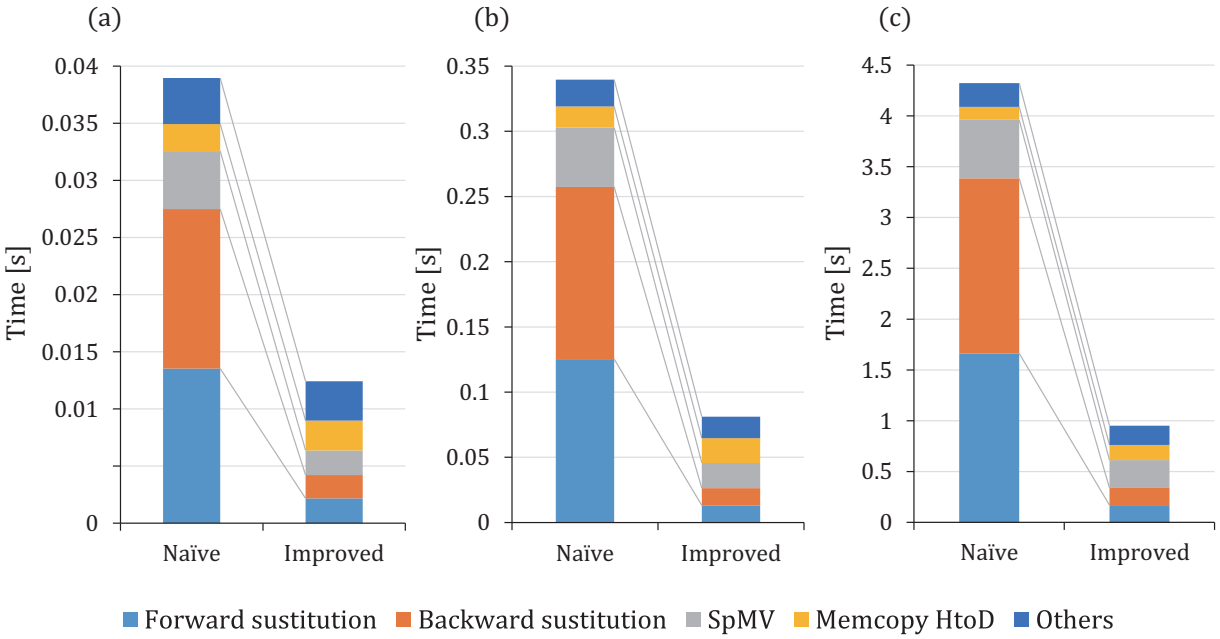

Figure 5.14: Comparison of naïve and improved implementations of CUDA Core $\times 2$ blocks profiling results on GP100.

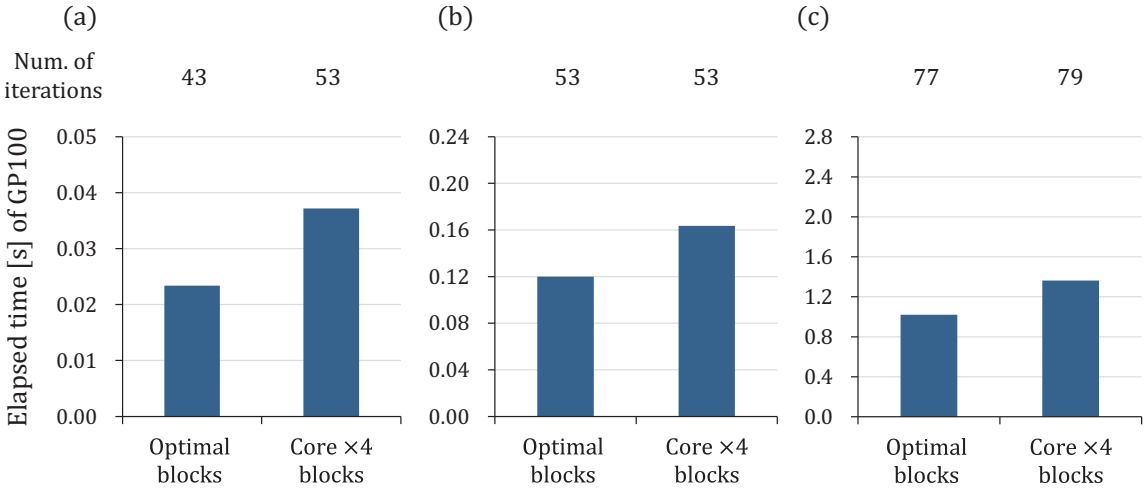

Figure 5.15: Computation time for simple block division condition.

precision method was faster. In particular, in the biggest problem, the overall computation time was reduced even though the number of iterations was increased by the use of mixed-precision. Similar results are shown for the case where the convergence criterion is 10^{-13}, suggesting that the mixed-precision technique is effective even when higher precision is required for larger problems.

5.4.2.3 Effect of Block Red-Black Ordering and Relaxed MILU(0) Preconditioner

In order to evaluate the effectiveness of BRB ordering and RMILU(0) preconditioner, we compare in Fig. 5.17 the performance of the four combinations. Here, ILU(0) factorization was used as a comparison to RMILU(0), and nodal RB ordering was used as a comparison to optimized BRB. The implementation shown in Section 5.3.4 has been adopted in all conditions. The computation time is affected by the number of iterations.

As shown in Fig. 5.5, the BRB requires fewer iterations to converge than the nodal RB, which corresponds to the right end of the graph. This is the same regardless of whether ILU(0) or RMILU(0) preconditioner is used. The BRB combined with RMILU(0) had fewer iterations to convergence than the combination with ILU(0) for all problem sizes. The advantage of the combination of RMILU(0) and BRB becomes more pronounced the larger the size of the problem and the larger the number of nodes contained in one block. For the largest size problem, RMILU(0) combined with nodal RB converged in fewer iterations than ILU(0) combined with it. However, RMILU(0) combined with nodal BR in the two smaller problems had the same number of iterations as ILU(0). In summary, we can say that RMILU(0) preconditioner has a higher convergence acceleration effect when combined with blocked RB.

5.4.2.4 Comparison with Existing Sparse Matrix Iterative Solver Libraries

There are libraries that include GPU-parallelized incomplete factorization and PBiCGSTAB routines. We compared the performance of our implementation with that of the PBiCGSTAB routines implemented in cuSPARSE [78], MAGMA [79], ViennaCL [80], and Ginkgo [81], which are the most widely used libraries.

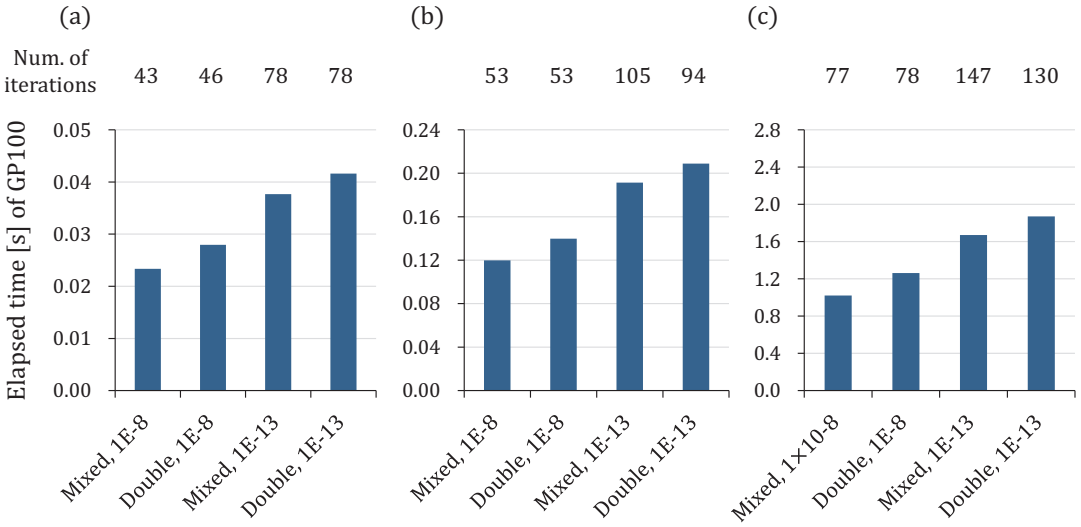

Figure 5.16: Comparison of computation time between double-precision and mixed-precision when using different precision convergence criteria.

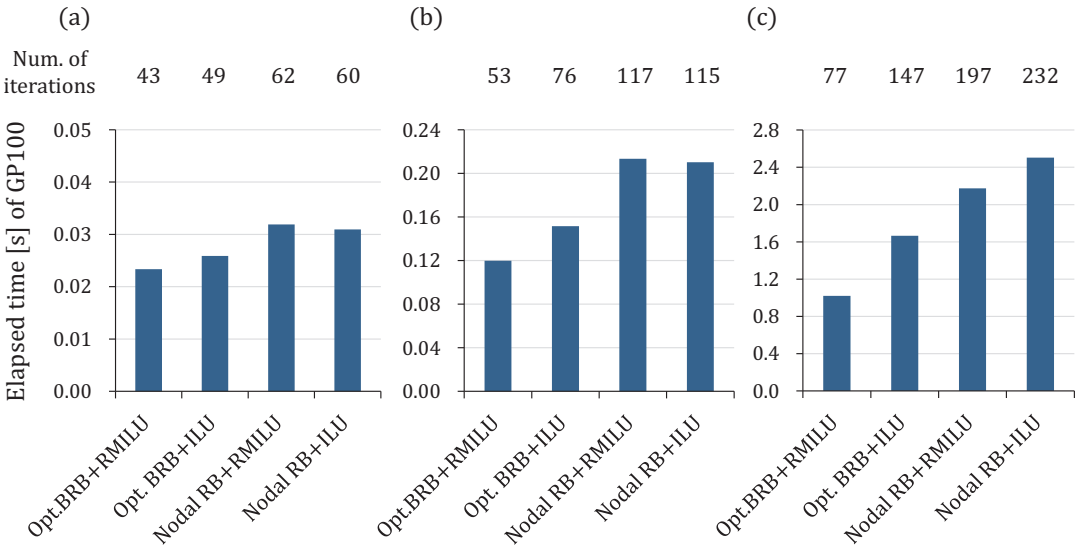

Figure 5.17: Comparison of computation time by combining ordering and preconditioner.

Table 5.5: Parallel preconditioning methods used in the comparison.

Library	ILU(0) / Triangular solver
cuSPARSE	Cohen-Castonguay's graph coloring algorithm
MAGMA	Level scheduling / ISAI
ViennaCL	Chow-Patel's fine-grained algorithm
Ginkgo	Chow-Patel's fine-grained algorithm / ISAI

Among the parallel preconditioning methods used in each library, the methods used for the comparison are shown in Table 5.5. These are the methods that yielded the fastest solution to our problem among the methods that use the LU factorization-based preconditioning employed by the respective libraries.

As input data, the coefficient matrix in CSR format, the right-hand side vector of the test problem, and the same initial estimated solution $\boldsymbol{x}_0 = \mathbf{0}$ as for the proposed method were given to these libraries. As input parameters for the library routines, convergence decision values equivalent to the convergence criteria of the proposed method shown in Section 5.4.1 were given. Furthermore, from the solutions obtained, it was checked whether the true residuals had been reduced to the specified standard. If non-default parameters are used with libraries, they are shown below. The features and parallelization methods of each library are as follows.

- **cuSPARSE**
  cuSPARSE is a sparse matrix and vector arithmetic library for CUDA. In addition to basic matrix-vector operations, cuSPASRE includes a triangular solver, which can be parallelized by level scheduling derived from the structure of the coefficient matrix and ordering routines based on Cohen-Castonguay's graph coloring [71, 91].
  cuSPARSE provides the functions used to compute each row in Algorithm 5, and the solver that calls these functions is used for comparison. The triangular solver routines exist in cuSPARSE, and mixed-precision can be implemented using these routines. However, cuSPARSE's triangular solver and other routines that target vectors require matching the precision of the input and output. Therefore, the input or output vectors that require double-precision elsewhere other in the algorithm need to be copied into single or double-precision arrays before or after calling the single-precision triangular solver. As a result, the mixing-precision was slightly faster for the K40t, but slower for the GP100. In this section, we show the results using double-precision for all arrays and calculations in cuSPARSE. Trianglar solver is parallelized with graph coloring. Since cuSPARSE is embedded in CUDA, the test code was implemented by Fortran as well as the proposed method and compiled by CUDA Fortran on each machine.
- **MAGMA**
  MAGMA is a library implemented primarily for CUDA, with APIs based on LAPACK and BLAS. Various preconditioner and iterative solution including PBiCGSTAB method routines are provided. As a triangular solver, a user can choose ILU(0) with either cuSPARSE level scheduling or incomplete sparse approximate inverse (ISAI) [74]. ISAI replaces the forward/backward substitution with the calculation of the product using the approximate inverse of L and U obtained from the ILU factorization. In this section, we show

the results of calculations using the double-precision PBiCGSTAB routine with ILU(0) and ISAI. MAGMA 2.3.0 was built using OpenBLAS 0.2.20 [92], and the test code was implemented in C and compiled using gcc 4.8.5.

- **ViennaCL**
  ViennaCL is a header library for C++ with interface to uBLAS included in Boost library [93]. In ViennaCL, PBiCGSTAB routines preconditioned by ILU are parallelized by using techniques such as level scheduling and blocking. Among the parallelization methods implemented in the library, Chow-Patel's fine-grained ILU(0) [94] is the fastest at solving our test problems. This algorithm considers the ILU factorization as the problem of computing the unknowns l_{ik} and u_{kj}. This problem is solved by fixed point iteration

$$\boldsymbol{x}^{(n+1)} = G(\boldsymbol{x}^{(n)}), \qquad (n = 1, 2, \cdots), \tag{5.4.1}$$

  where $\boldsymbol{x}$ is a vector consisting of l_{ik}'s and u_{kj}'s, and the elements of $\boldsymbol{x}^{(n+1)}$ can be computed in parallel. Similarly, by performing fixed point iteration as $G = I - D^{-1}K$, the triangular system $K\boldsymbol{x} = \boldsymbol{y}$ is solved, where D is a diagonal matrix with the same diagonal elements as K.
  In this section, we present computational results obtained from the double-precision PBiCGSTAB routine using Chow-Patel's fine-grained ILU(0). The test code was implemented with C++ and compiled by the CUDA compiler corresponding to the CUDA version installed in the respective test environment. The version used is ViennaCL 1.7.1 with Boost 1.68.0. Because the original PBiCGSTAB routine of ViennaCL uses preconditioned vectors such as $K^{-1}\boldsymbol{p}$ and $K^{-1}\boldsymbol{s}$ for convergence test, the convergence condition becomes stricter than the criterion used for the other libraries, while the number of iterations becomes larger. In order to determine convergence under the conditions shown in Section 5.4.1, two arrays holding the vectors $\boldsymbol{p}$ and $\boldsymbol{s}$ are copied and used for the determination. The operations required for this increased the execution time per iteration by about 0.8% over the original routine.
- **Ginkgo**
  Ginkgo is a newer library for linear systems than the others, and requires a C++ 11 compliant compiler to build. As a parallelized ILU factorization, Chow-Patel's fine-grained ILU(0), which is used in ViennaCL, is implemented. In addition, as a parallelized triangular solver, ISAI, which is used in MAGMA, is implemented.
  In this section, we present computational results for Ginkgo 1.1.1, obtained from Chow-Patel's fine-grained ILU(0) and the PBiCGSTAB routine using ISAI. The test code was implemented in C++, and gcc 6.3.1 was used to build Ginkgo and the test code.

Fig. 5.18 shows the execution time of the proposed method and the existing library routines for GPUs. The elapsed time in the figure is the time taken for the task performed each time the PIC simulation needs to calculate the potential. This includes the execution time for transfer of array data between the host and the device, reordering, ILU factorization and PBiCGSTAB.

The initialization time of the proposed method, which includes the execution time of `acc_init()` function and `data copyin` dummy integer directive, is 0.990 s and 0.275 s for GP100 and K40t, respectively. This time is especially large for GP100.

The time taken for tasks that are executed only once when the computational grid is invariant, such as the execution of the initialization function of the library and the tasks included in the initialization category of Section 5.3.4.1, is shown as the initialization time in Table 5.6.

Table 5.6: Initialization processing time [s].

			Opt. BRB	cuSPARSE	MAGMA	Ginkgo
GP100	(a)	$59 \times 59 \times 29$ grid	0.009	0.568	0.721	1.209
	(b)	$119 \times 119 \times 59$ grid	0.073	0.601	0.728	1.357
	(c)	$239 \times 239 \times 119$ grid	0.496	0.774	0.725	1.264
K40t	(a)	$59 \times 59 \times 29$ grid	0.016	0.705	0.049	
	(b)	$119 \times 119 \times 59$ grid	0.170	0.725	0.055	
	(c)	$239 \times 239 \times 119$ grid	1.351	0.972	0.055	

This initialization includes both tasks whose computational work depends on the size of the test problem and tasks with constant work. In ViennaCL, no explicit initialization is required.

According to the result of the proposed method, the difference between GP100 and K40t is small because the number of iterations is reduced due to the small optimal number of blocks for K40t, and the mixed accuracy is useful for the small memory bandwidth. The result of ViennaCL also shows a small difference, which is presumably due to the effect of the initialization time contributing to the computation time because of the absence of the initialization function.

The number of colors used for graph coloring in cuSPARSE was 33 for the (a) $59 \times 59 \times 29$ grid and (b) $119 \times 119 \times 59$ grid test problems, and 46 for the (c) $239 \times 239 \times 119$ grid test problem. Although cuSPARSE may have worse convergence due to coloring-based reordering, the number of iterations is smaller than that of MAGMA and Ginkgo using ISAI. In Ginkgo, the convergence behavior varies from run to run, and convergence may be judged even though the actual residuals have not decreased to the prespecified value. The results shown in the figure were obtained from a run that yielded a solution that converged to the prespecified value. In the runs where a solution was obtained, Ginkgo ran as fast as or faster than the other libraries.

The proposed method has the lowest number of iterations required for convergence for all test problems. Due to the convergence acceleration effect of the preconditioner and the optimization using the structure of the three-dimensional finite difference matrix based on the structural grid, the proposed method was able to obtain a fast solution. Especially when applied to the large test problems, the proposed method obtained solutions faster than cuSPARSE, MAGMA, ViennaCL or Ginkgo, even including device initialization. In the case of solving multiple simultaneous equations, the difference is even larger because the initialization time, which is performed only once, has a smaller impact.

The computation time when ILU(0) is used for preconditioner and nodal RB ordering is used for parallelization is shown in Fig. 5.17. In the results using the incomplete factorization and ordering, the number of iterations of our proposed routine is close to the number of iterations of other library routines, while its computation time is shorter than that of using the library routine. This suggests that even solvers implemented in OpenACC, which is generally considered to have a speed disadvantage, can achieve the same or better performance as those implemented in CUDA or OpenCL, depending on the method.

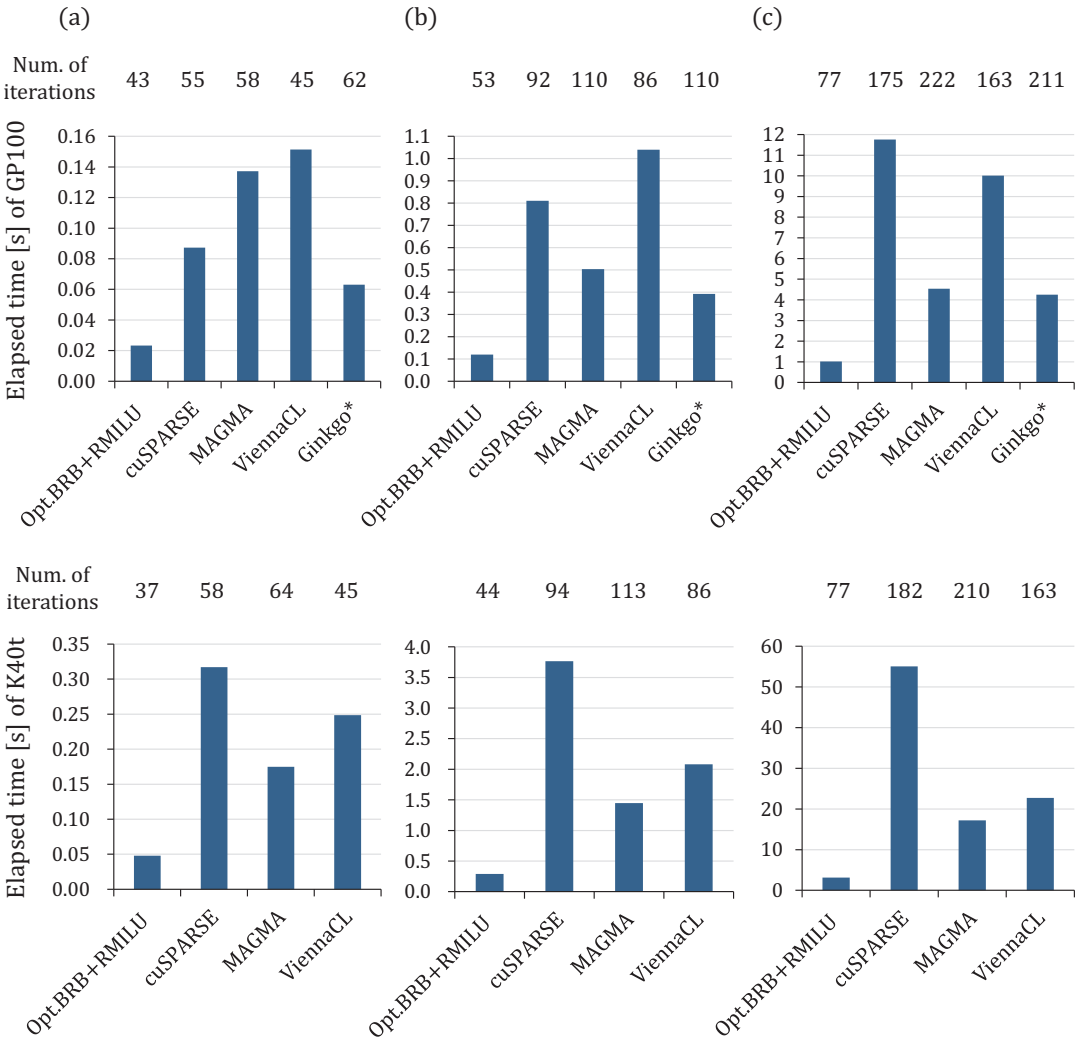

Figure 5.18: Comparison of the execution time of the proposed method with cuSPARSE, MAGMA, ViennaCL, and Ginkgo. *Results of Ginkgo are derived only from calculations that actually converged.

5.4.2.5 Performance of solving BRB-ordered systems with existing library routines

Some of the existing libraries verified in this study are equipped with a parallelization module based on level scheduling that takes advantage of the parallelism inherent in the structure of the coefficient matrix. Therefore, we evaluated the performance when the input to these libraries is a coefficient matrix with a highly parallel matrix structure by BRB ordering. To achieve coalesced memory access during parallel computation execution, the matrix elements were stored in an array in the order shown in Fig. 5.3. In its storage format, the storage position is determined by the number of red blocks n_{rb}, black blocks n_{bb}, and red nodes n_{rn}. The j-th element belonging to the i-th red block is stored at the position $i+(j-1)n_{rb}$. Whereas, the j-th element belonging to the i-th black block is at the position $i+(j-1)n_{bb}+n_{rn}$. Fig. 5.19 shows the computation time of parallelization using the level scheduling routines of each library on GP100 with the input being the coefficient matrix and right-hand side vector ordered by BRB ordering. Here, the 'Precondsetup' shown in the graph is the analysis of the matrix structure for level scheduling and the computation time for ILU factorization. Note that the results of Fig. 5.19 do not include the data copy and reordering times that were included in the results of Fig. 5.18. The results of cuSPARSE were equal to or better than those obtained when the parallelism was extracted by the graph coloring routines included in the library. The results for MAGMA and ViennaCL showed that the setup time was larger than the time required for the iterations. The setup time for the level scheduling of each library was very large compared to the setup time of ISAI in MAGMA, which was 0.107 s, 0.299 s, and 1.897 s, and the setup time of Chow-Patel's fine-grained ILU(0) in ViennaCL, which was 0.073 s, 0.542 s, and 4.385 s, in ascending order of the problem size. Therefore, to make effective use of BRB ordering in MAGMA or ViennaCL, it is necessary to devise a way to reduce the large setup time. One way to do this would be to add routines to assist in level scheduling based on the matrix structure.

5.4.2.6 Performance of PIC Simulation with Proposed Method Incorporated

We evaluated the speed-up effect of the proposed method when incorporated into the PIC-based plasma simulation described in Section 2.1. The original code used in the preliminary evaluation shown in Section 2.2 is an MPI parallel code for super particles using the particle decomposition method. In addition to the incomplete factorization and iterative solver proposed in this chapter, the charge density smoothing, and matrices and right-hand side vectors generation included in the 'potential calculation' routine, and the electric field calculation included in the 'electric field calculation' routine were GPU-parallelized. The processing in the initialization category of Section 5.3.4.1 is performed only once at the start of PIC simulation, and the processing in the iterative solver category of the same section is repeated each time a potential is needed at each step.

Compilation and execution was done using OpenMPI [95] on the GDEP MAS-i7WF with Quadro GP100 shown in Section 5.4.1. Fig. 5.20 shows the computation time of each subroutine for a test problem with a problem size of (a) $59 \times 59 \times 29$ and the same conditions as the test problem in this section, computed up to 10000 steps with MPI parallelism of 6. GPU parallelization requires additional time to transfer data, but the reduction in computation time due to faster computation outweighs this, resulting in shorter computation times for GPU-parallelized subroutines. The charge density calculation included in the potential calculation routine is not entirely GPU-parallelized because it includes the time for parallel computation

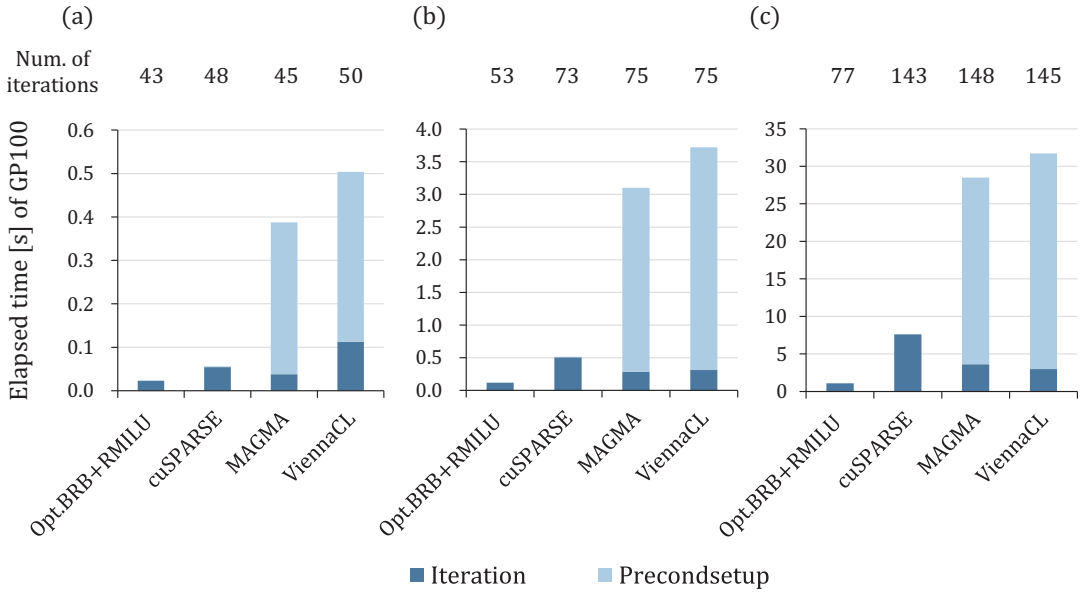

Figure 5.19: Comparison of parallel computational performance using library routines for systems ordered by BRB ordering.

by the particle decomposition method and process synchronization on the CPU. However, the ratio of the potential calculation routine to the total calculation time became small, and it is confirmed that the overall calculation time can be shortened by using the GPU parallelization of the proposed method.

5.5 Conclusion

In this chapter, we parallelize the solution of a linear system with the coefficient matrix preconditioned by RMILU(0) factorization and BRB ordering. The method is optimized for parallel computing using GPUs.

BRB ordering generally requires fewer iterations than nodal RB sequencing to obtain a convergent solution. However, in the forward/backward substitution of BRB ordered coefficient matrices, if the matrix elements are stored in the row or column direction, there is a problem of adjacent threads accessing the data at distant locations and causing overhead due to data reloading. To solve this problem, we changed the order of storing element data to achieve coalesced access. Furthermore, to take advantage of GPU's fast single-precision operations, and decrease the load time, the operations and the data for the forward/backward substitutions were made single-precision. For optimization of parameters, the block partitioning condition and the parallel granularity size of OpnenACC were examined. In general, there is a trade-off between increasing the parallelism by increasing the number of the same-colored blocks and the convergence speed. On the other hand, the parallelism must be sufficiently large to allocate tasks to the many threads of GPUs. In order to achieve these balances, the condition was set so that the number of blocks per SM is a multiple of 32 over 32×6, and that the number of nodes contained in each block in each axial direction is equal. The optimal values of the parameters for each of the parallel granularity `gang`, `worker`, and `vector` in the OpenACC directive depend

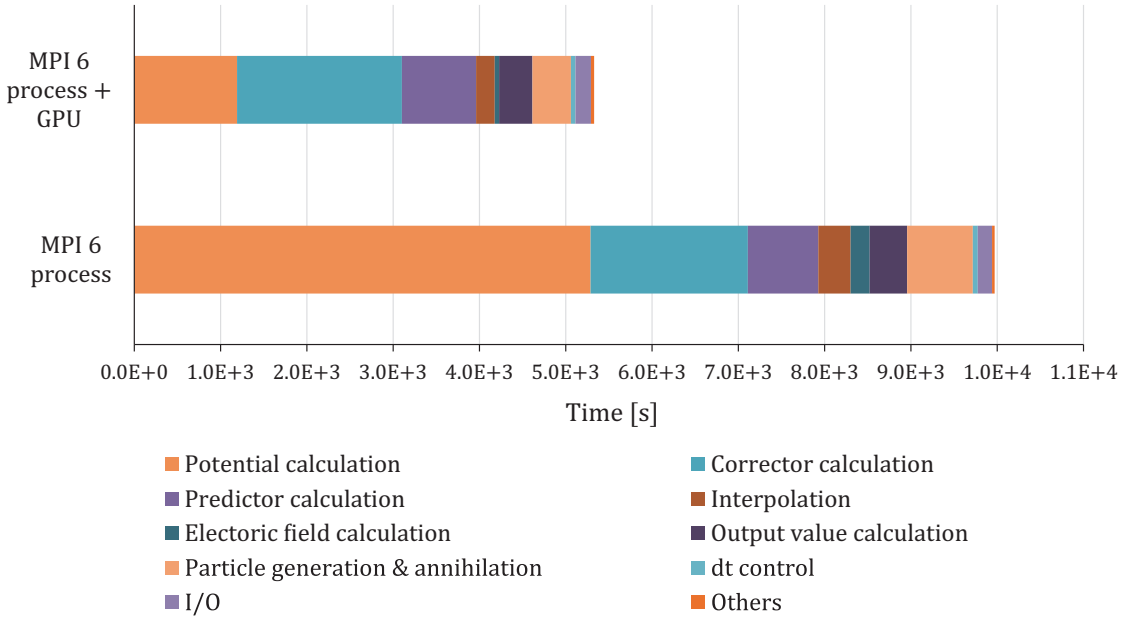

Figure 5.20: Comparison of computation time for each subroutine of PIC method simulation with and without GPU parallelization.

on the GPU specification, what processing is done in the loop, and the size of the array to be processed. Here we set the parameters that showed good results for all sizes of the test problem to the loops processed by the respective GPUs.

These implementation methods and optimizations resulted in a significant increase in computational speed compared to previous implementations, delivering solutions faster than the leading iterative solver libraries for GPUs.

Chapter 6

Convergence analysis of BRB ordered PMILU(0)/PMIC(0) preconditioning

6.1 Introduction

We consider simultaneous linear equations $A\boldsymbol{x} = \boldsymbol{b}$ obtained by applying a two-dimensional 5-point difference finite formula to the two-dimensional Poisson equation as a model problem. As described in Chapter 3, the condition number of the unpreconditioned coefficient matrix A is known to be $O(h^{-2})$, where h is the grid size. By contrast, it is known that the condition number of A preconditioned using PMILU(0)/PMIC(0) methods with natural ordering is $O(h^{-1})$, which is one rationale for the effectiveness of this preconditioner. Because of its sequential nature, several methods have been proposed to parallelize the ILU(0)/IC(0) preconditioner through a reordering. This includes BRB ordering, which is empirically known to realize a convergence acceleration comparable to sequential PMILU(0)/PMIC(0) factorization. Therefore, in this chapter, we investigate the reason for its high convergence acceleration effect by analyzing the properties of the matrices obtained by applying BRB ordering and a PMILU(0)/PMIC(0) preconditioner to the model problem.

In the model problem applied here, we consider a square region $\Omega = [0,1]^2$ and the Poisson equations

$$-\Delta u(x,y) = f(x,y) \quad \text{in } \Omega, \tag{6.1.1}$$

$$u(x,y) = 0 \quad \text{on } \partial\Omega. \tag{6.1.2}$$

Each of the x, y directions is divided into $N+1$ intervals, and the equation at grid point (i,j) is discretized as

$$-u_{i,j-1} - u_{i-1,j} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = h^2 f_{i,j} \qquad (i,j = 1,2,...,N), \tag{6.1.3}$$

where h is the grid width. Here, the diagonal elements of coefficient matrix A are 4, while all the off-diagonal elements representing the relationship with the neighboring nodes have values of -1, and there are at most four off-diagonal elements per row.

This model problem has been used to evaluate the convergence acceleration effects of the PMILU/PMIC preconditioner with natural ordering [3, 4, 5, 42]. In this chapter, we also use this problem to conduct an empirical analysis of the effects of BRB ordering.

6.2 Order of Condition Numbers of PMILU(0)/PMIC(0) Preconditioned Matrix

In general, the performance of Krylov subspace method solvers is improved when combined with an ILU(0)/IC(0) preconditioner. However, when two-dimensional elliptic partial differential equations are discretized using a finite difference or finite element method, the condition number $\kappa(K^{-1}A)$ of preconditioned matrix $K^{-1}A$ is $O(h^{-2})$, which is the same as $\kappa(A)$. This problem can be remedied using PMILU(0), which perturbs the diagonal elements in advance and compensates the diagonal elements for the fill-in values dropped during an incomplete factorization.

The PMILU(0) factorization algorithm is shown in Algorithm 8. In this chapter, the diagonal element a_{ii} updated through factorization is represented by $\tilde{a}_{ii}$ to distinguish it from the original diagonal elements. Consequently, line 5 in Algorithm 8 is rewritten as $a_{ik} = a_{ik}/\tilde{a}_{kk}$, and line 14 is rewritten as $\tilde{a}_{ii} = a_{ii} - s$.

In this factorization, the element a_{ij} is updated if two conditions are satisfied. As one condition, $a_{ij} \neq 0$, and as the other, there exists $k < i, j$ such that $a_{ik} \neq 0$ and $a_{kj} \neq 0$. For a matrix A obtained from a 2D 5-point stencil, these conditions are satisfied when all three nodes i, j, and k are adjacent to each other or $i = j$.

In a two-dimensional square grid, there are no combinations of nodes that satisfy the former condition, and thus the non-diagonal element a_{ij}, where $i \neq j$, is not updated and becomes

$$\tilde{a}_{ij} = a_{ij} \quad (i \neq j). \tag{6.2.4}$$

That is, in PMILU(0) factorization, only the diagonal elements for which the latter condition is satisfied are updated. Note that, even when ordering is applied, only the diagonal element is updated because the conditions in which i is adjacent to j and k, and j is adjacent to i and k, cannot be satisfied in a two-dimensional square grid (see Lemma 1).

The PMILU(0) factorization can also be regarded as the MILU(0) factorization for the matrix $A + E$, where $E = \zeta h^2 \mathrm{diag}(A)$. In addition, the fill-in $\hat{r}$, which is dropped when updating the (i, j) element, is $a_{ik} = a_{ik}/\tilde{a}_{kk}$ from line 5 of Algorithm 8, and thus from line 8 becomes

$$\hat{r}_{ij} = \frac{a_{ik} a_{kj}}{\tilde{a}_{kk}}. \tag{6.2.5}$$

Dropping this fill-in is equivalent to conducting an exact LU factorization of the matrix with $\hat{r}$ added to the (i, j) element of $A + E$. In addition, subtracting the fill-in for the same row element from a_{ii} is the equivalent to applying an exact LU factorization for a matrix with

$$\hat{r}_{ii} = -\sum_{j \neq i} \hat{r}_{ij} \tag{6.2.6}$$

added to the (i, j) element of $A + E$.

Define a matrix $\hat{R} = (\hat{r}_{ij})$, where $\hat{r}_{ij}$ is defined by Eqs. (6.2.5) and (6.2.6) for elements with a fill-in and diagonal elements, respectively, and $\hat{r}_{ij} = 0$ otherwise. Then, the PMILU(0)

factorization is the computation of an upper triangular matrix U and a lower triangular matrix L, whose diagonal elements have a value of 1 satisfying

$$A + E + \hat{R} = LU. \tag{6.2.7}$$

Here, U has the same sparse pattern as the upper triangular part of A, the diagonal element of which is $\tilde{a}_{ii}$, and the non-diagonal element is a_{ij}, which is the same as that of A.

The coefficient matrix A of the model problem (6.1.3) is a symmetric matrix. When A is a symmetric matrix, the ILU factorization becomes an IC factorization. In this case, from $l_{ik} = a_{ik}/\tilde{a}_{kk} = a_{ki}/\tilde{a}_{kk}$, we obtain

$$L = (DU)^\top, \tag{6.2.8}$$

where $D = \mathrm{diag}(1/\tilde{a}_{11}, 1/\tilde{a}_{22}, \ldots, 1/\tilde{a}_{nn})$. Eq. (6.2.7) is thus rewritten as

$$A + E + \hat{R} = U^\top DU, \tag{6.2.9}$$

where D is a diagonal matrix with positive diagonal elements, and $\hat{R}$ is a symmetric matrix. Consequently, the eigenvalues of $\hat{R}$ are real, and from Eq. (6.2.6) and $\hat{r}_{ij} \geq 0 (i \neq j)$,

$$|\hat{r}_{ii}| = \sum_{j \neq i} |\hat{r}_{ij}| \tag{6.2.10}$$

holds for any row i of $\hat{R}$. Here, $\hat{r}_{ii} \leq 0$, and thus $\hat{R}$ is a semi-negative definite matrix. In addition, if we define $K = U^\top DU$ and $R = E + \hat{R}$, we have

$$A = K - R. \tag{6.2.11}$$

The following theorem holds for the order of the eigenvalues.

Theorem 5. (Theorem 2.1 of Gustafsson [3]) *Assume A to be a matrix given by the model problem* (6.1.3). *In addition, $R = \hat{R} + E$, $A = K - R$, assuming $\hat{R}$ as a semi-negative definite matrix, and E as a diagonal matrix of diagonal element size $O(h^{-2})$. Further, assume that there is a constant c that is independent of h, and for any vector $\boldsymbol{v} \in \mathbb{R}^n$,*

$$0 \leq -(\hat{R}\boldsymbol{v}, \boldsymbol{v}) \leq \frac{1}{1+ch}(A\boldsymbol{v}, \boldsymbol{v}). \tag{6.2.12}$$

Then,

$$\frac{\lambda_{\max}(K^{-1}A)}{\lambda_{\min}(K^{-1}A)} = O(h^{-1}). \tag{6.2.13}$$

Based on a theoretical discussion of this theorem and experiments, it was observed that the condition number $O(h^{-1})$ is obtained for the model problem (6.1.3) if it is preconditioned through PMIC(0) factorization with natural ordering.

6.3 Experimental Analysis

6.3.1 Effect of Perturbation Factor

In PMILU(0)/PMIC(0), there is an option for the value of the perturbation factor ζ, which determines how much the diagonal elements are perturbed. For Poisson equations, it has been

pointed out that the convergence conditions are insensitive to this parameter [5]. By contrast, a good estimate $\zeta \approx 2\pi^2$ has been obtained by extending the condition with the smallest condition number obtained through Fourier analysis in the case of periodic boundary conditions to include Dirichlet conditions [96]. These analyses have been applied for natural ordering. Therefore, in this section, we explore the relationship between BRB ordering and the perturbation factor.

From the example obtained from the model problem (6.1.3) of unknown number $n = 1024$ for 32×32 grid points ($N = 32$), we evaluated the spectral condition number $\kappa(K^{-1}A) = \lambda_{\max}/\lambda_{\min}$, the Frobenius norm $\|R\|_F$ [53] of the remainder matrix $R = A - LU$, and the iteration number of the CG method. The eigenvalues were obtained using the `eigs` function in MATLAB. The algorithm of the CG method is shown in Algorithm 3. For the number of iterations, we used the right-hand vector obtained from the solution $u(x, y) = x(x - 1)y(y - 1)e^{xy}$, and obtained the number of iterations needed for the relative error norm to reach a value of less than 10^{-8}. Herein, we also show the result of ILU(0), which does not compensate dropped elements to diagonal elements.

The minimum eigenvalues are shown in Table 6.1, the maximum eigenvalues are shown in Table 6.2, and the condition numbers are shown in Table 6.3. In the natural ordering results, the value of ζ achieving the smallest condition number is lower than π^2, as shown in [96]. When $\zeta > 0$, the smaller the perturbation, the closer the minimum eigenvalue is to 1, and the larger the perturbation, the smaller the minimum and maximum eigenvalues. Furthermore, the smaller the perturbation, the larger the increase in the maximum eigenvalue owing to the increase in the number of blocks, and the condition number rapidly deteriorates. This is not as pronounced in the unperturbed MILU(0) case with $\zeta = 0$. Therefore, when the BRB ordered block number is larger, a larger ζ has the best condition, and in the case of node RB with equal block and node numbers, the ILU(0) condition number is smaller than the best value of ζ for PMILU(0).

The numbers of iterations are shown in Table 6.4. Overall, the number of iterations tends to be small when the condition number is small. However, a rapid increase in the number of iterations owing to the zero pivot when $\zeta = 0$ does not appear in the condition number. Conversely, a rapid deterioration in the condition number when $\zeta > 0$ and ζ is small is not that remarkable in terms of the number of iterations. In addition, the phenomenon in which the number of iterations becomes smaller under conditions with a large number of blocks and $\zeta \geq 100\pi^2$ cannot be predicted from the condition number.

The Frobenius norms $\|R\|_F$ of R are shown in Table 6.4. Here, ζ for which $\|R\|_F$ is small, tends to be larger than ζ for which the condition number becomes smaller. Moreover, a rapid deterioration in the condition number when $\zeta > 0$ and ζ is small is not reflected in $\|R\|_F$. By contrast, the phenomenon by which the number of iterations becomes smaller under conditions with a large number of blocks and $\zeta \geq 100\pi^2$ can be predicted from the reduction of $\|R\|_F$.

Table 6.1: Perturbation factor ζ and minimum eigenvalues $\lambda_{\min}$ for block partition $N_B \times N_B$.

ζ	0	0.01	$0.25\pi^2$	$0.5\pi^2$	π^2	$2\pi^2$	$5\pi^2$	$10\pi^2$	$15\pi^2$	$20\pi^2$	$25\pi^2$	$40\pi^2$	$50\pi^2$	$100\pi^2$	$200\pi^2$	ILU(0)
N_B:1	0.330	1.000	0.803	0.592	0.375	0.214	0.093	0.048	0.033	0.025	0.020	0.012	0.010	0.005	0.002	0.030
N_B:2	0.323	1.000	0.798	0.590	0.374	0.214	0.093	0.048	0.033	0.025	0.020	0.012	0.010	0.005	0.002	0.029
N_B:4	0.316	1.000	0.801	0.595	0.377	0.215	0.094	0.048	0.033	0.025	0.020	0.012	0.010	0.005	0.002	0.027
N_B:8	0.304	1.000	0.807	0.604	0.384	0.218	0.094	0.048	0.033	0.025	0.020	0.012	0.010	0.005	0.002	0.025
N_B:16	0.284	1.000	0.825	0.628	0.398	0.224	0.095	0.049	0.033	0.025	0.020	0.012	0.010	0.005	0.002	0.020
N_B:32	0.231	1.000	0.889	0.719	0.445	0.238	0.098	0.049	0.033	0.025	0.020	0.012	0.010	0.005	0.002	0.012

Table 6.2: Perturbation factor ζ and maximum eigenvalues $\lambda_{\max}$ for block partition $N_B \times N_B$.

ζ	0	0.01	$0.25\pi^2$	$0.5\pi^2$	π^2	$2\pi^2$	$5\pi^2$	$10\pi^2$	$15\pi^2$	$20\pi^2$	$25\pi^2$	$40\pi^2$	$50\pi^2$	$100\pi^2$	$200\pi^2$	ILU(0)
N_B:1	3.48	9.60	6.12	4.87	3.73	2.80	1.91	1.44	1.24	1.12	1.03	0.89	0.83	0.69	0.52	1.20
N_B:2	5.78	24.59	12.90	9.43	6.62	4.57	2.81	1.97	1.61	1.40	1.25	1.00	0.90	0.69	0.52	1.24
N_B:4	6.36	149.2	20.72	12.45	7.60	4.80	2.82	1.97	1.61	1.40	1.25	1.00	0.90	0.69	0.52	1.24
N_B:8	9.93	1.2E4	50.17	25.58	13.24	7.04	3.27	1.97	1.63	1.40	1.25	1.00	0.90	0.69	0.52	1.25
N_B:16	18.11	2.7E4	109.7	55.05	27.70	14.03	5.81	3.05	2.12	1.64	1.37	1.06	0.94	0.69	0.52	1.26
N_B:32	26.67	5.4E4	220.9	110.6	55.42	27.83	11.28	5.76	3.91	2.99	2.43	1.59	1.31	0.72	0.52	1.33

Table 6.3: Perturbation factor ζ and spectrum condition numbers $\kappa(K^{-1}A)$ for block partition $N_B \times N_B$.

ζ	0	0.01	$0.25\pi^2$	$0.5\pi^2$	π^2	$2\pi^2$	$5\pi^2$	$10\pi^2$	$15\pi^2$	$20\pi^2$	$25\pi^2$	$40\pi^2$	$50\pi^2$	$100\pi^2$	$200\pi^2$	ILU(0)
N_B:1	10.5	9.6	7.6	8.2	9.9	13.1	20.4	30	38.1	45.5	52.5	71.8	83.9	138.3	210.3	39.8
N_B:2	17.9	24.6	16.2	16	17.7	21.4	30.1	40.9	49.5	57	63.6	80.7	90.5	138.3	210.3	42.9
N_B:4	20.1	149.3	25.9	20.9	20.1	22.3	30.1	40.8	49.5	56.9	63.6	80.6	90.5	138.3	210.3	45.1
N_B:8	32.6	1.1E4	62.2	42.3	34.5	32.3	34.7	40.8	49.9	57.2	63.7	80.8	90.5	138.3	210.3	50.4
N_B:16	63.8	3.0E4	133	87.6	69.5	62.7	60.8	62.5	64.6	66.6	69.1	85.8	94.5	138.2	210.2	64.6
N_B:32	115.4	5.4E4	248.6	153.9	124.7	117.1	115.1	116.5	118.5	120.4	122.4	128	131.4	145.3	210.2	110.7

Table 6.4: Perturbation factor ζ and iteration number for block partition $N_B \times N_B$.

ζ	0	0.01	$0.25\pi^2$	$0.5\pi^2$	π^2	$2\pi^2$	$5\pi^2$	$10\pi^2$	$15\pi^2$	$20\pi^2$	$25\pi^2$	$40\pi^2$	$50\pi^2$	$100\pi^2$	$200\pi^2$	ILU(0)
N_B:1	25	25	23	22	22	23	27	31	34	37	39	45	48	61	74	35
N_B:2	30	30	28	26	27	27	32	35	37	39	40	46	49	61	74	35
N_B:4	46	45	36	32	31	31	33	35	38	39	41	46	49	61	74	36
N_B:8	738	128	55	46	41	37	35	37	39	40	42	47	49	61	74	38
N_B:16	1412	255	95	76	62	53	46	44	44	43	44	48	50	61	74	42
N_B:32	-	409	130	100	80	67	56	52	53	53	53	53	54	57	68	49

Table 6.5: Perturbation factor ζ and remainder matrix norm $\|R\|_F$ for block partition $N_B \times N_B$.

ζ	0	0.01	$0.25\pi^2$	$0.5\pi^2$	π^2	$2\pi^2$	$5\pi^2$	$10\pi^2$	$15\pi^2$	$20\pi^2$	$25\pi^2$	$40\pi^2$	$50\pi^2$	$100\pi^2$	$200\pi^2$	ILU(0)
N_B:1	34.2	34.2	33	32.2	30.7	28.3	22.9	16.6	13.4	14.1	17.9	34.8	47	107.7	226.5	12.8
N_B:2	34.4	34.4	33.4	32.6	31.2	28.9	23.5	17.1	13.7	14.2	17.8	34.5	46.7	107.4	226.4	12.9
N_B:4	35.5	35.4	34.6	33.8	32.4	30.2	24.8	18.4	14.7	14.6	17.7	33.9	46	106.9	226	13.2
N_B:8	37.5	37.5	36.9	36.3	35.2	33.2	28	21.5	17.2	15.9	17.8	32.7	44.6	105.6	225	14.2
N_B:16	43.3	43.3	42.9	42.5	41.7	40.1	35.6	29.1	24.1	20.9	20.3	30.5	41.6	102.3	222.7	17.2
N_B:32	-	68.9	68.6	68.2	67.5	66.2	62.3	56.3	51	46.6	43.2	40.5	44.9	96.3	216.8	24.5

6.3.2 Diagonal Element Size

As is shown by Eq. (6.2.4), for our model problem (6.1.3) the off-diagonal elements of the PMIC(0) factorization are not updated from -1 of the original coefficient matrix. Hence, Eq. (6.2.5) becomes

$$\hat{r}_{ij} = \frac{1}{\tilde{a}_{kk}}, \tag{6.3.14}$$

and thus the element at the position corresponding to the fill-in of remainder matrix R is the reciprocal of the diagonal element $\tilde{a}_{kk}$. It is necessary for the element at the position corresponding to the fill-in of R to be less than or equal to $(1+ch)^{-1}$, as a sufficient condition for the condition number of a matrix preprocessed based on Theorem 5 to be $O(h^{-1})$. Under natural ordering, this condition holds. To examine the degree to which this condition is satisfied in cases of BRB ordering, in this section, we obtain $\tilde{a}_{ii}$ and compute the following:

$$c_i = \frac{\tilde{a}_{ii} - [4 - (\text{Number of fill-ins})]}{h}. \tag{6.3.15}$$

The values of $\tilde{a}_{ii}$ and c_i at node i are shown in Figs. 6.1 and 6.2. Here, 64×64 ($N = 64$), 32×32 ($N = 32$), and 16×16 ($N = 16$) grid points are used. In PMILU(0), the natural ordering yields c_i, independent of the grid size. In other words, the sufficient condition for the condition number to be $O(h^{-1})$ is satisfied based on the theory. In the block division of 2×2 and 4×4, c_i corresponding to the nodes located at the left and bottom ends of each red block that are not at the periphery of the computational domain, shown in Fig. 6.1, is close to zero. For the other nodes, however, the value of c_i is close to the value under natural ordering.

In the diagonal element $\tilde{a}_{ii}$ corresponding to the node in the red block, the value of the row corresponding to the lower-left node in the block in Fig. 6.1 is first calculated. The computed value is then used to update the node values connected to the right and above the node. Because of this ordered update, $\tilde{a}_{ii}$ is symmetric about the diagonal from the lower-left to the upper-right of the computational domain.

The value of $\tilde{a}_{ii}$ for this diagonal node with a small number of blocks is shown in Fig. 6.3. The horizontal axis of this graph is the diagonal node index, where the index of the diagonal node to the lower-left of the computational domain is 1 and increases toward the upper-right. In the graph, the results with a larger number of blocks are displayed in the foreground, and thus only the points with the larger number of blocks are visible when the values are equal. As shown by the 'Block pattern' in Fig. 6.1, all diagonal nodes are those on the diagonal of the red block. Except for a few nodes from the bottom-left of each red block, the values are close to those of the natural ordering with a 1×1 block. The values of $\tilde{a}_{ii}$ in the red block near the left or bottom edge, except for cases in which these edges are part of the boundary, show the same pattern independently of the number of block divisions, and are the same if the relative positions from the bottom-left of the block are the same. As a result, the smaller the number of blocks, the larger the percentage of $\tilde{a}_{ii}$ that is close to the value of the natural ordering.

This similarity to natural ordering may be the reason why convergence close to natural ordering is achieved when the number of block divisions is small. However, as the number of divisions increases, the proportion of small $\tilde{a}_{ii}$ increases, and c_i becomes close to 0 within most of the area. This is considered the cause of deteriorating the convergence when the number of block divisions is large.

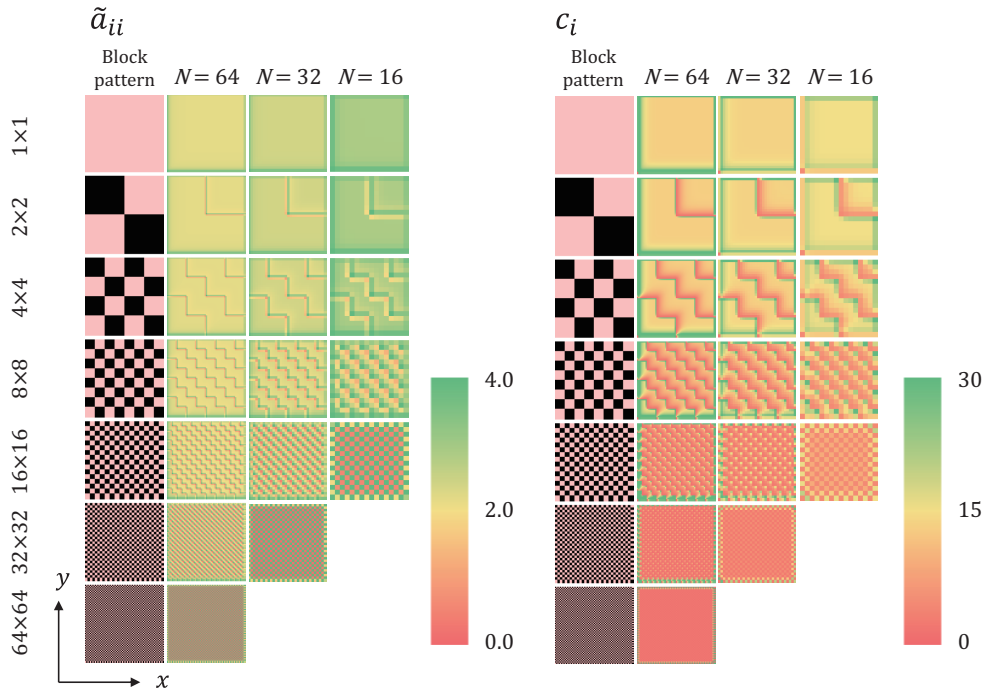

Figure 6.1: Diagonal element sizes $\tilde{a}_{ii}$ and c_i of A factorized with PMILU(0) ($\zeta = 2\pi^2$).

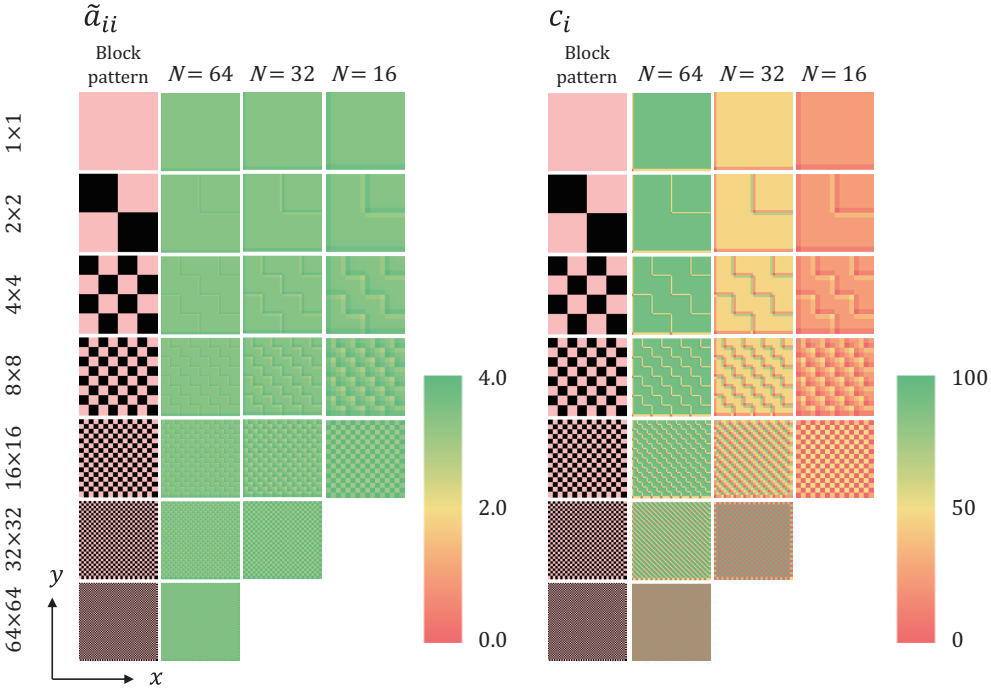

Figure 6.2: Diagonal element sizes $\tilde{a}_{ii}$ and c_i of A factorized with ILU(0).

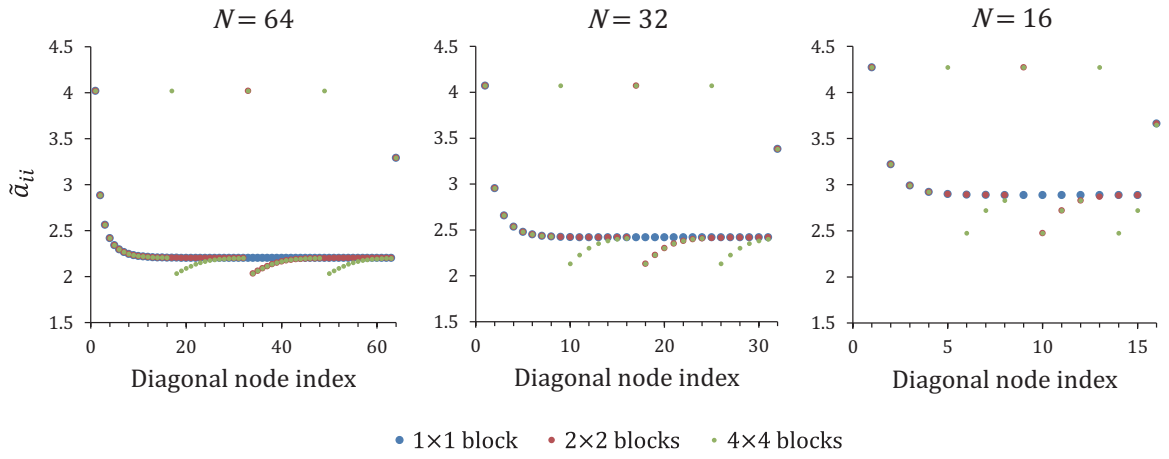

Figure 6.3: PMILU(0) ($\zeta = 2\pi^2$) factorized $\tilde{a}_{ii}$ corresponding to diagonal node i.

6.3.3 Eigenvalue Spectrum

As shown in Chapter 3, convergence depends not only on the condition number but also on the eigenvalue spectrum. Therefore, in this section, we compute all eigenvalues of the preconditioned matrix $K^{-1}A$ and investigate the eigenvalue spectrum.

Herein, an example obtained from a model problem (6.1.3) with unknowns $n = 1024$ on a 32×32 grid point is described. The eigenvalues of $K^{-1}A$ were obtained using the `eigs` function in MATLAB.

For the perturbation factor ζ, we use $\zeta = 2\pi^2$ as described in Section 6.3.2. Figs. 6.4 and 6.5 show the eigenvalues, arranged in ascending order, under the assumption that the 'λ index' of the smallest eigenvalue is 1, and the 'λ index' of the largest eigenvalue is 1024.

Fig. 6.4 shows the eigenvalue spectrum of PMILU(0) with $\zeta = 2\pi^2$. Because the increase in the maximum eigenvalue owing to the increase in the number of blocks is greater than the increase in the minimum eigenvalue, the condition number increases with the number of blocks. In Section 6.3.2, we show that the distribution of $\tilde{a}_{ii}$ with a small number of blocks, such as 2×2 and 4×4 blocks, is close to that of the natural ordering of a 1×1 block with the exception of some of the red blocks. The eigenvalue distribution under these conditions also shows the same trend as the 1×1 block except for some large isolated eigenvalues.

Fig. 6.5 shows the eigenvalue spectrum of ILU(0). In ILU(0), both the minimum and maximum eigenvalues are small, and no large eigenvalues appear even with a large number of blocks. Because the change with an increase in the number of blocks is small, the degradation is less than that of PMILU(0) under the condition in which the numbers of nodes and blocks are similar.

PMILU(0) is characterized by a small number of extreme values on both sides and a gradual change between them under natural ordering. A small number of large eigenvalues appear in the case of a low block count ordering. Such extreme eigenvalues have a smaller effect on convergence. In the CG method, their effect is removed by an additional number of iterations that is equal to the number of isolated large eigenvalues [37]. For block numbers of 16×16 or higher, the eigenvalues on the maximum side form several populations. In PMILU(0) and ILU(0), with nodal RB ordering, where the number of blocks is equal to the number of nodes,

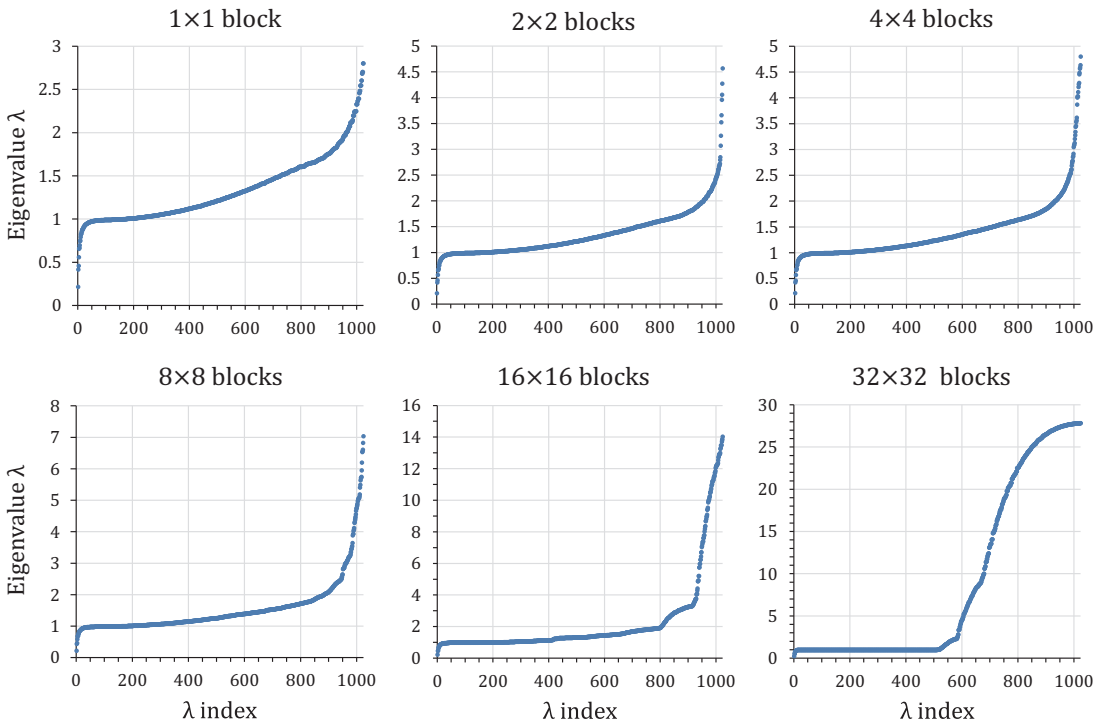

Figure 6.4: Eigenvalue spectrum of $K^{-1}A$ with PMILU(0) ($\zeta = 2\pi^2$).

a nearly clustering population of eigenvalues close to eigenvalue 1 appears, and the number of iterations increases slowly with respect to the condition number. Convergence is improved through an eigenvalue cluster formation, and this phenomenon occurs for a number of blocks close to the number of nodes. These features of the eigenvalue spectrum that lead to better convergence are thought to be one of the reasons for the excellent convergence of BRB ordering.

The eigenvalue spectrum of $K^{-1}A$ with the PMILU(0) factorization for the same three problem sizes as in Section 6.3.2, $N = 64$, $N = 32$ and $N = 16$, is shown in Fig. 6.6. When the number of blocks is small relative to the problem size, i.e., the number of nodes per block is large, the spectrum does not change significantly with the number of blocks. By contrast, when this quantity is small, conditions with the same number of nodes per block show results with a similar trend. These results are similar to the trend shown in Figs. 4.4, 4.5, and 5.12, where the number of iterations increases when the number of nodes per block becomes almost the same small value in the comparison with different problem sizes.

6.4 Conclusion

In this chapter, we examined the convergence acceleration effects of a preconditioner using a combination of BRB ordering and PMILU(0). Under the ordering condition with a large number of blocks, the number of discarded fill-ins increases, and thus the diagonal elements become smaller owing to the larger compensation in PMILU(0). Therefore, in such a case, the convergence is better under the condition of large perturbations to the diagonal elements.

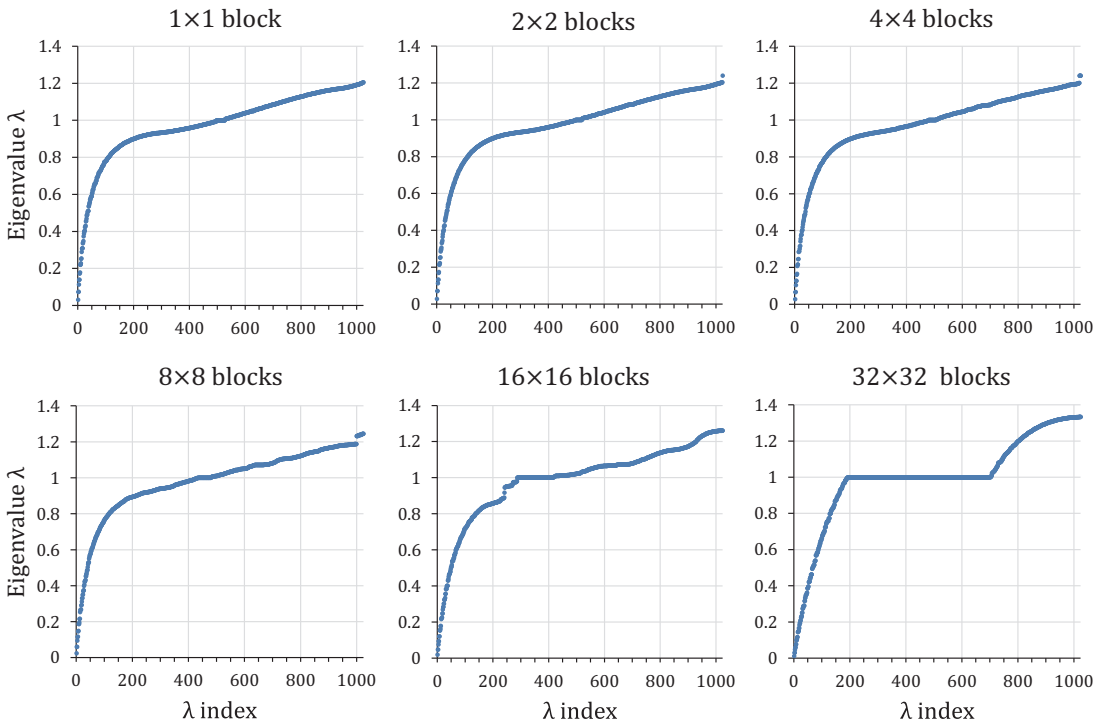

Figure 6.5: Eigenvalue spectrum of $K^{-1}A$ with ILU(0).

Except for cases in which the perturbation coefficient is extremely large, the condition number, remainder matrix norm, and number of iterations all become larger as the number of blocks increases. The perturbation coefficient with the best convergence differs depending on the number of blocks.

A sufficient condition for the condition number to be $O(h^{-1})$ is that the element corresponding to the fill-in position of the remainder matrix is less than or equal to $(1+ch)^{-1}$. This condition is satisfied for all nodes under natural ordering. For BRB ordering with a small number of blocks, these values are close to those of natural ordering except at the left and bottom boundaries of the red blocks. Although the condition number increased with the increase in the number of blocks, the eigenvalue spectrum showed favorable properties for a fast convergence, such as isolated extreme eigenvalues and clustering. These features partially explain the high convergence acceleration effect of BRB ordering.

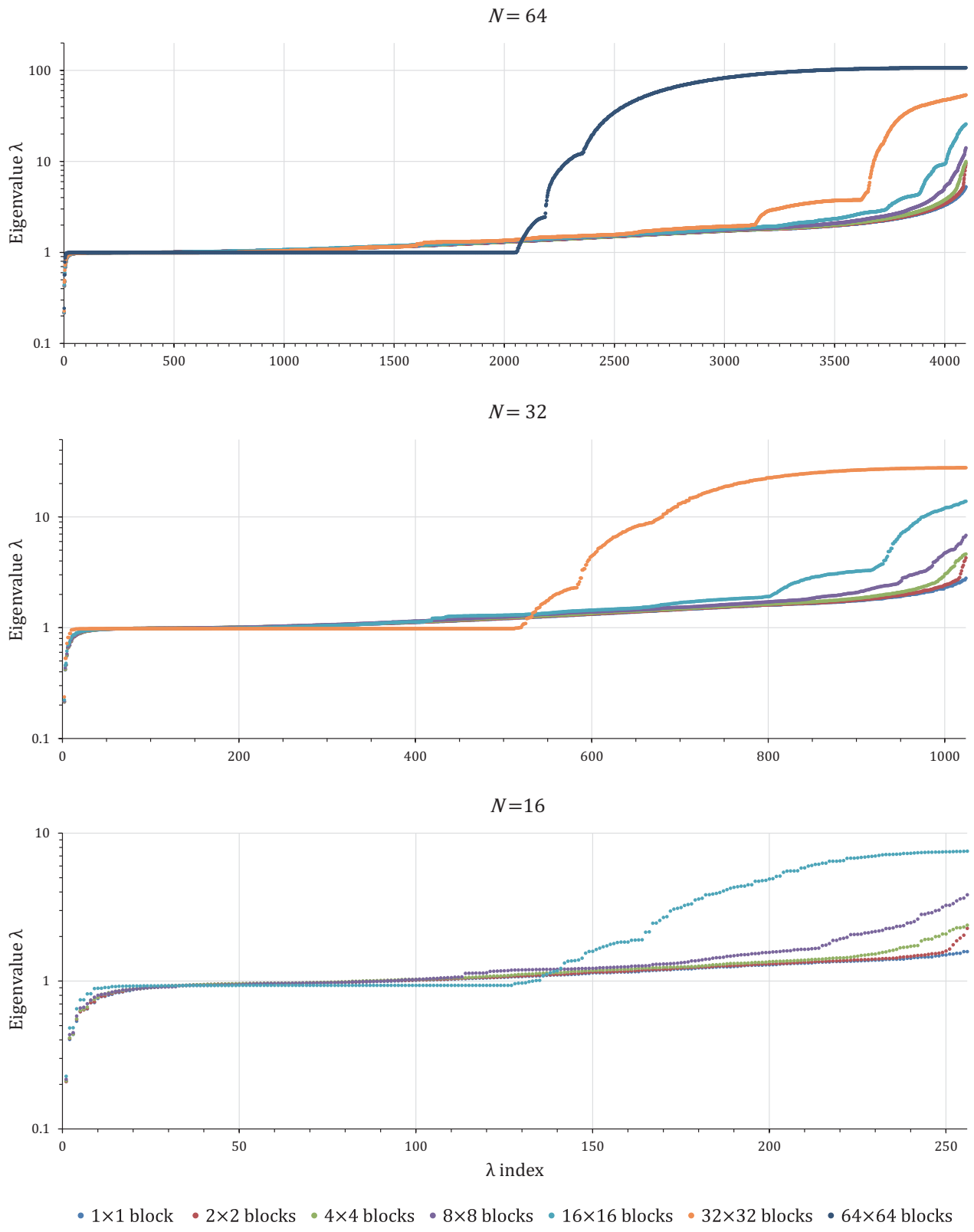

Figure 6.6: Eigenvalue spectrum of $K^{-1}A$ with PMILU(0) ($\zeta = 2\pi^2$) for problem size N.

Chapter 7

Conclusion

In this thesis, we describe a method for the acceleration of the solution-finding process of the Poisson equation, which accounts for a significant amount of the computational time associated with the application of plasma simulations to industrial manufacturing equipment using the PIC method. Our strategy is to use the Krylov subspace method preconditioned by the MILU(0) factorization as a solver and to parallelize it on a GPU.

In Chapter 4, we observed that combining a BRB ordering strategy and a unperturbed MILU(0) could yield a zero pivot and presented a necessary and sufficient condition for such a zero pivot to occur. The condition was defined as the presence of an inner black block such that all neighboring red blocks in the direction of decreasing x, y, and z are inner blocks. If i is the largest node number in the interior black block for which this condition is satisfied, then the value of the i-th pivot of the unperturbed MILU(0) factorized coefficient matrix is zero. To alleviate this problem, it is useful to add perturbations to the diagonal elements of the coefficient matrix in advance or to relax the amounts be added to the diagonal elements to compensate for the fill-in, which is dropped during factorization.

We present a theorem that states that introducing perturbations and relaxations solves the problem of zero pivots and that pivots remain above a certain threshold for at least a certain class of matrices. Numerical results have shown that the convergence of the unperturbed MILU(0) preconditioner deteriorates in the presence of node i, which satisfies the requirement for the occurrence of a zero pivot. The results of the numerical experiments further showed that introducing perturbation or relaxation not only suppresses the deterioration of convergence under in the conditions in which a zero-pivot occurs but also improves convergence over the unperturbed MILU(0) even in cases with no zero pivots.

In BRB ordering, the number of iterations required to satisfy a given convergence criterion tends to increase with the increase in the number of isochromatic blocks corresponding to the degree of parallelism. Numerical experimentation using a multicore CPU with the order of 10 parallel blocks showed relatively little effect of deterioration of convergence owing to the increase in the number of blocks. A simplified parallelization performed by inserting OpenMP directives into the loop has been shown to provide computational acceleration efficiencies. In the convergence study, the combination of the perturbed or relaxed MILU(0) (PMILU(0)/RMILU(0)) factorization and the BRB ordering led to high convergence rates of up to 10^3 blocks in the $10^5 - 10^6$ node problem. Therefore, this approach was considered promising as a prerequisite in massively parallel environments and was considered as an option.

In Chapter 5, we implemented a parallel linear equation solver for GPU hardware based

on the RMILU(0) preconditioner and BRB ordering. We used OpenACC directives for GPU parallelization for high maintainability and made several optimizations to fully exploit the performance capabilities of GPU.

In BRB ordering, the number of iterations is smaller than in nodal RB ordering. However, there is a problem associated with adjacent threads accessing data at remote locations, because as parallel computation is performed for each block, which contains multiple rows and columns when the elements of the matrix are stored in the row or column direction. When manipulating sparse matrix data on a GPU, this problem results in slower data access because the memory throughput of the GPU hardware cannot be fully utilized. Consequently, in simple parallelization, nodal RB ordering is faster despite the large number of iterations required, because neighboring rows are computed in parallel. To solve this problem, we realize coalesced access by sorting the data to enable adjacent threads to access consecutive data. In addition, a mixed-precision iterative solver that uses single precision for preconditioner was implemented as a preconditioner using single-precision to reduce the amount of data and utilize the fast single-precision computational capabilities of the GPU.

To optimize the parameters, we examined the effects of the block partitioning condition of the BRB ordering and the size of the parallel granularity of OpenACC. Balancing the two requirements of moderate block size to retain the convergence speed and a sufficiently large number of blocks to assign tasks to a large number of threads in a GPU resulted in blocks in multiples of 32 per SM, greater than or equal to 32×6. The optimal value of the calculation granularity `gang-worker-vector` set in the OpenACC directive clause is not only dependent on the performance of GPU and the processing content in the loop but also on the problem size.

These implementation techniques and optimizations dramatically improved the computational speed, and the results showed that the proposed method was able to obtain solutions faster than major GPU-based iterative solver libraries.

A further investigation of the convergence of the perturbed MILU(0) preconditioning with BRB ordering using the model problem has been presented in Chapter 6. When the spectral condition number of the preconditioned coefficient matrix is $O(h^{-1})$, it exhibits good convergence. In the case of a small number of blocks, a sufficient condition for satisfying this requirement of the diagonal element being $O(h^{-1})$ was almost achieved. Increasing the number of blocks resulted in an increase in the number of elements for which the condition was not satisfied; however, the eigenvalue spectrum exhibited characteristics which accelerated convergence, such as isolated maximum and minimum eigenvalues, clustering, and gaps. We considered them to be the reason the reason behind the good convergence of BRB ordering over a wide range of degree of parallelism.

Herein, we mainly focused on the experimental analysis of a preconditioner, targeting NVIDIA GPU of the Kepler and Pascal generations with superior double-precision arithmetic performance. As the next step in this research, we intend to proceed with a theoretical analysis of the characteristics of the preconditioner. Furthermore, additional implementations must be developed variations to accommodate accelerator devices with various specifications, including less than single precision. These may be expected to lead to better insights into the properties of other ordering strategies and preconditioners, and accelerate the solution of linear simultaneous equations to broaden the capability of practical simulation models.

Acknowledgment

I would like to express my gratitude to my advisor, Professor Yusaku Yamamoto, for his guidance in conducting this research. I am deeply grateful to Professors Hidenori Ogata, Nobito Yamamoto, Tadashi Yamazaki, Tetsu Narumi, and Tomoya Tatsuno for serving on the review committee. I would also like to give a special acknowledgment to Professor Takeshi Iwashita, Hokkaido University, for his valuable comments and advice as the originator of the block red-black ordering. Furthermore, I would like to thank Professor Shuhei Kudo and all of my colleagues in the Yusaku Yamamoto's laboratory at the University of Electro-Communications for their continuous support and suggestions.

Bibliography

[1] Charles K. Birdsall. Particle-in-cell charged-particle simulations, plus Monte Carlo collisions with neutral atoms, PIC-MCC. In *IEEE Transactions on Plasma Science*, 19(2), pp. 65–85, 1991.

[2] V. Vahedi, G. DiPeso, C. K. Birdsall, M. A. Lieberman, T. D. Rognlien. Capacitive RF discharges modelled by particle-in-cell Monte Carlo simulation. I. Analysis of numerical techniques. In *Plasma Sources Science and Technology*, 2(4), pp. 261–272, 1993.

[3] Ivar Gustafsson. A class of first order factorization methods. In *BIT Numerical Mathematics*, 18(2), pp. 142–156, 1978.

[4] Ivar Gustafsson. Modified incomplete Cholesky (MIC) methods. In Evans, D. (ed.), *Preconditioning methods: analysis and applications*, Gordon and Breach, New York, pp. 265–294, 1983.

[5] Anne Greenbaum. *Iterative methods for solving linear systems.* Society for Industrial and Applied Mathematics, 1997.

[6] Elizabeth Cuthill, James McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM Proceedings of the 24th National Conference*, pp. 157–172, 1969.

[7] Alan George. Nested dissection of a regular finite element mesh. In *SIAM Journal on Numerical Analysis*, 10(2), pp. 345–363, 1982.

[8] Alan George. An automatic one-way dissection algorithm for irregular finite element problems. In *SIAM Journal on Numerical Analysis*, 17(6), pp. 740–751, 1980.

[9] Wing-Man Chan, Alan George. A linear time implementation of the reverse Cuthill-McKee algorithm. In *BIT Numerical Mathematics*, 20(1), pp. 8–14, 1980.

[10] Loyce M. Adams, James M. Ortega. A multi-color SOR method for parallel computation. In *proceedings 1982 Internat. Conf. on Parallel Processing*, pp. 53–56, 1982.

[11] Yousef Saad. Iterative Methods for Sparse Linear Systems, 2nd Edition, SIAM, Philadelphia, 2003.

[12] Victor Eijkhout. Introduction to High Performance Scientific Computing, Lulu.com, 2013.

[13] Takeshi Iwashita, Masaaki Shimasaki. Block red-black ordering: A new ordering strategy for parallelization of ICCG method. In *International Journal of Parallel Programming*, 31(1), pp. 55–75, 2003.

[14] Takeshi Iwashita, Masaaki Shimasaki. Block red-black ordering for parallelized ICCG solver with fewer synchronization points. In *IPSJ Journal*, 43(4), pp. 893–904, 2002.

[15] Takeshi Iwashita, Yuuichi Nakanishi, Masaaki Shimasaki. Comparison criteria for parallel orderings in ILU preconditioning. In *SIAM Journal on Scientific Computing*, 26(4), pp. 1234–1260, 2005.

[16] N. Guessous, O. Souhar. The effect of block red-black ordering on block ILU preconditioner for sparse matrices. In *Journal of Applied Mathematics and Computing*, 17(1), pp. 283–296, 2005.

[17] Victor Eijkhout. Beware of unperturbed modified incomplete factorizations. In B. Belgium, R. Beauwens and P. de Groen (Eds.), *Proc. of the IMACS International Symposium on Iterative Methods in Linear Algebra*, 1992.

[18] Eric J. Hallman, Kristian C. Beckwith, Peter Stoltz. Performance and scalability of parallel PIC and fluid codes on Xeon Phi based supercomputers. In *2014 IEEE 41st International Conference on Plasma Sciences (ICOPS) held with 2014 IEEE International Conference on High-Power Particle Beams (BEAMS)*, IEEE, 2014.

[19] Viktor K. Decyk, Singh V. Tajendra . Particle-in-cell algorithms for emerging computer architectures. In *Computer Physics Communications*, 185(3), pp. 708–719, 2014.

[20] Steven W. D. Chien, Jonas Nylund, Gabriel Bengtsson, Ivy B. Peng, Artur Podobas, Stefano Markidis. sputniPIC: an Implicit Particle-in-Cell Code for Multi-GPU Systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 149–156. IEEE, 2020.

[21] A. Vapirev, J. Deca, G. Lapenta, S. Markidis, I. Hur. Initial results on computational performance of Intel many integrated core, sandy bridge, and graphical processing unit architectures: implementation of a 1D c++/OpenMP electrostatic particle-in-cell code. In *Concurrency and Computation: Practice and Experience*, 27(3), pp. 581–593, 2015.

[22] Akemi Shioya, Yusaku Yamamoto. The danger of combining block red-black ordering with modified incomplete factorizations and its remedy by perturbation or relaxation. In *Japan Journal of Industrial and Applied Mathematics*, 35(1), pp. 195–216, 2018.

[23] Akemi Shioya, Yusaku Yamamoto. Block red-black MILU (0) preconditioner with relaxation on GPU. In *Parallel Computing*, 103, 102760, 2021.

[24] 塩谷 明美, 山本有作. ブロック赤黒順序付けされた摂動付き修正不完全分解前処理の収束性解析. 応用数理学会 *2021* 年度年会, D1-1-4, 2021.

[25] M. Lobet, E. d'Humires, M. Grech, C. Ruyer, X. Davoine, L. Gremillet. Modeling of radiative and quantum electrodynamics effects in PIC simulations of ultra-relativistic laser-plasma interaction. In *Journal of Physics: Conference Series*, 688, 012058. IOP Publishing, 2016.

[26] T. Umeda, Y. Omura, T. Tominaga, H. Matsumoto. A new charge conservation method in electromagnetic particle-in-cell simulations In *Computer Physics Communications*, 156(1), pp. 73–85, 2003.

[27] L. Garrigues, B. Tezenas du Montcel, G. Fubiani, F. Bertomeu, F. Deluzet, J. Narski. Application of sparse grid combination techniques to low temperature plasmas particle-in-cell simulations. I. Capacitively coupled radio frequency discharges. In *Journal of Applied Physics*, 129(15), 153303, 2021.

[28] Yohei Miyake, Hiroshi Nakashima. Low-cost load balancing for parallel particle-in-cell simulations with thick overlapping layers. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 1107–1114. IEEE, 2013.

[29] D. Wnderlich, S. Mochalskyy, I. M. Montellano, A. Revel. Review of particle-in-cell modeling for the extraction region of large negative hydrogen ion sources for fusion. In *Review of Scientific Instruments*, 89(5), 052001, 2018.

[30] J. Deca, G. Lapenta, R. Marchand, S. Markidis. Spacecraft charging analysis with the implicit particle-in-cell code iPic3D. In *Physics of Plasmas*, 20(10), 102902, 2013.

[31] Ahmad Nizam, Hideyuki Usui, Yohei Miyake. The Particle-In-Cell simulation on LEO spacecraft charging and the wake structure using EMSES. In *Journal of Advanced Simulation in Science and Engineering*, 6(1), pp. 21–31, 2019.

[32] C. H. Shon, J. K. Lee. Modeling of magnetron sputtering plasmas. In *Applied Surface Science*, 192(1-4), pp. 258–269, 2002.

[33] Ivan Kolev, Annemie Bogaerts. Detailed numerical investigation of a dc sputter magnetron. In *IEEE Transactions on Plasma Science*, 34(3), pp. 886–894, 2006.

[34] Ivan Kolev, Annemie Bogaerts. Numerical study of the sputtering in a dc magnetron. In *Journal of Vacuum Science & Technology A: Vacuum, Surfaces, and Films*, 27(1), pp. 20–28, 2009.

[35] MPICH, http://www.mpich.org/.

[36] Marcin Turek, Marcin Brzuszek, Juliusz Sielanko. An MPI-based parallel code for high performance 3-D particle-in-cell ion source plasma simulation. In *Annales Universitatis Mariae Curie-Sklodowska, sectio AI-Informatica*, 6(1), pp. 137–148, 2007.

[37] Owe Axelsson. A class of iterative methods for finite element equations. In *Computer Methods in Applied Mechanics and Engineering*, 9(2), pp. 123–137, 1976.

[38] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, Henk van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*. Society for Industrial and Applied Mathematics, 1994.

[39] Paul Concus, Gene H. Golub, Dianne P.O'Leary. A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations. In *Sparse Matrix Computations*, Academic Press, pp. 309–332, 1976.

[40] David Kratzer, Seymour V. Parter, Michael Steuerwalt. Block splittings for the conjugate gradient method. In *Computers & Fluids*, 11(4), pp. 255–279, 1983.

[41] Owe Axelsson, Gunhild Lindskog. On the eigenvalue distribution of a class of preconditioning methods. In *Numerische Mathematik*, 48(5), pp. 479–498, 1986.

[42] Are Magnus Bruaset. *A survey of preconditioned iterative methods.* Pitman Research Notes in Mathematics Series, vol. 328. Lonman Scientific and Technical: Harlow, Esssex, England, 1995.

[43] Todd Dupont, Richard P. Kendall, H. H. Rachford, Jr.. An approximate factorization procedure for solving self-adjoint elliptic difference equations. In *SIAM Journal on Numerical Analysis*, 5(3), pp. 559–573, 1968.

[44] Owe Axelsson. Bounds of eigenvalues of preconditioned matrices. In *SIAM Journal on Matrix Analysis and Applications*, 13(3), pp. 847–862, 1992.

[45] Tony F. Chan. Fourier analysis of relaxed incomplete factorization preconditioners. In *SIAM Journal on Scientific and Statistical Computing*, 12(3), pp. 668–680, 1992.

[46] Henk A. van der Vorst. High performance preconditioning. In *SIAM Journal on Scientific and Statistical Computing*, 10(6), pp. 1174–1185, 1989.

[47] OpenACC, http://www.openacc.org/.

[48] Message Passing Interface Forum, http://www.mpi-forum.org/.

[49] Joshua Payne, Dana Knoll, Allen McPherson, William Taitano, Luis Chacon, Guangye Chen, Scott Pakin. Computational co-design of a multiscale plasma application: A process and initial results. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IEEE, pp. 1093–1102, 2014.

[50] Xiaocheng Liu, Ziming Zhong, Kai Xu. A hybrid solution method for CFD applications on GPU-accelerated hybrid HPC platforms. In *Future Generation Computer Systems*, 56, pp. 759–765, 2016.

[51] Alfredo Buttari, Markus Huber, Philippe Leleux, Théo Mary, Ulrich Ruede, Barbara Wohlmuth. Block Low Rank Single Precision Coarse Grid Solvers for Extreme Scale Multigrid Methods. https://hal.archives-ouvertes.fr/hal-02528532, 2020.

[52] A. B. Alves, E. N. Asada, A. Monticelli. Critical evaluation of direct and iterative methods for solving Ax= b systems in power flow calculations and contingency analysis. In *Proceedings of the 21st International Conference on Power Industry Computer Applications. Connecting Utilities. PICA 99. To the Millennium and Beyond (Cat. No. 99CH36351)*, IEEE, pp. 15–21, 1999.

[53] Iain S. Duff, and Gerard A. Meurant. The effect of ordering on preconditioned conjugate gradients. In *BIT Numerical Mathematics*, 29(4), pp. 635–657, 1989.

[54] Shun Doi, Takumi Washio. Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations. In *Parallel Computing*, 25(13-14), pp. 1995–2014, 1999.

[55] Robert Bridson, Wei-Pai Tang. Comparison criteria for parallel orderings in ILU preconditioning. In *SIAM Journal on Scientific Computing*, 22(5), pp. 1527–1532, 2001.

[56] Takeshi Iwashita, Hiroshi Nakashima, Yasuhito Takahashi. Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in ICCG method. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IEEE, pp. 474–483, 2012.

[57] Gaku Ishii, Yusaku Yamamoto, Takeshi Takaishi. Acceleration and Parallelization of a Linear Equation Solver for Crack Growth Simulation Based on the Phase Field Model. Mathematics, 9(18), 2248, 2021.

[58] Victor Eijkhout. Analysis of parallel incomplete point factorizations. In *Linear Algebra and Its Applications*, 154, pp. 723–740, 1991.

[59] NVIDIA, CUDA Toolkit, https://developer.nvidia.com/cuda-toolkit.

[60] The Khronos Group Inc, OpenCL, https://www.khronos.org/opencl/.

[61] Lucas Grillo, Ruymn Reyes, Francisco de Sande. Performance evaluation of OpenACC compilers. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, IEEE, pp. 656–663, 2014.

[62] Stefan Rosenberger, Gundolf Haase. Effective OpenACC Parallelization for Sparse Matrix Problems. SFB-Report No. 2017–008, 2017.

[63] Olav Aanes Fagerlund, Takeshi Kitayama, Gaku Hashimoto, Hiroshi Okuda. Effect of GPU communication-hiding for SpMV using OpenACC. In *International Journal of Computational Methods*, 13(02), 1640011, 2016.

[64] Jeff Bolz, Ian Farmer, Eitan Grinspun, Peter Schroder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *ACM Transactions on Graphics (TOG)*, 22(3), pp. 917–924, 2003.

[65] Rudi Helfenstein, Jonas Koko. Parallel preconditioned conjugate gradient algorithm on GPU. In *Journal of Computational and Applied Mathematics*, 236(15), pp. 3584–3590, 2012.

[66] Maxim Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. In *Technical Report NVR-2011*, NVIDIA Corp., Westford, MA, USA, 2011.

[67] Jiaquan Gao, Ronghua Liang, Jun Wang. Research on the conjugate gradient algorithm with a modified incomplete Cholesky preconditioner on GPU. In *Journal of Parallel and Distributed Computing*, 74(2), pp. 2088–2098, 2014.

[68] Weifeng Liu, Ang Li,,Jonathan Hogg, Iain S. Duff, Brian Vinter. A synchronization-free algorithm for parallel sparse triangular solves. In *European Conference on Parallel Processing*, Springer, Cham, pp. 617–630, 2016.

[69] Daniel Erguiz, Ernesto Dufrechou, Pablo Ezzatti. Assessing sparse triangular linear system solvers on GPUs. In *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, IEEE, pp. 37–42, 2017.

[70] Pingfan Li, Xuhao Chen, Zhe Quan, Jianbin Fang, Huayou Su, Tao Tang, Canqun Yang. High performance parallel graph coloring on GPGPUs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, pp. 845–854, 2016.

[71] Maxim Naumov, Patrice Castonguay, Jonathan Cohen. Parallel Graph Coloring with Applications To the Incomplete-LU Factorization on the GPU. In *Technical Report NVR-2015-001*, NVIDIA Corp., Westford, MA, USA, 2015.

[72] Edmond Chow, Hartwig Anzt, Jennifer Scott, Jack Dongarra. Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. In *Journal of Parallel and Distributed Computing*, 119, pp. 219–230, 2018.

[73] Jiaquan Gao, Yu Wang, Jun Wang, Ronghua Liang. Adaptive optimization modeling of preconditioned conjugate gradient on multi-GPUs. In *ACM Transactions on Parallel Computing (TOPC)*, 3(3), pp. 219–230, 2016.

[74] Hartwig Anzt, Thomas K. Huckle, Jurgen Brackle, Jack Dongarra. Incomplete sparse approximate inverses for parallel preconditioning. In *Parallel Computing*, 71, pp. 1–22, 2018.

[75] Marco Ament, Gunter Knittel, Daniel Weiskopf, Wolfgang Strasser. A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2010.

[76] Yan Chen, Xuhong Tian, Hui Liu, Zhangxin Chen, Bo Yang, Wenyuan Liao, Peng Zhang, Ruijian He, Min Yang. Parallel ILU preconditioners in GPU computation. In *Soft Computing*, 22(24), pp. 8187–8205, 2018.

[77] NVIDIA, cuBLAS, https://developer.nvidia.com/cublas.

[78] NVIDIA, cuSPARSE, https://developer.nvidia.com/cusparse.

[79] MAGMA, http://icl.cs.utk.edu/magma/.

[80] ViennaCL, http://viennacl.sourceforge.net/.

[81] Ginkgo, https://ginkgo-project.github.io/.

[82] PETSc, https://www.mcs.anl.gov/petsc.

[83] Takeshi Iwashita, Senxi Li, and Takeshi Fukaya. Hierarchical block multi-color ordering: A new parallel ordering method for vectorization and parallelization of the sparse triangular solver in the ICCG method. In *CCF Transactions on High Performance Computing*, 2(2), pp. 84–97, 2020.

[84] Hiroto Tadano, Tetsuya Sakurai. On single precision preconditioners for Krylov subspace iterative methods. In *International Conference on Large-Scale Scientific Computing*, Springer, Berlin, Heidelberg, pp. 721–728, 2007.

[85] Yvan Notay. Flexible conjugate gradients. In *SIAM Journal on Scientific Computing*, 22(4), pp. 1444–1460, 2000.

[86] Judith A. Vogel. Flexible BiCG and flexible Bi-CGSTAB for nonsymmetric linear systems. In *Applied Mathematics and Computation*, 188(1), pp. 226–233, 2007.

[87] Jie Chen, Lois C. McInnes, Hong Zhang. Analysis and practical use of flexible BiCGStab. In *Journal of Scientific Computing*, 68(2), pp. 803–825, 2016.

[88] Azamat Mametjanov, Daniel Lowell, Ching-Chen Ma, Boyana Norris. Autotuning stencil-based computations on GPUs. In *2012 IEEE international conference on cluster computing*, pp. 266–274, 2012.

[89] Daniel Lowell, Jeswin Godwin, Justin Holewinski, Deepan Karthik, Chekuri Choudary, Azamat Mametjanov, Boyana Norris, Gerald Sabin, P. Sadayappan, Jason Sarich. Stencil-aware GPU optimization of iterative solvers. In *SIAM Journal on Scientific Computing*, 35(5), pp. S209–S228, 2013.

[90] Kazuya Matsumoto, Yasuhiro Idomura, Takuya Ina, Akie Mayumi, Susumu Yamada. Implementation and performance evaluation of a communication-avoiding gmres method for stencil-based code on GPU cluster. In *The Journal of Supercomputing*, 75(12), pp. 8115–8146, 2019.

[91] Jonathan Cohen, Patrice Castonguay. Efficient graph matching and coloring on the GPU. In *GPU Technology Conference*, pp. 1–10, 2012.

[92] OpenBLAS: An optimized BLAS library, https://www.openblas.net/.

[93] Boost C++ Libraries, https://www.boost.org/.

[94] Edmond Chow, Aftab Patel. Fine-grained parallel incomplete LU factorization. In *SIAM Journal on Scientific Computing*, 37(2), pp. C169–C193, 2015.

[95] OpenMPI, https://www.open-mpi.org/.

[96] Tony F. Chan, Howard C. Elman. Fourier Analysis of Iterative Methods for Elliptic Problems. In *SIAM Review*, 31(1), pp. 20–49, 1989.

Appendix A

Convergence of Poisson Equation in Other Time Steps of PIC Method

In Section 4.4.2, the Poisson equation obtained from the 1,000th time step of the PIC simulation with the conditions shown in Section 4.4.1 was solved. In this appendix, we show the number of iterations using the coefficient matrices and right-hand side obtained from 100th, 10,000th, and 30,000th time steps as input to the same solver. The electric susceptibility χ and the charge density ρ in the Poisson equation (2.1.14) change with the progression of time steps. Therefore, the coefficient matrix and the right-hand side vector that depend on χ and ρ are different for each time step.

Figs. A.1 – A.3, Figs. A.4 – A.6 and Figs. A.7 – A.9 show the number of iterations required for convergence at 100th step, 10,000th step and 30,000th step, respectively.

Figs. A.1, A.4 and A.7 are the results of unperturbed MILU(0) preconditioner under the same conditions as Fig. 4.3. In these figures, the missing plots in graph (b) are the conditions under which the calculation was terminated because the number of iterations exceeded 100,000. Figs. A.2, A.5 and A.8 are the results of using PMILU(0) preconditioner with the same conditions as Fig. 4.4. Similarly, Figs. A.3, A.6 and A.9 are the results of using RMILU(0) preconditioner with the same conditions as Fig. 4.5.

The number of iterations tested in each step, with the same parameters of incomplete factorization and the same block partitioning condition, shows the same trend as that of the 1,000th time step shown in Section 4.4.2.

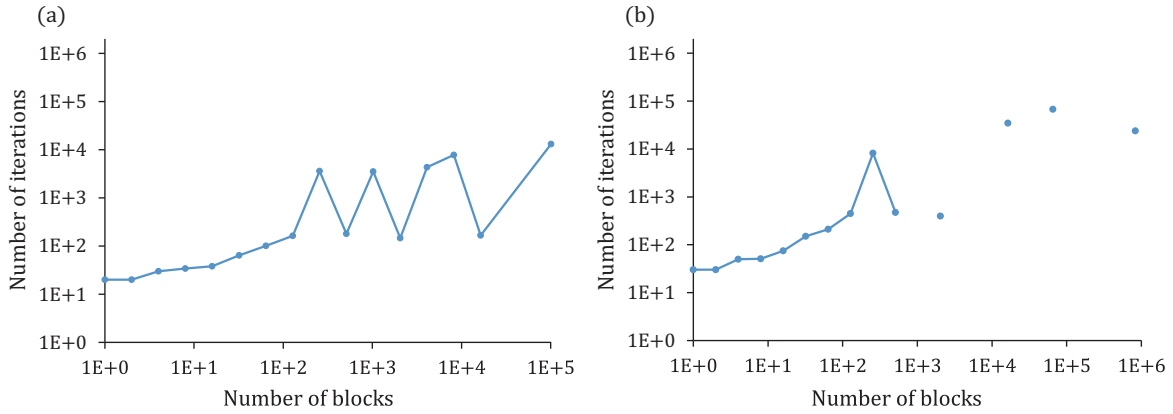

Figure A.1: Number of blocks N and number of iterations with MILU(0) preconditioner at PIC 100th step. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

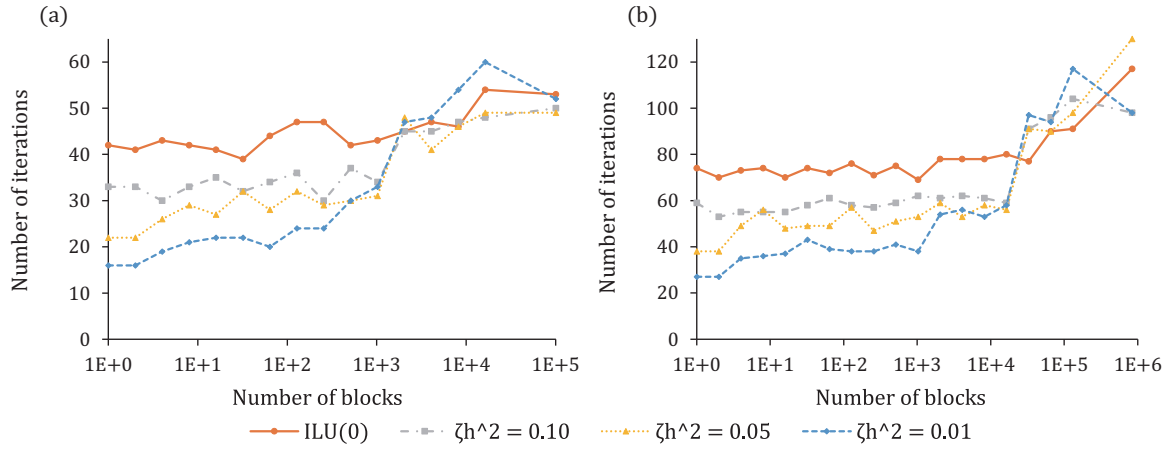

Figure A.2: Number of blocks N and number of iterations with PMILU(0) preconditioner at PIC 100th step. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

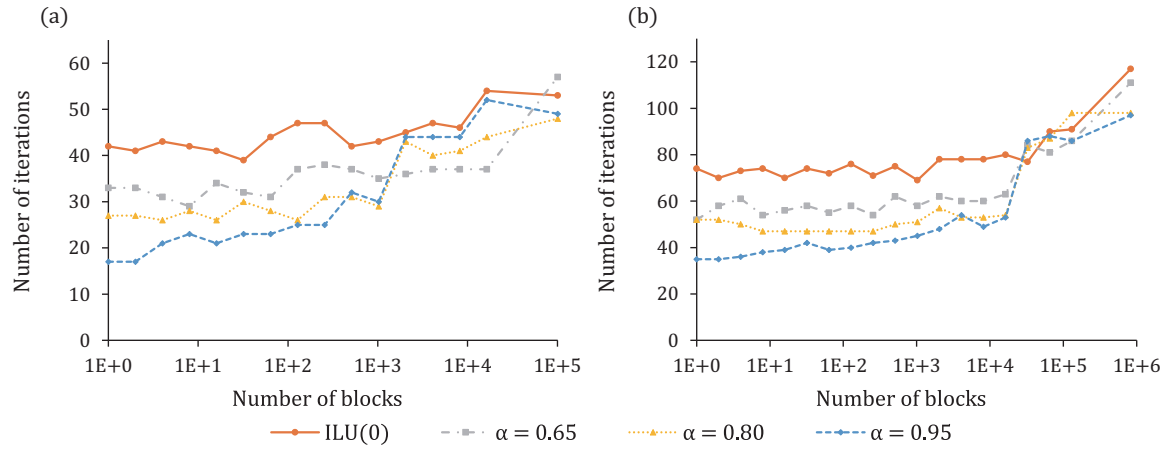

Figure A.3: Number of blocks N and number of iterations with RMILU(0) preconditioner at PIC 100th step. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

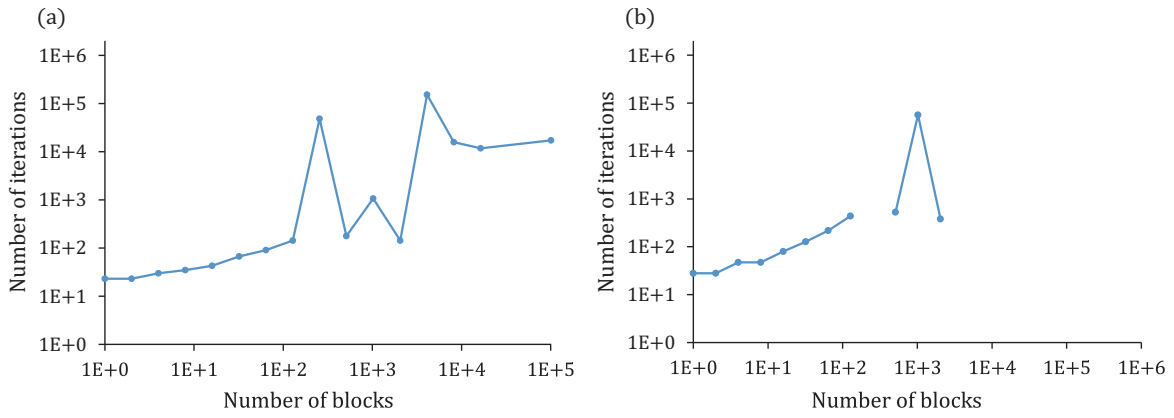

Figure A.4: Number of blocks N and number of iterations with MILU(0) preconditioner at PIC 10,000th step. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

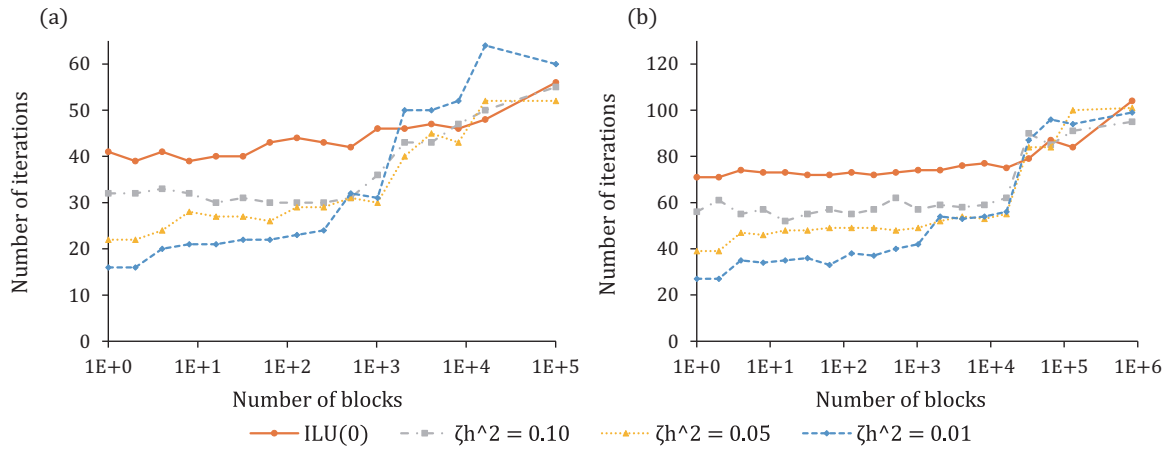

Figure A.5: Number of blocks N and number of iterations with PMILU(0) preconditioner at PIC 10,000th step. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

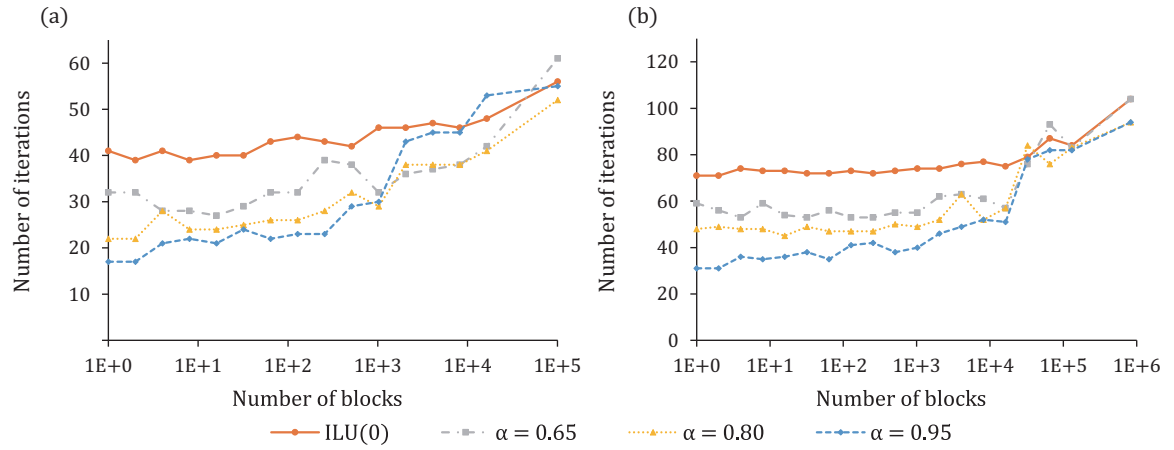

Figure A.6: Number of blocks N and number of iterations with RMILU(0) preconditioner at PIC 10,000th step. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

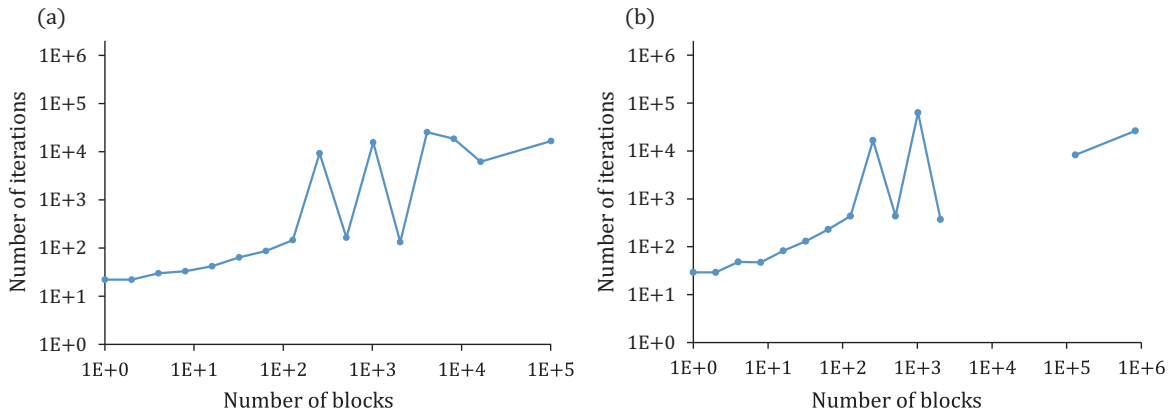

Figure A.7: Number of blocks N and number of iterations with MILU(0) at PIC 30,000th step. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

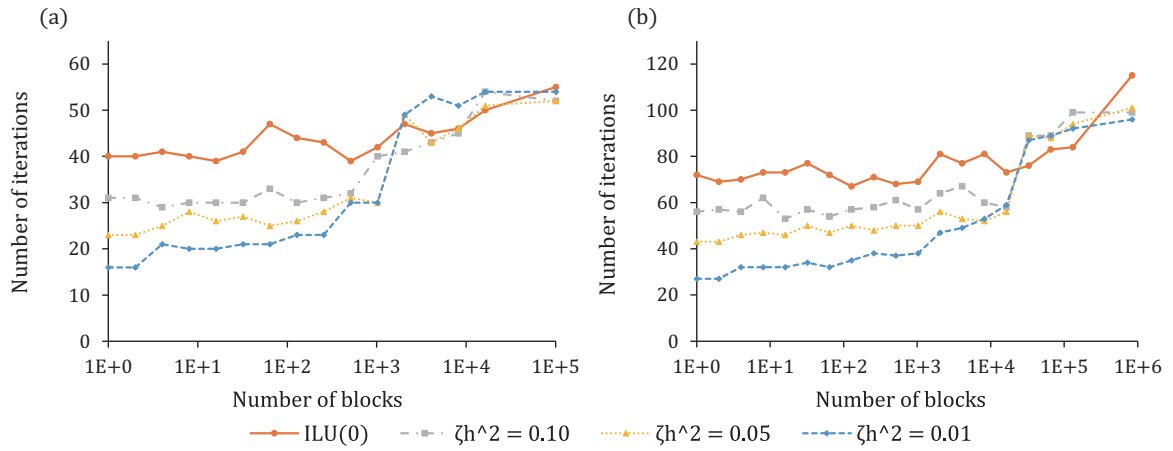

Figure A.8: Number of blocks N and number of iterations with PMILU(0) preconditioner at PIC 30,000th step. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

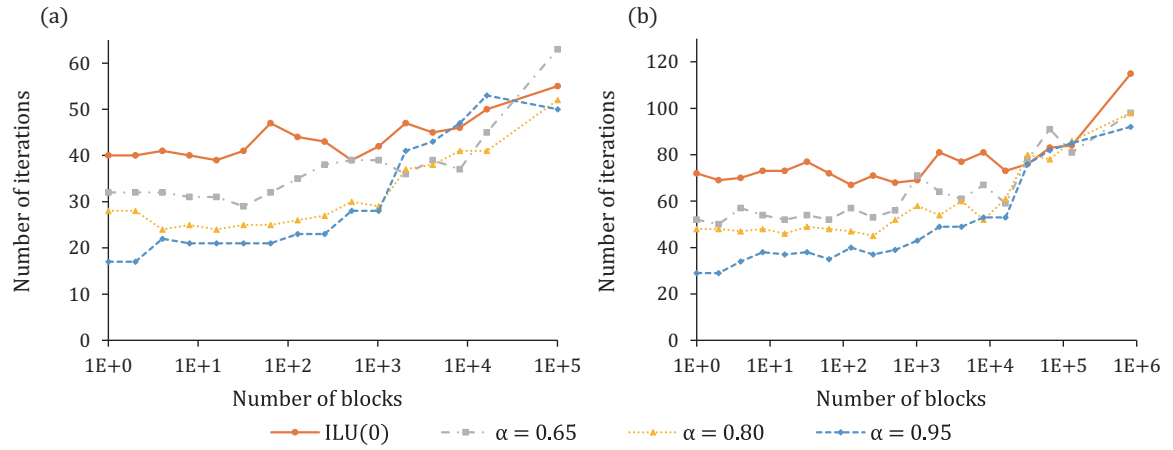

Figure A.9: Number of blocks N and number of iterations with RMILU(0) preconditioner at PIC 30,000th step. (a) $59 \times 59 \times 29$ grid ($h = 0.001$) and (b) $119 \times 119 \times 59$ grid ($h = 0.0005$).

Appendix B

Performance of GPU Implementation vs. Theoretical Peak

In this appendix, we compare the performance of the GPU implementation shown in Chapter 5 with the theoretical peak. For comparison, we use the performance of the PBiCGSTAB iterative loop, which has the largest impact on the computation time, executed on GP100. The time required for data transfer between the CPU and GPU, data reordering, and PMILU(0) factorization are not included in the results shown here.

Table B.1 shows the number of floating-point operations and data requirements for the PBiCGSTAB iterative loop on a GP100 with the specifications shown in Table 5.1. The FLOP time (single/double peak) in the table is the number of floating point operations multiplied by the reciprocal of the peak FLOPS for each precision. Peak performance (mixed) is the sum of the number of single and double precision floating point operations divided by the sum of the peak FLOP times. Similarly, Performance (mixed) is the sum of the number of floating-point operations divided by the Elapsed time. The ratio of that to Peak performance (mixed) is shown as Rate of peak performance. For all test problem sizes, the actual execution time is far from it estimated from the peak operation performance alone, and is not compute bound. By contrast, the data transfer rate reaches a value close to the peak as the size of the test problem increases. Therefore, the performance of loading and storing of data on the GPU determines the computation speed.

Table B.1: Execution of PBiCGSTAB iteration loop on GP100.

	$59 \times 59 \times 29$ grid	$119 \times 119 \times 59$ grid	$239 \times 239 \times 119$ grid
Elapsed time	1.709×10^{-2} s	7.413×10^{-2} s	7.142×10^{-1} s
Floating point operation (single)	152,206,068	1,611,145,728	18,986,442,552
FLOP time (single peak)	1.478×10^{-5} s	1.564×10^{-4} s	1.843×10^{-3} s
Floating point operation (double)	222,227,706	2,263,779,586	26,490,871,678
FLOP time (double peak)	4.274×10^{-5} s	4.353×10^{-4} s	5.094×10^{-3} s
Peak performance (mixed)	6510 GFLOPS	6549 GFLOPS	6556 GFLOPS
Performance (mixed)	21.91 GFLOPS	52.27 GFLOPS	63.68 GFLOPS
Rate of peak performance	0.3366%	0.7982%	0.9713%
Memory transfer FP32	257,572,580	2,685,984,880	31,618,630,120
Memory transfer FP64	197,098,761	2,006,238,531	23,409,363,999
Memory transfer integer	171,931,852	1,712,842,902	20,050,555,482
Required data	192.8 GB/s	453.9 GB/s	551.6 GB/s
Rate of peak streaming bandwidth	26.89%	63.30%	76.93%

Appendix C

Avoiding Indirect References Using Grid Structures

In Chapter 5, we showed an implementation using indirect references based on a sparse matrix format. BRB ordering is based on the fact that the grid structure is known, and by using it, indirect references can be avoided. In this appendix, we show an example of storing the data for our test problem in an array without indirect references.

Listing C.1 shows an example of rewriting the forward substitution for the red block shown in Listing 5.2 to get the data in the array without indirect references.

The `nxblock` and `nyblock` are the number of blocks along each axial direction, x and y, respectively, and are known from the grid structure. The `nxnode` and `nynode` are the number of nodes along the x and y axes, respectively, and are also known from the grid structure. From these values, the block indices `ibx`, `iby` and `ibz` in each axis direction of the block under calculation are first obtained. Next, the number of nodes `ixmax` and `iymax` in each axial direction of the block under calculation is calculated, which is used to calculate the position of the data to be used in the array `x`.

The iterations of the proposed method are bandwidth-bound, as shown in Appendix B. Therefore, reducing the amount of loading from the integer array, which accounts for 20% of the memory transfer, has the potential to speed up the computation.

However, in arrays targeted for indirect reference avoiding, the data is block-ordered for parallelization and stored in such a way that coalesced access is achieved when computing in block parallel. This makes the code to calculate the position of the data in the array complicated. Thus, there is also the possibility that the increase in time due to the additional operations outweighs the decrease in time due to the omission of loading the integer array for indirect references. The computation time for the forward substitution for the red block, which is rewritten to Listing 5.2 and the other conditions are the same as in Chapter 5, is 40% larger for test question size (a) and 4% larger for (b), while it is 8% smaller for (c).

The forward substitution for the black block, the backward substitution for the red block, and SpMV require a reference to `x` corresponding to a node contained in a block other than the block that contains itself. Therefore, the calculation to find the location of `x` to be loaded is more complicated. The computation time for theforward substitution for the black block, which finds the position of `x` by the same method as for the red block, is 65% larger for (a), 26% larger for (b), and 10% larger for (c).

For both of the two types of forward substitutions, the time advantage of loading the `x` position data from the array over computing position becomes smaller as the problem size and the number of nodes in the block increase. Therefore, optimizing this indirect-reference omission method for large problem sizes may result in shorter solution times.

Listing C.1: Example of revising Listing 5.2 to no indirect reference

```
do iblock = 1, lastblock
  x(iblock+1) = real(y(iblock+1))
  rownum = nblock * (iend(iblock) - istart(iblock)) + iblock
  ibz = int(iblock/(nxblock*nyblock/2.0) &
         - (1.0/(nxblock*nyblock/1.5))) + 1
  if(mod(ibz,2).ne.0) then
    iby = (int((iblock-ceiling((ibz-1)*(nxblock*nyblock/2.0))) &
           /(nxblock/2.0)-(1.0/(nxblock/1.5)))+1)
  else
    iby = (ceiling((iblock-ceiling((ibz-1)*(nxblock*nyblock/2.0))) &
           /(nxblock/2.0)))
  endif
  if(mod(ibz,2).ne.0)then
    ibx = ceiling((iby-1)*(nxblock/2.0))
  else
    ibx = int((iby-1)*(nxblock/2.0))
  endif
  ibx = (iblock - ceiling((ibz-1)*(nxblock*nyblock/2.0))  - ibx &
            )*2 - mod(ibz+iby+1,2) * 1
  iymax = nynode/nyblock
  if((mod(nynode,nyblock)-iby+1).gt.0) then
    iymax = iymax + 1
  endif
  ixmax = nxnode/nxblock
  if((mod(nxnode,nxblock)-ibx+1).gt.0) then
    ixmax = ixmax + 1
  endif

  do i = (iblock + nblock), rownum, nblock
    s = real(y(i+1))
    k = jl(i)
    if((i-nblock*iymax*ixmax).gt.0) then
      p0 = i-nblock*iymax*ixmax
    else
      p0 = 0
    endif
    s = s-alr(k)*x(p0+1)
    if((i-nblock*ixmax).gt.0) then
      p1 = i-nblock*ixmax
    else
      p1 = 0
    endif
    s = s-alr(k+tempnblock) *x(p1+1)
    if((i-nblock).gt.0) then
      p2 = i-nblock
```

```
    else
      p2 = 0
    endif
    s = s-alr(k+2*tempnblock)*x(p2+1)
    x(i+1) = s
  enddo
enddo
```