Electronic Preprint for Journal of Information Processing Vol.27

Regular Paper

Enhancement of Algebraic Block Multi-Color Ordering for ILU Preconditioning and Its Performance Evaluation in Preconditioned GMRES Solver

Senxi Li[1,a)] Takeshi Iwashita[2,b)] Takeshi Fukaya[1,c)]

Received: April 13, 2018, Accepted: May 26, 2018

Abstract: Algebraic block multi-color ordering is known as a parallelization method for a sparse triangular solver. In the previous work, we confirmed the effectiveness of the method in a multi-threaded ICCG solver for a linear system with a symmetric coefficient matrix. In this study, we enhance the method so as to deal with an unsymmetric coefficient matrix. We develop a multi-threaded ILU-GMRES solver based on the enhanced method and evaluate its performance in terms of both the runtime and the number of iterations.

Keywords: linear solver, preconditioner, multi threading, multi color ordering

1. Introduction

A preconditioned Krylov subspace solver is widely used for a linear system of equations arising in various numerical simulations such as finite element analyses [1]. When the coefficient matrix of the linear system is unsymmetric, a preconditioned GMRES (Generalized Minimal RESidual) solver [2] is one of the most popular methods to solve it. In each iteration of the method, the residual norm is monotonically reduced, but memory space used is enlarged. Therefore, in practical simulations, "restart" is commonly applied to the GMRES solver. Generally, GMRES(m) means the GMRES method with restarting every m iterations.

In this paper, we focus on ILU (more precisely ILU(0)) preconditioned GMRES(m) method. The ILU (factorization) preconditioning is one of the well known preconditioning techniques applied to an unsymmetric linear system of equations and making it more computationally feasible [3]. Moreover, the same procedure as its preconditioning step is used in the context of multigrid method as an ILU smoother.

While there are many parallelization techniques for ILU preconditioning, they can be roughly classified into two kinds; domain decomposition type methods and parallel orderings [4], [5]. In this paper, we discuss the latter one. A parallel ordering technique reorders the unknowns to change the nonzero element pattern of the coefficient matrix to an appropriate form for parallel processing. Initially, parallel orderings were investigated in the context of a structured grid analysis [6], [7]. In the analysis, grid-points are reordered based on their geometrical information. Red-black, multi-color, block multi-color, dissection, and domain decomposition orderings are typical parallel ordering techniques. Among various parallel orderings, multi-color ordering is the most popular one, and block multi-color ordering is the enhanced version of it. In multi-color ordering, an increase in the number of colors generally improves the convergence of ILU preconditioned iterative solver, but it also increases the number of synchronization points in the parallelized preconditioning step (forward and backward substitutions) [8]. In block multi-color ordering, the grid-points are divided into multiple blocks to which multi-color ordering is applied. The blocking of grid-points improves the convergence rate without increasing the number of synchronizations, and also improves the cache hit ratio. Consequently, the block multi-color ordering shows better parallel performance than the multi-color ordering [9], [10].

Considering the advantages of the block multi-color ordering method, Iwashita et al. developed its algebraic version that can be applied to a general linear system arising in unstructured problems [11]. The algebraic block multi-color ordering (ABMC) method has discovered its effectiveness in a multi-threaded ICCG (Incomplete Cholesky Conjugate Gradient) solver [12] and has also been used for high performance implementation of HPCG benchmark programs [13]. In these research activities, this method has been mainly discussed in symmetric coefficient matrix cases. In this paper, we introduce the enhancement of the ABMC method for ILU preconditioning that is used for a linear system with an unsymmetric coefficient matrix. We examine the enhanced version on recent multi-core and many-core processors and confirm its effectiveness in comparison with multi-color ordering, which is the most standard parallel ordering technique.

This paper is outlined as follows. Section 2 gives a brief explanation for the basics of ILU(0) preconditioning and GMRES method. Section 3 introduces the algebraic block multi-color ordering method for a linear system with an unsymmetric coeffi-

[1] Graduate School of Information Science and Technology, Hokkaido University, Sapporo, 060–0814 Japan
[2] Information Initiative Center, Hokkaido University, Sapporo, Hokkaido 060–0811, Japan
[a)] lisenxi@eis.hokudai.ac.jp
[b)] iwashita@iic.hokudai.ac.jp
[c)] fukaya@iic.hokudai.ac.jp

© 2019 Information Processing Society of Japan

cient matrix. Section 4 gives the results of numerical tests. Section 5 describes the related work of parallel ILU preconditioning. Section 6 gives a summary of the paper.

2. ILU Preconditioned GMRES Method

2.1 Basic GMRES Algorithm

The generalized minimal residual method (GMRES) is a projection method based on Krylov subspace to solve a linear system of equations. In this paper, we focus on a linear system with an n by n unsymmetric coefficient matrix as follows:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}. \tag{1}$$

The GMRES method finds the approximate solution vector $\tilde{\boldsymbol{x}}_s \in \boldsymbol{x}_0 + K_s$ which minimizes the 2-norm of the residual $\boldsymbol{r}_s = \boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}_s$ in the s-th iteration, where $\boldsymbol{x}_0$ denotes an arbitrary initial guess and K_s is the s-dimensional Krylov subspace. The space K_s is given by

$$K_s := \mathrm{Span}(\boldsymbol{r}_0,\ \boldsymbol{A}\boldsymbol{r}_0,\ \boldsymbol{A}^2\boldsymbol{r}_0,\ \boldsymbol{A}^3\boldsymbol{r}_0,\ \ldots,\ \boldsymbol{A}^{s-1}\boldsymbol{r}_0), \tag{2}$$

where $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$. In the original GMRES algorithm, memory space proportional to the iteration count is required. Namely, in every iteration, the algorithm requires additional memory space. Accordingly, it is practically difficult to use the original GMRES algorithm when such a linear system should be solved within a number of iterations. Consequently, a restarting technique is often used for practical problems. Here, we denote the GMRES method with restarting within every m iterations by GMRES(m). In GMRES(m), after m-times iterations, we start again the GMRES iteration with the initial guess $\boldsymbol{x}_0 = \boldsymbol{x}_m$. Although GMRES($m$) cannot minimize the norm of $\boldsymbol{r}_s$ when $s > m$, it monotonically reduces the residual norm.

2.2 ILU(0) Preconditioning

Lack of robustness is a widely recognized weakness of iterative solvers, when they are compared with a direct solver. Both efficiency and robustness of iterative methods can be improved by using preconditioning. The first step in preconditioning is to find a preconditioning matrix $\boldsymbol{M}$, which is also called a preconditioner. The preconditioner can be defined in many ways. Practically, $\boldsymbol{M}$ should be an approximation of $\boldsymbol{A}$, and the linear system with the coefficient matrix $\boldsymbol{M}$ for the preconditioning step must be easily solved compared to the original linear system (1).

In our research, we use a general left preconditioning which is written in a form of

$$\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b}.$$

We here focus on ILU(0) preconditioning. In ILU preconditioning, the coefficient matrix is incompletely factorized as follows:

$$\boldsymbol{A} \simeq \boldsymbol{L}\boldsymbol{U},$$

where $\boldsymbol{L}$ and $\boldsymbol{U}$ are lower and upper triangular matrices, respectively. In ILU(0) preconditioning, no fill-ins are permitted during the factorization process. In other words, $\boldsymbol{L}/\boldsymbol{U}$ has the same nonzero pattern as that of the lower or upper triangular part of $\boldsymbol{A}$.

When applying the GMRES method to ILU(0) preconditioned linear system, an operation of $\boldsymbol{w} = \boldsymbol{M}^{-1}\boldsymbol{z}$ or some similar operations are computed at each step, where $\boldsymbol{z}$ is a given vector. The operation is done through the following forward and backward substitutions:

$$\boldsymbol{t} = \boldsymbol{L}^{-1}\boldsymbol{z} \tag{3}$$

and

$$\boldsymbol{w} = \boldsymbol{U}^{-1}\boldsymbol{t}. \tag{4}$$

Algorithm 1 shows the procedure of ILU(0)-GMRES(m) method.

Algorithm 1 ILU(0)-GMRES(m) method

1: Compute $(\boldsymbol{LU})\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$, $\boldsymbol{v}_1 = \boldsymbol{r}_0/\|\boldsymbol{r}_0\|$.
2: **for** $i = 1$ to m **do**
3: Solve $(\boldsymbol{LU})\boldsymbol{w} = \boldsymbol{A}\boldsymbol{v}_i$
4: Genearte Hessenberg matrix $\boldsymbol{H}_i$ and vector $\boldsymbol{v}_i$ with size i using Arnoldi algorithm starting with $\boldsymbol{v}_1$.
5: Compute $\boldsymbol{y}_i = \mathrm{argmin}\| \|\boldsymbol{r}_0\|\boldsymbol{e}_1 - \boldsymbol{H}_i\boldsymbol{y}_i \|$
6: Update $\boldsymbol{x}_i = \boldsymbol{x}_0 + [\boldsymbol{v}_1\, \boldsymbol{v}_2\, \cdots\, \boldsymbol{v}_i]\boldsymbol{y}_i$ and if the relative residual norm of $\boldsymbol{x}_i$ is small enough, quit.
7: **end for**
8: If converges then stop, otherwise set $\boldsymbol{x}_0 = \boldsymbol{x}_m$ and go to 1.

3. Algebraic Block Multi-Color Ordering Method for Unsymmetric Matrices

In the paper, we investigate multi-threaded parallelization of ILU(0)-GMRES method. Most of the calculations involved in the method can be directly parallelized, including inner product, matrix-vector multiplication and vector updating. However, the forward and backward substitutions (preconditioning steps) cannot be straightforwardly parallelized due to its data dependency and inherent sequentiality. In the present research, we use reordering techniques for parallelization of the ILU(0) preconditioning step.

3.1 Multi-Color Ordering

Multi-Color ordering is the most popular parallel ordering technique. We here consider the non-directed graph $G(V, E)$ corresponding to the coefficient matrix $\boldsymbol{A}$. The graph involves n nodes and each edge corresponds to a nonzero entry of $\boldsymbol{A}$. That is, $E(i, j)$ exists if and only if a_{ij} or a_{ji} is not zero, where i, j are two unknowns and a_{ij} is the i-row j-column element in $\boldsymbol{A}$. In C-color ordering, the set of nodes V is divided into C subsets $G_1, G_2, \ldots, G_C$ with C colors, where the nodes in each color should be mutually independent. While there are a couple of strategies for multi-coloring, we here focus on the greedy algorithm that is the simplest but practically an effective one.

The greedy algorithm follows the greedy choice property. At each step of coloring nodes with color c, one node is selected as the seed. Once the seed is determined, all the other uncolored nodes should be scanned following the original order of the nodes. Each node is added into G_c if it has no data relationship with all existing nodes in G_c. This process should be repeated until all uncolored nodes are checked and all satisfactory nodes

© 2019 Information Processing Society of Japan

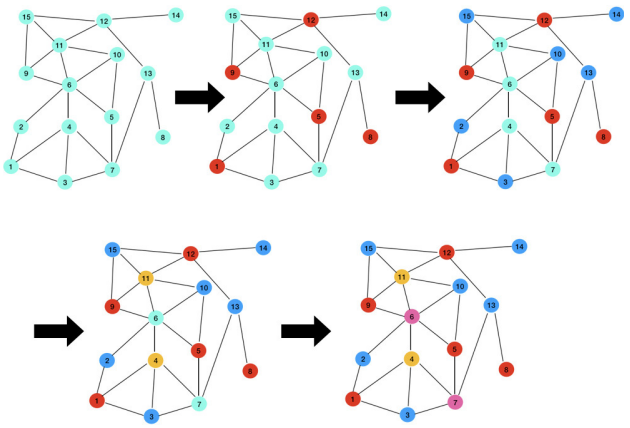


Fig. 1 multi-coloring of a directed graph with 15 nodes.

should be added into G_c before painting with color $c+1$.

Figure 1 shows an example of the coloring procedure. Following the original order of the nodes, the order from $1, 2, \ldots$ to 15, the node with the lowest number, node 1 is selected as the seed for G_1. All the other unpainted nodes are checked following the original order. Considering the relationship with node 1, node 5 is the next one assigned to G_1. We continue the procedure to find the node that does not have relationship with the nodes already assigned to G_1. When such a node is found, it is added to G_1. In the graph, nodes 5, 8, 9 and 12 are added to G_1. Finally, the nodes in the graph are divided into 4 subsets (colors).

In the multi-color ordering technique, the unknowns of the linear system to be solved are reordered following the order of color. Because the unknowns with an identical color have no relationship, they can be processed in parallel. That is, the reordered matrix has diagonal matrix forms in its block diagonal part. Accordingly, the forward and backward substitutions can be parallelized within each color. However, thread synchronization is required between different colors.

In the application of the multi-color ordering to an ILU/IC preconditioned iterative solver, an increase in the number of colors usually results in the reduction of the iteration count for convergence. However, it also entails a reduction of degree of parallelism. Moreover, because the nodes with the same color tend to be far away from each other, efficient use of data cache is often prevented due to lower data access locality. Therefore, block multi-color ordering method has been proposed to mitigate the above problems in the multi-color ordering. In the method, blocking of unknowns (nodes) leads to an improvement in convergence without increasing the number of synchronization points and improves the data access locality in multi-threaded substitutions.

3.2 Algebraic Block Multi-Color Ordering for Unsymmetric Matrices

Algebraic block multi-coloring strategy is an extension of multi-coloring method. Nodes of the matrix are first divided into blocks, then we color the blocks instead of coloring nodes. Since blocks with the same color are independent of each other, the forward and backward substitutions can be parallelized among blocks and nodes inside the block should be computed sequentially.

3.2.1 Blocking method

One of the main issues in the method is how to generate the blocks. Block generation is based on the information from the coefficient matrix. Suppose the block size is set to be b, we generate $n_b = \lceil n/b \rceil$ number of blocks. Consider the directed graph $\bar{G}(V, \bar{E})$ derived from A. In $\bar{G}$, unlike G, a_{ij} and a_{ji} are differently treated. When a_{ij} is non-zero, there is an arrow from i to j. If both a_{ij} and a_{ji} are non-zero, two (directed) edges exist between the nodes i and j. In the technique, we aim to divide V into n_b subsets $V_1, V_2, \cdots, V_{n_b}$, where $|V_i| = b$, $(i = 1, 2, \ldots, n_b)$ and $V_i \bigcap V_j = \emptyset$ if $i \neq j$. Algorithm 2 is our proposed method for block generating.

Algorithm 2 Blocking method

```
1:  i=1;  j=1;
2:  while i < n_b do
3:      Select a seed s for V_i according to the seed selection policy.
4:      V_i ← V_i ∪ s;  V ← V \ s
5:      while j < b do
6:          if No node is found to satisfy the node selection policy then
7:              Select a node v in V according to the seed selection policy.
8:          else
9:              Pick one node v in V according to the node selection policy.
10:         end if
11:         V_i ← V_i ∪ {v}; V ← V \ v
12:         j ← j + 1
13:     end while
14:     i ← i + 1; j = 1;
15: end while
```

In the present research, we use a simple seed selection policy in the algorithm. That is, we choose the node with the smallest number in the original order in V. For the node selection policy in *line* 6, we provide the following five methods:

- *Method* 1: Pick the node with the minimal number in V.
- *Method* 2: Pick the node connected to the seed in V.
- *Method* 3: Pick the node with the most tightly connected to the seed in V. That is, pick the node v with the largest value of $|a_{sv}| + |a_{vs}|$, where s denotes the seed.
- *Method* 4: Pick the node from V with the maximum number of edges connected to existing nodes in the current subset V_i.
- *Method* 5: Basically use *Method* 4. If multiple candidates are found, pick the node with the largest value of $\sum_{i \in V_i} |a_{iv}| + |a_{vi}|$ among them.

We, here, review the above five blocking methods. *Method* 1 is the easiest and lightest method for implementation and execution. However, it is expected to work when the original order of nodes (unknowns) is sufficiently good. For example, when the nodes have been already reordered by using RCM ordering, the blocking method can be satisfactory.

In *Method* 2 and 3, a data structure of tree graph with the seed is considered. When the nodes in a block are more strongly connected to each other, the data access locality in substitutions and

© 2019 Information Processing Society of Japan

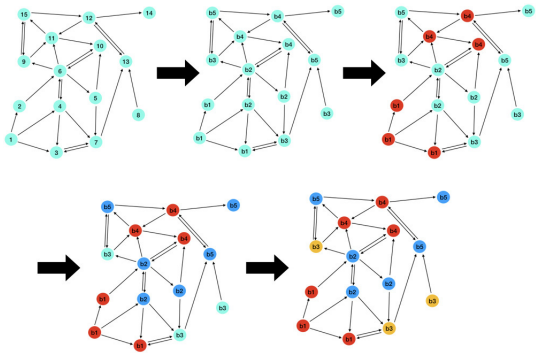


Fig. 2 ABMC with block size = 3, *Method* 1.

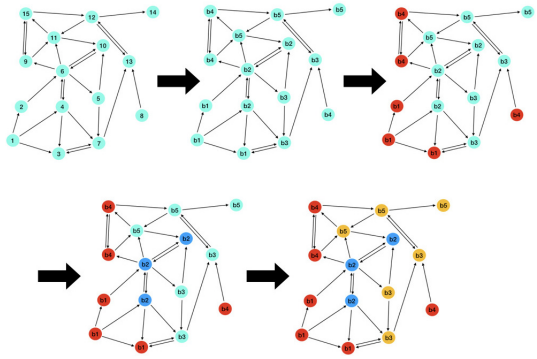


Fig. 3 ABMC with block size = 3, *Method* 5.

sparse matrix vector multiplication is increased. Moreover, it is also expected that the convergence rate is improved. Consequently, we expect the increased cache hit ratio and convergence rate by using *Method* 2 and 3. When we compare both methods, we expect a better convergence rate of *Method* 3 because of the consideration of the impact of each edge.

Method 4 and 5 are enhanced versions of *Method* 2 and 3. In these methods, the strategy for improving the data access locality and the convergence is more precisely implemented. That is, the effect of the nodes which have been already assigned to the block is considered. However, the implementation of the methods is relatively complicated.

3.2.2 Coloring of blocks

After successfully blocking the unknowns, similar to the multi coloring approach, blocks need to be colored and reordered in order to obtain parallelism. The process of coloring blocks is logically equivalent to the method for nodes (unknowns). We, here, consider the adjacency matrix of the blocks, denoted by $\boldsymbol{T}$, with dimension of n_b. The matrix $\boldsymbol{T}$ shows the connecting relationship between blocks, where the k-th row and l-th column, t_{kl}, of $\boldsymbol{T}$ is given by

$$t_{kl} = \begin{cases} 1 & \text{if } \exists i \in V_k,\ \exists j \in V_l \text{ s.t. } a_{ij} \neq 0 \ \vee \ a_{ji} \neq 0 \\ 0 & \text{otherwise} \end{cases}. \tag{5}$$

By applying a coloring technique to the nodes of the non-directed graph corresponding to $\boldsymbol{T}$, we can complete the coloring process for the blocks. Notice that the use of $\boldsymbol{T}$ is not always necessary in the implementation of the block coloring process. In this research, we also use the greedy algorithm shown in Section 3.1. **Figures 2** and **3** show examples of blocking and coloring unknowns. It can be easily discovered that by different blocking methods the generated blocks falls into distinct shapes, resulting in various coloring patterns and hence various reordered coefficient matrices.

After the blocking and coloring processes, the unknowns are reordered following of the order of color. It is noted that the nodes in an identical block should be contiguously ordered. In the technique, the order of blocks with the same color and the order of the nodes inside a block can be arbitrary. In our implementation, we use the order that is naturally obtained in the blocking method for these orders.

3.3 Parallelization of Triangualr Solver

We, here, denote the reordered linear system by $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}}$. When the (algebraic) block multi-color ordering is applied to the original coefficient matrix, the resulting matrix has the following form:

$$\tilde{\boldsymbol{A}} = \begin{pmatrix} \tilde{\boldsymbol{A}}_{1,1} & \tilde{\boldsymbol{A}}_{1,2} & \dots & \tilde{\boldsymbol{A}}_{1,C} \\ \tilde{\boldsymbol{A}}_{2,1} & \tilde{\boldsymbol{A}}_{2,2} & \dots & \tilde{\boldsymbol{A}}_{2,C} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{\boldsymbol{A}}_{C,1} & \tilde{\boldsymbol{A}}_{C,2} & \dots & \tilde{\boldsymbol{A}}_{C,C} \end{pmatrix}, \tag{6}$$

and

$$\tilde{\boldsymbol{A}}_{c,c} = \begin{pmatrix} \tilde{\boldsymbol{B}}_c^1 & & & \boldsymbol{0} \\ & \tilde{\boldsymbol{B}}_c^2 & & \\ & & \ddots & \\ \boldsymbol{0} & & & \tilde{\boldsymbol{B}}_c^{n(c)} \end{pmatrix}, \tag{7}$$

where $n(c)$ is the number of blocks assigned to color c and $\tilde{\boldsymbol{B}}_c^k$ is the b by b matrix corresponding to the unknowns in the k-th block with color c, which we denote by b_c^k.

In ILU(0) preconditioning, the matrices for preconditioning, $\tilde{\boldsymbol{L}}$ and $\tilde{\boldsymbol{U}}$ that are obtained from the incomplete LU factorization of $\tilde{\boldsymbol{A}}$ have the same non-zero element pattern as the lower or upper triangular part of $\tilde{\boldsymbol{A}}$, respectively. Therefore, $\tilde{\boldsymbol{L}}$ and $\tilde{\boldsymbol{U}}$ are written as

$$\tilde{\boldsymbol{L}} = \begin{pmatrix} \tilde{\boldsymbol{L}}_{1,1} & & & \\ \tilde{\boldsymbol{L}}_{2,1} & \tilde{\boldsymbol{L}}_{2,2} & & \boldsymbol{0} \\ \vdots & \vdots & \ddots & \\ \tilde{\boldsymbol{L}}_{C,1} & \tilde{\boldsymbol{L}}_{C,2} & \dots & \tilde{\boldsymbol{L}}_{C,C} \end{pmatrix}, \tag{8}$$

$$\tilde{\boldsymbol{U}} = \begin{pmatrix} \tilde{\boldsymbol{U}}_{1,1} & \dots & \tilde{\boldsymbol{U}}_{1,C-1} & \tilde{\boldsymbol{U}}_{1,C} \\ & \ddots & \vdots & \vdots \\ \boldsymbol{0} & & \tilde{\boldsymbol{U}}_{C-1,C-1} & \tilde{\boldsymbol{U}}_{C-1,C} \\ & & & \tilde{\boldsymbol{U}}_{C,C} \end{pmatrix}, \tag{9}$$

$$\tilde{\boldsymbol{L}}_{c,c} = \begin{pmatrix} \tilde{\boldsymbol{L}}_c^1 & & & \boldsymbol{0} \\ & \tilde{\boldsymbol{L}}_c^2 & & \\ & & \ddots & \\ \boldsymbol{0} & & & \tilde{\boldsymbol{L}}_c^{n(c)} \end{pmatrix}, \tag{10}$$

$$\tilde{\boldsymbol{U}}_{c,c} = \begin{pmatrix} \tilde{\boldsymbol{U}}_c^1 & & & \boldsymbol{0} \\ & \tilde{\boldsymbol{U}}_c^2 & & \\ & & \ddots & \\ \boldsymbol{0} & & & \tilde{\boldsymbol{U}}_c^{n(c)} \end{pmatrix}, \tag{11}$$

where $\tilde{\boldsymbol{L}}_c^k$ and $\tilde{\boldsymbol{U}}_c^k$ is the b by b lower or upper triangular matrix corresponding to block b_c^k, respectively. From Eqs. (8) and (10),

© 2019 Information Processing Society of Japan

```
for c = 1 to C
  Thread Parallelization
   for k = 1 to n(c)
      Solve L̃_c^{k,k} t̃_c^k = q̃_c^k
   end for
  End Thread Parallelization
end for
```

Fig. 4 Pseudo code for parallelized forward substitution.

the forward substitution for color c is given by

$$\tilde{\boldsymbol{t}}_c = \tilde{\boldsymbol{L}}_{c,c}^{-1}\left(\tilde{\boldsymbol{z}}_c - \sum_{d=1}^{c-1}\tilde{\boldsymbol{L}}_{c,d}\tilde{\boldsymbol{t}}_d\right), \tag{12}$$

which has $n(c)$ degree of parallelism, where $\tilde{\boldsymbol{t}}_c$ and $\tilde{\boldsymbol{t}}_d$ are the segments of $\tilde{\boldsymbol{t}}$ corresponding to the color c and d, respectively. Consequently, the forward substitution can be multi-threaded in the block-wise manner in each color. **Figure 4** shows the pseudo code of the multi-threaded forward substitution, where $\tilde{\boldsymbol{q}}_c^k$ is the segment of the vector $\tilde{\boldsymbol{z}}_c - \sum_{d=1}^{c-1}\tilde{\boldsymbol{L}}_{c,d}\tilde{\boldsymbol{t}}_d$ corresponding to the block k. From Eq. (11), it is shown that the backward substitution can also be performed in parallel among blocks in each color.

4. Numerical Tests

4.1 Environment Setup

The program code was written in Fortran90 and parallelized with OpenMP instructions. Standard CRS format is used for the storage of the sparse test coefficient matrices. Four test problems were picked up from The SuiteSparse Matrix Collection (formally known as the University of Florida Sparse Matrix Collection). We selected relatively large coefficient matrices from the problem domains in which the linear system is often solved. The information of all four test matrices are shown in **Table 1**. The first two matrices arise in computational fluid dynamics problems, and the other two are from a circuit simulation problem and an electromagnetics problem, respectively. The right-hand side vector is given by a vector having all elements of 1 and the convergence criterion is set as the relative residual norm (2-norm) being less than 10^{-7}. The restart period of GMRES method is set to be 50.

Numerical tests were executed on two different types of nodes operated at the Academic Center for Computing and Media Studies, Kyoto University. One node is a computational node of Cray CS400 system, which is equipped with two Intel Xeon Broadwell multi-core processors and 128 GB memory. The processor has 18 computing cores and its operation frequency is 2.1 GHz. For this system, the program was compiled by Intel Fortran compiler version 17.0.2.174 with the options of -mcmodel=medium -shared-intel -qopenmp -xHost -O3 -ipo. In the numerical test, we use all 36 cores in the computational node.

The other node is a computational node of Cray XC40 system. The node has a Xeon Phi (KNL) many-core processor which is equipped with 68 cores. The computational node has 96 GB memory, while the many-core processor is equipped with 16 GB fast device memory. For this system, the program was compiled by Cray Fortran compiler version 8.5.8 with the options of -h omp. In the numerical test, we use all 68 cores of the computational node.

Table 1 Matrix Information Of Test Linear Systems From UF Matrix Collection.

Data set	Problem type	Dimension	# nonzero
Atmosmodl	Computational fluid dynamics	1,489,752	10,319,760
Atmosmodj	Computational fluid dynamics	1,270,432	8,814,880
Memchip	Circuit simulation	2,707,524	13,343,948
T2em	Electromagnetics	921,632	4,590,832

Table 2 Atmosmodl test results on Intel Xeon (Broadwell) processors.

(i) Multi-color GMRES on 36 threads

# iteration	Comp. time (s)	time/iteration (ms)
135	1.47	10.9

(ii) ABMC GMRES on 36 threads

Blocking method	Block size	# ite.	Comp. time (s)	time/ite. (ms)
1	16	76	0.795	10.4
	64	73	0.763	10.4
	256	63	0.735	11.6
2	16	75	0.855	11.4
	64	69	0.806	11.6
	256	63	0.739	11.7
3	16	75	0.906	12.0
	64	69	0.828	12.0
	256	63	0.736	11.6
4	16	71	0.904	12.7
	64	63	0.773	12.2
	256	57	0.735	12.9
5	16	71	0.782	11.0
	64	62	**0.647**	**10.4**
	256	**57**	0.748	13.1

Table 3 Atmosmodj test results on Intel Xeon (Broadwell) processors.

(i) Multi-color GMRES on 36 threads

# iteration	Comp. time (s)	time/iteration (ms)
379	4.56	12.0

(ii) ABMC GMRES on 36 threads

Blocking method	Block size	# ite.	Comp. time (s)	time/ite. (ms)
1	16	192	1.76	9.19
	64	141	1.29	9.18
	256	134	1.34	10.1
2	16	218	2.23	10.2
	64	142	1.39	9.81
	256	134	1.35	10.1
3	16	218	2.23	10.8
	64	142	1.41	9.91
	256	134	1.36	10.1
4	16	206	2.12	10.2
	64	138	1.32	9.57
	256	134	1.48	11.0
5	16	186	1.66	**8.93**
	64	137	**1.24**	9.12
	256	**132**	1.32	10.0

4.2 Numerical Results

In this subsection, we present test results of all given linear systems. All results are arranged in the following tables and several convergence behaviour figures of comparison between multi-color (MC) and algebraic block multi-color (ABMC) orderings are also given below.

4.2.1 Comparison Between MC and ABMC

Firstly, we discuss the main issue of this paper, that is, the comparison of the ABMC method with the conventional multi-color ordering technique. **Tables 2–5** list the number of iterations, the computational time, and the computational time per iteration measured in the numerical tests on Intel Xeon multi-core processors. **Tables 6–9** show the results of the numerical tests on an Intel Xeon Phi processor. **Figures 5** and **6** show the comparison

© 2019 Information Processing Society of Japan

Table 4 Memchip test results on Intel Xeon (Broadwell) processors.

(i) Multi-color GMRES on 36 threads

# iteration	Comp. time (s)	time/iteration (ms)
85	1.72	20.3

(ii) ABMC GMRES on 36 threads

Blocking method	Block size	# ite.	Comp. time (s)	time/ite. (ms)
1	16	76	1.46	**19.3**
	64	70	1.42	20.3
	256	**65**	**1.37**	21.0
2	16	79	1.75	22.2
	64	67	1.48	22.1
	256	67	1.41	21.1
3	16	79	1.80	22.8
	64	67	1.47	22.0
	256	67	1.42	21.3
4	16	89	1.96	22.0
	64	88	1.79	20.4
	256	88	1.84	21.0
5	16	89	1.92	21.5
	64	84	1.74	20.7
	256	88	1.83	20.8

Table 5 T2em test results on Intel Xeon (Broadwell) processors.

(i) Multi-color GMRES on 36 threads

# iteration	Comp. time (s)	time/iteration (ms)
4,081	25.9	6.35

(ii) ABMC GMRES on 36 threads

Blocking method	Block size	# ite.	Comp. time (s)	time/ite. (ms)
1	16	**2,299**	**14.5**	6.31
	64	3,206	19.6	6.13
	256	4,233	26.8	6.34
2	16	2,819	19.3	6.86
	64	3,659	25.3	6.93
	256	3,731	22.8	**6.12**
3	16	2,819	19.5	6.91
	64	3,659	24.2	6.63
	256	3,731	23.3	6.24
4	16	2,587	18.2	7.06
	64	3,717	25.4	6.84
	256	3,414	22.5	6.60
5	16	2,587	18.3	7.08
	64	3,717	25.0	6.74
	256	3,414	21.3	6.23

Table 6 Atmosmodl test results on Intel Xeon Phi processor.

(i) Multi-color GMRES on 68 threads

# iteration	Comp. time (s)	time/iteration (ms)
135	1.03	7.60

(ii) ABMC GMRES on 68 threads

Blocking method	Block size	# ite.	Comp. time (s)	time/ite. (ms)
1	16	76	0.594	7.82
	64	73	0.519	7.11
	256	63	**0.437**	**6.94**
2	16	75	0.771	10.2
	64	69	0.568	8.24
	256	63	0.477	7.57
3	16	75	0.793	10.5
	64	69	0.642	9.31
	256	63	0.452	7.17
4	16	71	0.817	11.5
	64	63	0.642	10.1
	256	57	0.534	9.28
5	16	71	0.518	7.30
	64	62	0.455	7.34
	256	**57**	0.594	10.4

Table 7 Atmosmodj test results on Intel Xeon Phi processor.

(i) Multi-color GMRES on 68 threads

# iteration	Comp. time (s)	time/iteration (ms)
379	2.41	6.36

(ii) ABMC GMRES on 68 threads

Blocking method	Block size	# ite.	Comp. time (s)	time/ite. (ms)
1	16	192	1.24	6.45
	64	141	0.953	6.76
	256	134	**0.862**	**6.43**
2	16	218	1.86	8.55
	64	142	1.03	7.32
	256	134	0.912	6.80
3	16	218	1.87	8.58
	64	142	1.04	7.37
	256	134	0.889	6.63
4	16	206	2.08	10.1
	64	138	0.958	6.94
	256	134	1.07	8.02
5	16	186	1.21	6.55
	64	137	0.970	7.08
	256	**132**	0.916	6.94

Table 8 Memchip test results on Intel Xeon Phi processor.

(i) Multi-color GMRES on 68 threads

# iteration	Comp. time (s)	time/iteration (ms)
85	1.40	16.4

(ii) ABMC GMRES on 68 threads

Blocking method	Block size	# ite.	Comp. time (s)	time/ite. (ms)
1	16	77	1.46	19.0
	64	71	1.30	18.3
	256	**65**	**1.16**	17.9
2	16	81	1.67	20.7
	64	67	1.28	19.1
	256	67	1.23	18.4
3	16	81	1.67	20.7
	64	67	1.25	18.7
	256	67	1.34	20.0
4	16	89	1.85	20.8
	64	89	1.71	19.2
	256	89	1.68	18.9
5	16	89	2.43	27.3
	64	85	1.57	18.5
	256	88	1.57	**17.8**

Table 9 T2em test results on Intel Xeon Phi processor.

(i) Multi-color GMRES on 68 threads

# iteration	Comp. time (s)	time/iteration (ms)
4,104	19.5	4.76

(ii) ABMC GMRES on 68 threads

Blocking method	Block size	# ite.	Comp. time (s)	time/ite. (ms)
1	16	**2,299**	**11.6**	5.06
	64	3,205	16.2	**5.05**
	256	4,236	22.1	5.23
2	16	2,818	15.4	5.48
	64	3,655	19.5	5.35
	256	3,725	19.6	5.27
3	16	2,818	15.6	5.56
	64	3,655	19.2	5.25
	256	3,725	19.8	5.33
4	16	2,587	15.5	6.00
	64	3,725	20.9	5.61
	256	3,416	18.9	5.54
5	16	2,587	15.7	6.09
	64	3,725	20.9	5.62
	256	3,416	19.0	5.58

© 2019 Information Processing Society of Japan

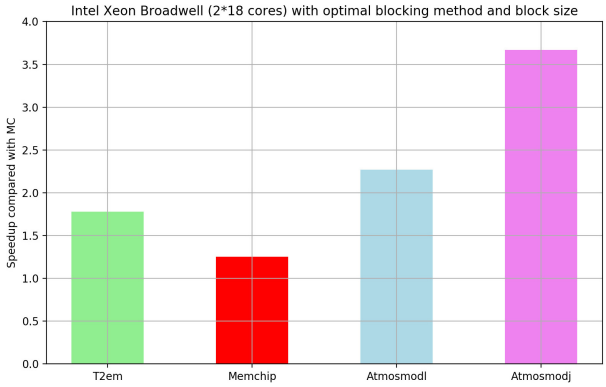


Fig. 5 Computational time compared with MC with optimal blocking method and block size on Intel Xeon Broadwell.

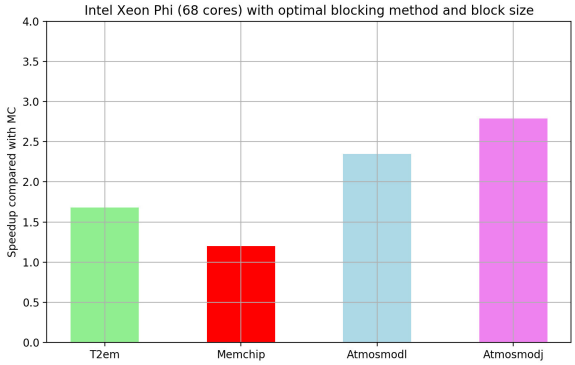


Fig. 6 Computational time compared with MC with optimal blocking method and block size on Intel Xeon Phi.

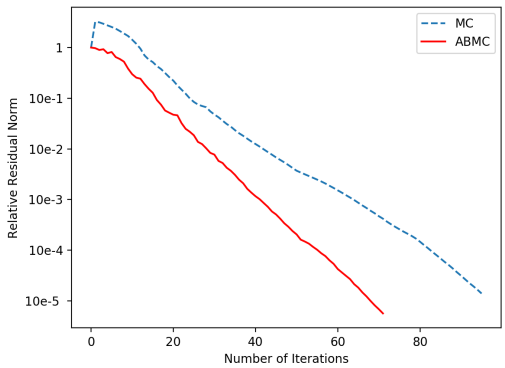


Fig. 7 Convergence behavior in Atomosmodl dataset test (MC and ABMC with block size = 256, *Method* 5).

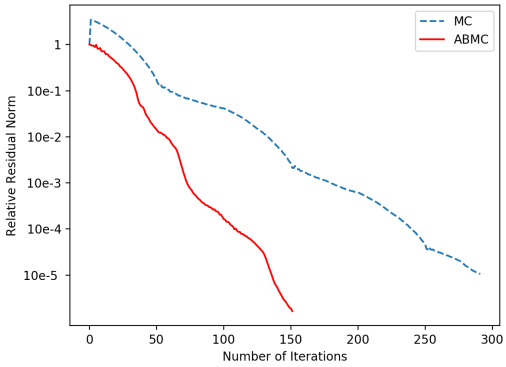


Fig. 8 Convergence behavior in Atomosmodj dataset test (MC and ABMC with block size = 256, *Method* 5).

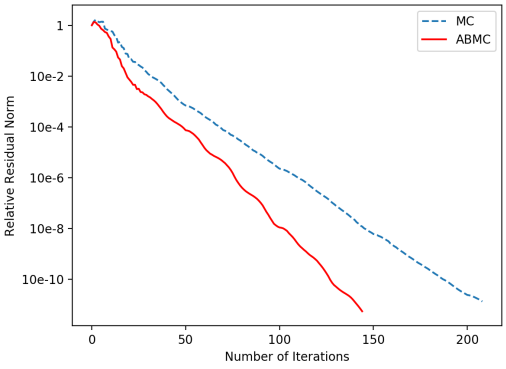


Fig. 9 Convergence behavior in Memchip dataset test (MC and ABMC with block size = 256, *Method* 1).

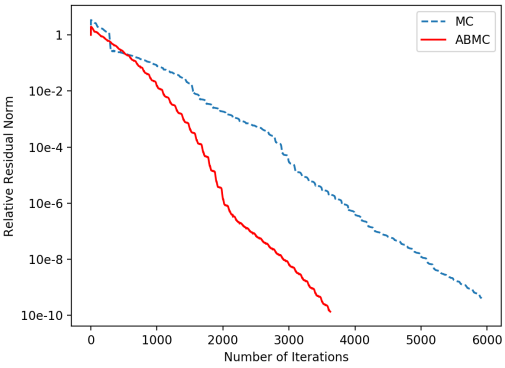


Fig. 10 Convergence behavior in T2em dataset test (MC and ABMC with block size = 16, *Method* 1).

of total computational time between ABMC and MC with optimal blocking method and block size on both systems. The numerical result indicates that the developed multi-threaded solver outperforms the solver based on multi-color (MC) ordering in all 8 cases (4 datasets × 2 systems) when the blocking method and the block size are properly set.

One of the main advantages of ABMC over MC is better convergence. **Figures 7–10** show the convergence behaviors of MC and ABMC solvers, which confirm the advantage of ABMC in the convergence rate. In the Atmosmodl and Atmosmodj dataset tests, the ABMC solver attains more than twice as fast convergence as the MC solver. The numerical test indicates that the blocking method can improve the convergence rate of the ILU(0)-GMRES solver.

Next, we examine the computational time in an iteration. On multi-core Xeon processors, the blocking method contributes to the reduction of the computational time for an iteration in all test cases. This can be ascribed to the structure of ABMC. Closer nodes tend to be assigned in the same block, which can lead to a potential high cache hit rate. In order to verify our assumption, we use Intel VTune Amplifier as the performance profiler to analyze the solver performance on Intel Xeon Broadwell processors with the chosen dataset Atmosmodj. Typically, we focus on the L3 cache miss rate. We use the following command and option to run the program: amplxe-cl -collect memory-access. The results show that MC gives a 0.06% L3 cache miss ratio, and ABMC leads to that of 0.03%. The results indicate that cache hit ratio is improved by ABMC. However, on a many-core Xeon Phi processor, the computational time per iteration is not reduced in three test cases. Although we selected relatively large datasets from the database, the test matrices are smaller than the size of the MC-

© 2019 Information Processing Society of Japan

DRAM (fast memory) in the processor. Accordingly, the computation of ILU-GMRES can be performed on the fast memory. Because the difference of the bandwidth between the cache memory and the fast memory is less significant than that between the cache and the main memory, the effect of the blocking to increase the cache hit ratio becomes relatively weak. Moreover, in the multi-threaded preconditioning step based on ABMC, the computation related to the diagonal block is not vectorized, though the computation with respect to off-diagonal blocks can be vectorized. Because the Xeon Phi processor is equipped with the wide (512 bit) SIMD instruction, it may affect the performance of ABMC. However, from the viewpoint of the total computational time, ABMC attains better performance than MC owing to the improved convergence even on a Xeon Phi processor.

4.2.2 Effect with Various Block Sizes and Different Blocking Methods

One of the tricky parts while applying ABMC is the choice of the block size. In general, it is expected that a larger block size results in better convergence. But, use of excessively large blocks compared with the dimension of the coefficient matrix causes an insufficient degree of parallelism and a decline in the convergence rate. In Atmosmodl, Atmosmodj, and Memchip dataset tests, it is confirmed that the larger block size results in the better convergence. In the relatively small size problem, T2em, the number of iterations significantly varies and the increase in the block size does not lead to the improvement of convergence.

Another worthwhile comparison is among the blocking methods. It is recognized that with different blocking methods, ABMC gains different effects in both iteration count and computational time. In Atmosmodj and Atmosmodl dataset tests, the best case is obtained with *Method* 5. While with the other two problems, *Method* 1 works better. As described in the previous section, *Method* 1 is efficiently workable if the original matrix is ordered well though it is the simplest method for implementation. Since matrices from the database are usually well ordered in advance, the test results show that *Method* 1 provides satisfactory performance for all matrices. Thus, for fair comparison, we randomize the order of nodes for one of the test matrices, T2em and focus on how these 5 blocking methods work with fixed block size of 16 on multicore Xeon processors. The result shows that the solver with *Method* 2 and 3 converges in around 2,400 iterations. When *Method* 4 and 5 are used, the solver converges in about 2,600 iterations. However, it converges in 2,680 iteration counts and 15.9 s in running time with *Method* 1 (without randomization converging in 2,299 counts and 14.5 s shown in Table 5). The result indicates that more sophisticated blocking method is expected to work for the coefficient matrix ill-ordered.

From all test results one can say that it is hard to determine the best blocking method and block size for varieties of problems. Development of auto tuning of the blocking method should be an important issue and remains our future work. However, in practical situations, we can suggest starting with a relatively small block size around 16 and blocking *Method* 1, since from our numerical results most cases with block size 16 and *Method* 1 gain better performance than MC (although one exception happens with Memchip on a Xeon Phi processor). If the speedup is not sufficient enough, one can try with larger block size. Usually better performance can be obtained with various block sizes if one sticks to *Method* 1, otherwise try with *Method* 5 and continue the block size adjustment until getting sufficient speedup.

5. Related works

This subsection gives a brief review of parallelization techniques for ILU, more precisely ILU(0) preconditioning. Since ILU preconditioning is a standard preconditioning technique, its parallelization method has been investigated for more than 20 years. We can see some early activities in Ref. [4]. In Ref. [4], the parallelization methods are classified into "Reordering" and "Preconditioning by domains". One of the simplest methods in the latter class is "localized" block Jacobi ILU preconditioning. In the technique, the block diagonal parts of the coefficient matrix are factorized and used for preconditioning. In this method, the preconditioning step (forward and backward substitutions) can be parallelized without process communication or thread synchronization. However, when the number of threads (or processes) is increased, the number of nonzero elements ignored in the factorization is also increased, which results in a decline in the preconditioning effect. One of remedies for this preconditioning effect is the overlapping technique. Although the overlapping entails an increase in operations, it often reduces the iteration counts for convergence. It strongly depends on the characteristics of the problem whether the (overlapped) block Jacobi ILU preconditioning attains a good parallel speedup. Chen et al. reports their recent implementation of the method on GPU in Ref. [14]. But, in general, the method is not suitable for the execution on a number of cores due to the convergence deterioration.

The reordering (parallel ordering) technique is a major parallelization method for ILU/IC preconditioning. One of the most important issues of the technique is in the trade-off problem between the parallelism and the convergence. The trade-off in the parallel ordering was discussed in many literatures including [6], [7], [8], [10], [15], [16], [17], [18]. A popular parallel ordering technique which has been often used in practical simulations gives a good compromise for the problem. Domain decomposition (dissection) ordering is one of well-known techniques. It gives a good convergence rate when the number of threads/processes is relatively small. Multi-color ordering is the most standard parallel ordering technique. When the relatively large number of colors, for example, more than 8, is used, it attains sufficiently good convergence. The block multi-color ordering is the enhanced version of the multi-color ordering. The blocking of the nodes (or unknowns) can improve the convergence without increasing the number of synchronization points in parallel execution.

When the unstructured problem is solved, the reordering process should be algebraically performed only using the information obtained from the coefficient matrix. A couple of heuristics are proposed for the parallel ordering. HID is one of algebraic versions of domain decomposition ordering [23]. The method has advantage in the communication in multi-process execution. Graph partitioning tools such as METIS [19] and SCOTCH [20] can be also used for a basis of domain decomposition ordering.

© 2019 Information Processing Society of Japan

For the application of multi-color ordering to unstructured analyses, the greedy algorithm is the most popular method. Moreover, Jones and Plassman's work [21] is also well-known. In the application of the multi-color ordering to certain types of problems, a relatively large number of colors are preferable. For this purpose, the algebraic multi-color ordering proposed in Ref. [22] can be used. As a recent work, Kawai et al. proposed a new heuristics for implementation of the multi-color ordering on a distributed parallel computer [24]. The enhancement of the block multi-color ordering for unstructured problems is reported in Ref. [11] and this paper for IC and ILU preconditioning, respectively.

Finally, we describe latest research activities on related techniques to ILU(0) preconditioning. Nakajima evaluated two types of domain decomposition procedures, LBJ (Localized Block Jacobi) and HID (Hierarchical Interface Decomposition) for ill-conditioned problems and proved that HID provides better efficiency and robustness than LBJ in Ref. [25]. Gupta introduced a blocking framework to improve the reliability and performance of ILU factorization-based preconditioners. The proposed blocking framework has been proved to lead to faster and more robust preconditioning in Ref. [27]. Bahi et al. proposed an efficient parallel implementation of GMRES for GPU clusters and verified the high data-parallel nature of GPUs in Ref. [26]. Chen et al. developed a group of ILU-family preconditioners on GPUs in Ref. [14], including ILU(0), ILU(K), ILUT and block-wise ILU(K).

The difference between our method and other related techniques is simply in the difference of algorithm. In the area of preconditioned iterative solvers, it is hard to determine the best method for a wide variety of problems. The comparison of the methods is greatly affected by the characteristics of the problem and the used computer. However, our algebraic version of block multi-color ordering shows a better performance than the standard multi-color ordering technique in numerical tests using matrices obtained from an well-known matrix database. Therefore, it can be expected to be one of candidates for parallelization of ILU preconditioning.

6. Conclusion

In this paper, we proposed an enhanced method of block multi-color ordering for multi-threading of ILU(0)-GMRES solver. Five blocking methods are introduced in consideration of the unsymmetric property of the coefficient matrix to aim for the improvement of convergence. Numerical tests using four matrices obtained from a matrix collection database were conducted on two types of computational nodes. One computational node is equipped with two multi-core Intel Xeon processors, and the other one is based on an Intel Xeon Phi processor. The developed multi-threaded solver based on algebraic block multi-color ordering outperforms the solver based on multi-color ordering in all test cases. Since the multi-color ordering is the most standard parallel ordering technique, it is shown that the developed block based method can be one of strong candidates for the parallelization method of ILU(0) preconditioning.

For future work, we would try to develop the auto tuning technique to select an appropriate blocking method and set an optimal block size. Moreover, we will study a parallel ordering method in which both thread parallelization and vectorization are efficiently utilized.

References

[1] Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and van der Vorst, H.: Templates for the solution of linear systems: Building blocks for iterative methods, SIAM, Philadelphia, PA (1994).

[2] Saad, Y.: *Iterative Methods for Sparse Linear Systems*, Second ed., SIAM, Philadelphia, PA (2003).

[3] van der Vorst, H.A.: Iterative Krylov Methods for Large Linear Systems, Cambridge University Press, UK (2003).

[4] Duff, I.S. and van der Vorst, H.A.: Developments and trend in the parallel solution of linear systems, *Parallel Comput.*, Vol.25, pp.1931–1970 (1999).

[5] Dongarra, J.J., Duff, I.S., Sorensen, D.C. and van der Vorst, H.A.: Solving Linear Systems on Vector and Shared Memory Computers, *SIAM*, Philadelphia, PA (1990).

[6] Duff, I.S. and Meurant, G.A.: The effect of ordering on preconditioned conjugate gradients, *BIT*, Vol.29, pp.635–657 (1989).

[7] Benzi, M., Joubert, W. and Mateescu, G.: Numerical experiments with parallel orderings for ILU preconditioners, *Electron. Trans. Numer. Anal.*, Vol.8, pp.88–114 (1999).

[8] Doi, S. and Washio, T.: Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorization, *Parallel Comput.*, Vol.25, pp.1995–2014 (1999).

[9] Iwashita, T. and Shimasaki, M.: Block red-black ordering: A new ordering strategy for parallelization of ICCG method, *Internat. J. Parallel Programming*, Vol.31, pp.55–75 (2003).

[10] Iwashita, T., Nakanishi, Y. and Shimasaki, M.: Comparison criteria for parallel orderings in ILU preconditioning, *SIAM J. Sci. Comput.*, Vol.26, pp.1234–1260 (2005).

[11] Iwashita, T., Nakashima, H. and Takahashi, Y.: Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method, *Proc. IPDPS2012*, pp.474–483 (2012).

[12] Semba, K., Tani, K., Yamada, T., Iwashita, T., Takahashi, Y. and Nakashima, H.: Parallel Performance of Multithreaded ICCG Solver Based on Algebraic Block Multicolor Ordering in Finite Element Electromagnetic Field Analyses, *IEEE Trans. Magn.*, Vol.49, pp.1581–1584 (2013).

[13] Park, J., Smelyanskiy, M., Vaidyanathan, K., Heinecke, A., Kalamkar, D.D., Liu, X., Patwary, M.M.A., Lu, Y. and Dubey, P.: Efficient Shared-Memory Implementation of High-Performance Conjugate Gradient Benchmark and Its Application to Unstructured Matrices, *Proc. SC14*, pp.945–955 (2014).

[14] Chen, Y., Tian, X., Liu, H., Chen, Z., Yang, B., Liao, W., Zhang, P., He, R. and Tang, M.: Parallel ILU preconditioners in GPU computation, *Soft Comput.* (2017).

[15] Doi, S.: On parallelism and convergence of incomplete LU factorizations, *Appl. Numer. Math.*, Vol.7, pp.417–436 (1991).

[16] Doi, S. and Lichnewsky, A.: Some parallel and vector implementations of preconditioned iterative methods on Cray-2, *Internat. J. High Speed Comput.*, Vol.2, pp.143–179 (1990).

[17] Doi, S. and Lichnewsky, A.: A graph-theory approach for analyzing the effects of ordering on ILU preconditioning, *INRIA Report*, No.1452, INRIA, France (June 1991).

[18] Eijkhout, V.: Analysis of parallel incomplete point factorizations, *Linear Algebra Appl.*, Vol.154–156, pp.723–740 (1991).

[19] Karypis Lab, available from ⟨http://glaros.dtc.umn.edu/gkhome/views/metis⟩

[20] SCOTCH, available from ⟨http://www.labri.fr/perso/pelegrin/scotch/⟩

[21] Jones, M.T. and Plassmann, P.E.: The efficient parallel iterative solution of large sparse linear systems, in Graph Theory and Sparse Matrix Computations, George, A., Gilberta, J.R. and Liu, J.W.H. (Eds.), IMA 56, Springer, Berlin, pp.229–245 (1994).

[22] Iwashita, T. and Shimasaki, M.: Algebraic multi-color ordering for parallelized ICCG solver in finite element analyses, *IEEE Trans. Magn.*, Vol.38, pp.429–432 (2002).

[23] Hénon, P. and Saad Y.: A Parallel Multistage ILU Factorization Based on a Hierarchical Graph Decomposition, *SIAM J. Sci. Comput.*, Vol.28, pp.2266–2293 (2006).

[24] Kawai, M., Ida, A. and Nakajima, K.: Hierarchical Parallelization of Multicoloring Algorithms for Block IC Preconditioners, *Proc. HPCC2017*, pp.138–145 (2017).

[25] Nakajima, K.: Parallel Iterative Solvers for Ill-conditioned Prob-

© 2019 Information Processing Society of Japan

lems with Heterogeneous Material Properties, *Procedia Computer Science*, Vol.80, pp.1635–1645, ISSN 1877-0509, DOI: https://doi.org/10.1016/j.procs.2016.05.498 (2016).
[26] Bahi, J.M., Couturier, R. and Khodja, L.Z.: Parallel GMRES implementation for solving sparse linear systems on GPU clusters, *Proc. 19th High Performance Computing Symposia* (*HPC '11*), Society for Computer Simulation International (2011).
[27] Gupta, A.: Enhancing Performance and Robustness of ILU Preconditioners by Blocking and Selective Transposition, *SIAM Journal on Scientific Computing 2017*, Vol.39, No.1, pp.A303–A332, DOI: 10.1137/15M1053256 (2017).



Senxi Li was born in 1994. He received his B.E. degree in Electrical and Computer Engineering from Joint Institute, Shanghai Jiao Tong University in 2016. He is currently a master student in the Graduate School of Information Science and Technology at Hokkaido University. His recearch interests include high performance computing and linear iterative solvers.



Takeshi Iwashita was born in 1971. He received his B.E., M.E., and Ph.D. degrees from Kyoto University in 1992, 1995, and 1998, respectively. In 1998–1999, he worked as a post-doctoral fellow of the JSPS project in the Graduate School of Engineering, Kyoto University. He moved to the Data Processing Center of the same university in 2000. In 2003–2014, he worked as an associate professor in the Academic Center for Computing and Media Studies, Kyoto University. He currently works as a professor in the Information Initiative Center, Hokkaido University. His research interests include high performance computing, linear iterative solver, and electromagnetic field analysis. He is a member of IEEE, SIAM, IPSJ, IEEJ, JSIAM, JSCES, and JSAEM.



Takeshi Fukaya was born in 1983. He received his B.E., M.E., and Ph.D. degrees from Nagoya University in 2007, 2009, and 2012, respectively. He worked in Kobe University and RIKEN AICS, and became an assistant professor in the Information Initiative Center, Hokkaido University in 2015. His major research interests include high performance computing and numerical linear algebra. He is a member of IPSJ, JSIAM.

© 2019 Information Processing Society of Japan