IPSJ Transactions on Mathematical Modeling and Its Applications Vol.9 No.2 1–8 (Aug. 2016)

Regular Paper

Performance Evaluation of Golub-Kahan-Lanczos Algorithm with Reorthogonalization by Classical Gram-Schmidt Algorithm and OpenMP

Masami Takata[1,a) Hiroyuki Ishigami[2,b) Kinji Kimura[2,c) Yuki Fujii[2] Hiroki Tanaka[2]
Yoshimasa Nakamura[2,d)

Received: September 1, 2015, Revised: October 23, 2015/November 27, 2015,
Accepted: December 9, 2015

Abstract: The Golub-Kahan-Lanczos algorithm with reorthogonalization (GKLR algorithm) is an algorithm for computing a subset of singular triplets of large-scale sparse matrices. The reorthogonalization tends to become a bottleneck of the execution time, as the iteration number of the GKLR algorithm increases. In this paper, OpenMP-based parallel implementation of the classical Gram-Schmidt algorithm with reorthogonalization (OMP-CGS2 algorithm) is introduced. The OMP-CGS2 algorithm has the advantage of data reusability and is expected to achieve higher performance of the reorthogonalization computations on shared-memory multi-core processors with large caches than the conventional reorthogonalization algorithms. Numerical experiments on shared-memory multi-core processors show that the OMP-CGS2 algorithm accelerates the GKLR algorithm more effectively for computing a subset of singular triplets of some matrices than the conventional reorthogonalization algorithms. In addition, we discuss the cache utilization in the OMP-CGS2 algorithm and a condition that the OMP-CGS2 algorithm achieves higher performance than the CGS2 algorithm.

Keywords: subset computation of singular pairs, Golub-Kahan-Lanczos algorithm with reorthogonalization, classical Gram-Schmidt algorithm with reorthogonalization, OpenMP, sharedmemory multi-core processing

1. Introduction

Let A be a real $m \times n$ matrix and $\mathrm{rank}(A) = r$ ($r \leq \min(m, n)$). Then A has singular values $\sigma_j \in \mathbb{R}$ such that $\sigma_1 \geq \cdots \geq \sigma_r > 0$ and their corresponding left and right singular vectors $\boldsymbol{u}_j \in \mathbb{R}^m$ and $\boldsymbol{v}_j \in \mathbb{R}^n$ ($1 \leq j \leq r$). A subset of singular triplets, i.e. the ℓ largest singular values $\sigma_1, \ldots, \sigma_\ell$ and their corresponding singular vectors, is often required in low-rank matrix approximation [16] and statistical processing such as principal component analysis and the least-squares method. In such applications, the target matrix is often large and sparse, and ℓ is often much smaller than both m and n. It is difficult to directly compute a subset of singular triplets of a large-scale sparse matrix because of the computational cost and need for large amounts of memory.

The Krylov subspace methods are better for such a computation. They transform the target matrix into a significantly smaller matrix than the target matrix and the singular values of the generated matrix sufficiently approximate a subset of singular values of the target matrix. The Golub-Kahan-Lanczos (GKL) algorithm [5], [6] is one of the Krylov subspace methods and generates approximate bidiagonal matrices from the target matrix. However, the GKL algorithm usually loses the orthogonality of the Krylov subspace because of the computational error. To improve the orthogonality, the GKL algorithm with reorthogonalization (GKLR algorithm) [1] employs the reorthogonalization process. Note that these algorithms are generally parallelized in terms of the Basic Linear Algebra Subprograms (BLAS) [2], such as the matrix multiplications and the matrix-vector multiplications, because they are iterative algorithms. In addition, we implement the bisection algorithm and the inverse iteration algorithm [12] for computing a subset of singular triplets of the approximate matrices generated by the GKLR algorithm.

Although the GKLR algorithm is stable because of the reorthogonalization, the reorthogonalization tends to become a bottleneck in terms of the computational cost and the execution time as the iteration number increases. However, since the reorthogonalization of the GKLR algorithm is mainly implemented using the matrix-vector multiplications, the reorthogonalization is not effectively accelerated in parallel processing and then the overall execution time of the GKLR algorithm is not effectively reduced.

In this paper, to accelerate the reorthogonalization of the GKLR algorithm more effectively in parallel processing, we present a parallel implementation of the classical Gram-Schmidt algorithm with reorthogonalization (CGS2 algorithm) [4]. This implementation of the CGS2 algorithm is parallelized by the

1 Research Group of Information and Communication Technology for Life, Nara Women's University, Nara 630–8506, Japan
2 Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
a) takata@ics.nara-wu.ac.jp
b) hishigami@amp.i.kyoto-u.ac.jp
c) kkimur@amp.i.kyoto-u.ac.jp
d) ynaka@i.kyoto-u.ac.jp

OpenMP [11], and hereafter is referred to as *OMP-CGS2* algorithm. The OMP-CGS2 algorithm enables to use effectively the cache of processors. Thus, the OMP-CGS2 algorithm is expected to achieve higher performance than the conventional reorthogonalization algorithms, which are parallelized in terms of the BLAS routines.

The rest of this paper is organized as follows. In Section 2, the GKLR algorithm and its implementation in this paper are described. In Section 3, a BLAS-based parallel implementation of reorthogonalization algorithms and the OMP-CGS2 algorithm are presented. Section 4 provides performance evaluations of the OMP-CGS2 algorithm on multi-core processors. In Section 5, we discuss the cache utilization in the OMP-CGS2 algorithm and a condition under which the OMP-CGS2 algorithm achieves higher performance than the CGS2 algorithm. We end with conclusions and future works in Section 6.

2. GKLR Algorithm and Its Implementation

In this section, we consider the GKLR algorithm and describe its implementation in this paper.

2.1 GKLR Algorithm

The GKL [5], [6] algorithm generates new bases $\boldsymbol{p}_k \in \mathbb{R}^n$ and $\boldsymbol{q}_k \in \mathbb{R}^m$ at the k-th iteration. The $\boldsymbol{p}_k$ is an orthonormal basis of the Krylov subspace $\mathcal{K}(A^\top A, \boldsymbol{p}_1, k)$ and the $\boldsymbol{q}_k$ is an orthonormal basis of the alternative Krylov subspace $\mathcal{K}(AA^\top, A\boldsymbol{p}_1, k)$. Note that the Krylov subspace $\mathcal{K}(S, \boldsymbol{p}_1, k)$ is defined by

$$\mathcal{K}(S, \boldsymbol{p}_1, k) = \mathrm{span}\left\{\boldsymbol{p}_1, S\boldsymbol{p}_1, S^2\boldsymbol{p}_1, \cdots, S^{k-1}\boldsymbol{p}_1\right\}, \tag{1}$$

where S is a real $n \times n$ symmetric matrix. In the GKLR algorithm [1], each time a new basis is added with the expansion of the Krylov subspace, the existing orthonormal basis, and the new basis are reorthogonalized.

Algorithm 1 shows the pseudocode of the GKLR algorithm, whose lines 6 and 10 are for the reorthogonalization. At the beginning of the k-th iteration for $k = 1, 2, \ldots$ in Algorithm 1, the $k \times k$ approximate matrices

$$B_k = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ & \alpha_2 & \beta_2 & & \\ & & \ddots & \ddots & \\ & & & \alpha_{k-1} & \beta_{k-1} \\ & & & & \alpha_k \end{bmatrix} \tag{2}$$

are obtained and the following equations hold

$$AP_k = Q_k B_k, \tag{3}$$

$$A^\top Q_k = P_k B_k^\top + \beta_k \boldsymbol{p}_{k+1} \boldsymbol{e}_k^\top, \tag{4}$$

where $\boldsymbol{e}_k$ is the k-th column vector of the $k \times k$ identity matrix. Note that if the ℓ largest singular values of B_k sufficiently approximate those of A, we can stop the iterations of the GKLR algorithm. In line 8 of Algorithm 1, we check whether the ℓ largest singular values of B_k sufficiently approximate those of A or not. Criteria for this check are discussed in Section 2.2.1.

Let $\sigma_j^{(k)}$, $\boldsymbol{s}_j^{(k)} \in \mathbb{R}^k$, and $\boldsymbol{t}_j^{(k)} \in \mathbb{R}^k$ ($j = 1, \ldots, k$) be a singular

Algorithm 1 GKLR algorithm

```
1: Set an unit vector $\boldsymbol{p}_1 \in \mathbb{R}^n$
2: $\boldsymbol{q} := A\boldsymbol{p}_1$, $\alpha_1 := \|\boldsymbol{q}\|_2$, $\boldsymbol{q}_1 := \boldsymbol{q}/\alpha_1$
3: $P_1 := [\boldsymbol{p}_1]$, $Q_1 := [\boldsymbol{q}_1]$
4: do $k := 1, 2, \ldots$
5:     $\boldsymbol{p} := A^\top \boldsymbol{q}_k$
6:     $\tilde{\boldsymbol{p}} :=$ Reorthogonalization($P_k$, $\boldsymbol{p}$)
7:     $\beta_k := \pm\|\tilde{\boldsymbol{p}}\|_2$, $\boldsymbol{p}_{k+1} := \tilde{\boldsymbol{p}}/\beta_k$
8:     Check the singular values of $B_k$
9:     $\boldsymbol{q} := A\boldsymbol{p}_{k+1}$
10:    $\tilde{\boldsymbol{q}} :=$ Reorthogonalization($Q_k$, $\boldsymbol{q}$)
11:    $\alpha_{k+1} := \pm\|\tilde{\boldsymbol{q}}\|_2$, $\boldsymbol{q}_{k+1} := \tilde{\boldsymbol{q}}/\alpha_{k+1}$
12:    $P_{k+1} := \begin{bmatrix} P_k & \boldsymbol{p}_{k+1} \end{bmatrix}$, $Q_{k+1} := \begin{bmatrix} Q_k & \boldsymbol{q}_{k+1} \end{bmatrix}$
13: end do
```

value of B_k, the left singular vector, and the right singular vector corresponding to $\sigma_j^{(k)}$, respectively. If $\sigma_j^{(k)}$ sufficiently approximates σ_j, then $\boldsymbol{u}_j$ and $\boldsymbol{v}_j$ corresponds to $\boldsymbol{u}_j^{(k)}$ and $\boldsymbol{v}_j^{(k)}$ defined by the following equations, respectively:

$$\boldsymbol{u}_j^{(k)} = Q_k \boldsymbol{s}_j^{(k)}, \ \boldsymbol{v}_j^{(k)} = P_k \boldsymbol{t}_j^{(k)}. \tag{5}$$

In order to improve the accuracy of singular vectors, this computation is implemented by combining with the QR factorization [10].

As seen in Algorithm 1, the GKLR algorithm must be parallelized in terms of the computations of each line. Since the computation of each line can be implemented using the BLAS routines, we parallelize the GKLR algorithm in terms of the BLAS routines.

2.2 Implementation of GKLR Algorithm

This section shows the implementation of the GKLR algorithm. In particular, we discuss the methods to check whether the singular values of B_k sufficiently approximate those of A or not and a stopping strategy of the GKLR algorithm. Then we present the implementation for computing a subset triplets of B_k in this paper.

2.2.1 Stopping Strategy of GKLR Algorithm

At first, stopping criteria of the GKLR algorithm are designed on the basis of the similar discussion to Section 13.2 in Ref. [12] as follows:

Recalling $(\sigma_j^{(k)},\ \boldsymbol{s}_j^{(k)}\ \boldsymbol{t}_j^{(k)})$, the j-th singular triplets of B_k ($j = 1, \ldots, \ell$), we then have the following equations:

$$B_k \boldsymbol{t}_j^{(k)} = \sigma_j^{(k)} \boldsymbol{s}_j^{(k)}, \ B_k^\top \boldsymbol{s}_j^{(k)} = \sigma_j^{(k)} \boldsymbol{t}_j^{(k)}. \tag{6}$$

Using Eqs. (3), (4), (5), and (6), we obtain

$$\begin{aligned} A^\top \boldsymbol{u}_j^{(k)} - \sigma_j^{(k)} \boldsymbol{v}_j^{(k)} &= A^\top Q_k \boldsymbol{s}_j^{(k)} - \sigma_j^{(k)} P_k \boldsymbol{t}_j^{(k)} \\ &= \left(A^\top Q_k - P_k B_k^\top\right) \boldsymbol{s}_j^{(k)} \\ &= \beta_k \boldsymbol{p}_{k+1} \boldsymbol{e}_k^\top \boldsymbol{s}_j^{(k)} \\ &= \beta_k s_j^{(k)}(k) \boldsymbol{p}_{k+1}, \end{aligned} \tag{7}$$

where $s_j^{(k)}(k)$ is the k-th element of $\boldsymbol{s}_j^{(k)}$. Thus, the following inequality holds:

$$\left\| A^\top \boldsymbol{u}_j^{(k)} - \sigma_j^{(k)} \boldsymbol{v}_j^{(k)} \right\|_2 = \left| \beta_k s_j^{(k)}(k) \right|. \tag{8}$$

Algorithm 2 Stopping strategy of GKLR algorithm

```
1: Compute $(\sigma_\ell^{(k)}, \boldsymbol{s}_\ell^{(k)}, \boldsymbol{t}_\ell^{(k)})$
2: if $|\beta_k s_\ell^{(k)}(k)| \le \delta$, then
3:     Compute $(\sigma_j^{(k)}, \boldsymbol{s}_j^{(k)}, \boldsymbol{t}_j^{(k)})$ for $j = 1, \ldots, \ell$
4:     if $|\beta_k s_j^{(k)}(k)| \le \delta$ for $j = 1, \ldots, \ell$, then
5:         Stop the iteration of GKLR algorithm
6:     end if
7: end if
```

As the results, if the right-hand side of inequality (8) is sufficiently small, then the singular value $\sigma_j^{(k)}$ of B_k can be regarded to sufficiently approximate that of A. Hence, the following inequality is considered as one of the stopping criteria of the GKLR algorithm:

$$\left|\beta_k s_j^{(k)}(k)\right| \le \delta, \quad j = 1, \ldots, \ell, \tag{9}$$

where δ is a threshold value for stopping the iteration of the GKLR algorithm and determined arbitrarily by users. If we use this criterion based on inequality (9), we have to compute the ℓ singular triplets of B_k, i.e. $(\sigma_j^{(k)}, \boldsymbol{s}_j^{(k)}, \boldsymbol{t}_j^{(k)})$, $j = 1, \ldots, \ell$, before checking whether inequality (9) is satisfied or not. The computational cost of computing singular triplets of B_k is higher than that of this check. In order to reduce the overall execution time for the GKLR algorithm, the computational cost of computing singular triplets of B_k has to be reduced. Hereafter, let k_t be the number of iterations where inequality (9) is satisfied for the first time.

Now let us consider the following inequality, which is one of the necessary conditions for inequality (9):

$$\left|\beta_k s_\ell^{(k)}(k)\right| \le \delta. \tag{10}$$

We have to compute only the ℓ-th largest singular triplet of B_k in order to check whether inequality (10) is satisfied. Hence, from the viewpoint of the computational cost, inequality (10) is more suitable for the stopping criterion of the GKLR algorithm than inequality (9). In addition, for the iteration ordinal k_n at which inequality (10) is satisfied for the first time, it is observed that $k_\mathrm{t} = k_\mathrm{n}$ in many cases of numerical experiments. From these facts, inequality (10) can be also considered as one of the stopping criteria of the GKLR algorithm. However, since the theorems in Ref. [13] imply that the value of k_t depends on the distribution of singular values for the target matrix, $k_\mathrm{t} = k_\mathrm{n}$ is not always guaranteed. Thus, even if inequality (10) is satisfied, we must check whether inequality (9) is also satisfied for all j.

Summarizing the discussions above, the stopping strategy for the GKLR algorithm is shown by Algorithm 2. In the experiments presented in Section 4, we set $\delta = 1.0 \times 10^{-14}$ as the stopping criterion. Note that Algorithm 2 is used in Algorithm 1 for its line 8. After stopping the iteration of the GKLR algorithm, we compute the ℓ largest singular triplets of A, i.e. $(\sigma_j, \boldsymbol{u}_j, \boldsymbol{v}_j)$ for $j = 1, \ldots, \ell$, using Eq. (5).

2.2.2 Subset Computation for Singular Triplets of Approximate Matrices

As mentioned in Section 2.2.1, a subset of singular triplets of the approximate matrices is required for stopping the GKLR algorithm. In this subsection, we discuss the subset computations of singular triplets of the approximate matrices in lines 1 and 3 of Algorithm 2.

The approximate matrix B_k, generated by the GKLR algorithm, is a lower bidiagonal matrix. As mentioned in Ref. [5], the singular value problem of the bidiagonal matrix can be transformed into the eigenvalue problem of the symmetric tridiagonal matrix without any computational cost. From this fact, the singular triplets of the lower bidiagonal matrix can be obtained using the bisection algorithm and the inverse iteration algorithm (BI algorithm) for symmetric tridiagonal matrices [12]. The BI algorithm enables us to compute only the desired eigenpairs and is suitable for the subset computation of singular triplets in Algorithm 2. While computing ℓ singular triplets (line 3 in Algorithm 2), we parallelize the subset computation of singular triplets as follows: The bisection algorithm is parallelized in terms of each singular value, and the inverse iteration algorithm is parallelized in terms of the BLAS routines.

3. Reorthogonalization Algorithms

To improve the orthogonality of the Krylov subspace and the accuracy of the resulting singular vectors, the reorthogonalization is inevitable for the GKLR. However, the computational cost of the reorthogonalization is higher than that of the other processes in the GKLR, as the iteration number increases. Thus, it is important to accelerate the reorthogonalization in the GKLR.

In this section, at first, we consider three conventional reorthogonalization algorithms for the GKLR algorithm. The classical Gram-Schmidt with reorthogonalization (CGS2) algorithm [4], the modified Gram-Schmidt (MGS) algorithm [6], and the reorthogonalization algorithm using the Householder transformations in terms of the compact WY representation (cWY algorithm) [18]. These algorithms are parallelized in terms of the BLAS routines in recent days. Secondly, we present the OpenMP-based parallel implementation of the CGS2 algorithm for shared-memory multi-core processors and describe the advantage of this implementation with respect to the data usability.

In the followings, we discuss the computation of $\tilde{\boldsymbol{x}}_i \in \mathbb{R}^m$, which is the orthogonalized vector of $\boldsymbol{a}_i \in \mathbb{R}^m$ $(2 \le i \le n)$ and satisfies $\langle \tilde{\boldsymbol{x}}_i, \boldsymbol{x}_j \rangle = 0$ for $j \ne i$, where $\boldsymbol{x}_j = \tilde{\boldsymbol{x}}_j / \|\tilde{\boldsymbol{x}}_j\|$. In addition, let X_{i-1} be $X_{i-1} = \left[\boldsymbol{x}_1 \cdots \boldsymbol{x}_{i-1}\right]$ $(2 \le i \le n)$. Note that X_{i-1}, $\tilde{\boldsymbol{x}}_i$, and $\boldsymbol{a}_i$ correspond to P_k, $\tilde{\boldsymbol{p}}$, and $\boldsymbol{p}$ of line 6 in Algorithm 1, and also correspond to Q_k, $\tilde{\boldsymbol{q}}$, and $\boldsymbol{q}$ of line 10 in Algorithm 1.

3.1 BLAS-based Parallel Implementation Algorithms

3.1.1 CGS2 Algorithm

The classical Gram-Schmidt (CGS) algorithm [6] is a well-known reorthogonalization algorithm. The reorthogonalization of $\boldsymbol{a}_i$ using the CGS algorithm is formulated as follows:

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{a}_i - \sum_{j=1}^{i-1} \langle \boldsymbol{x}_j, \boldsymbol{a}_i \rangle \boldsymbol{x}_j. \tag{11}$$

Equation (11) is composed of the level 1 BLAS routines, such as inner-dot products and AXPY operations. The computational cost of the CGS algorithm is about $2mk^2$ if the reorthogonalization of $\boldsymbol{a}_i$ for $i = 1, \ldots, k$ is performed. Using the matrix-vector multiplications, Eq. (11) is also replaced as

Algorithm 3 CGS2 algorithm

```
1: function CGS2$X_{i-1}(:= [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i-1}])$, $\boldsymbol{a}_i$
2:     do $I := 1, 2$
3:         $\boldsymbol{w} := X_{i-1}^\top \boldsymbol{a}_i$
4:         $\boldsymbol{a}_i := \boldsymbol{a}_i - X_{i-1}\boldsymbol{w}$
5:     end do
6:     return $\tilde{\boldsymbol{x}}_i := \boldsymbol{a}_i$
7: end function
```

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{a}_i - X_{i-1}X_{i-1}^\top \boldsymbol{a}_i. \tag{12}$$

In general, to achieve better performance, we reduce the number of data synchronizations on shared-memory multi-core processors as much as possible. The level 2 BLAS routines, such as the matrix-vector multiplications, have less data synchronization than the level 1 BLAS routines. Thus, the level 2 BLAS routines achieves better performance than the level 1 BLAS routines in parallel processing. Given this property, the CGS is conventionally implemented using matrix-vector multiplications.

However, the orthogonality of the vectors computed by the CGS algorithm deteriorates if the condition number of the original vectors is large. To improve the orthogonality, the variants of the CGS algorithm have been proposed.

One of the variants is the CGS algorithm with reorthogonalization (CGS2 algorithm) [4], which repeats the CGS algorithm twice. A pseudocode of the CGS2 is shown in Algorithm 3. Although the orthogonality of computed vectors by the CGS2 algorithm is theoretically better than that by the CGS algorithm [4], the computational cost of the CGS2 is twice higher than that of the CGS.

3.1.2 MGS Algorithm

Another variant of the CGS algorithm is the modified Gram-Schmidt (MGS) algorithm [6]. The MGS algorithm is composed of the level 1 BLAS routines such as inner-dot products and AXPY operations. Thus, this reorthogonalization is difficult to achieve a high performance in parallel processing. However, the computational cost of the MGS algorithm is the same as that of the CGS algorithm since they are algebraically equivalent.

3.1.3 Householder-based Reorthogonalization Algorithm

The Householder transformations [6] are also used for the reorthogonalization. The straightforward implementation of this reorthogonalization is composed of the level 1 BLAS routines. In Ref. [18], the implementation of this reorthogonalization based on the level 2 BLAS routines is proposed by introducing the compact WY representation [15] into the product of the Householder transformations, and thus is shown to achieve high scalability in parallel processing. Hereafter, the algorithm in Ref. [18] is referred to as the cWY algorithm. In addition, the computational cost of the cWY algorithm can be reduced from $4mk^2 + k^3$ to $4mk^2 - k^3$ [8].

3.2 OMP-CGS2 Algorithm

Recalling Eq. (11), the CGS and CGS2 algorithms can be parallelized in terms of the summation. Such parallel implementation is easily realized by adding OpenMP directives for shared-memory multi-core processors. From these facts, an OpenMP-based parallel implementation of the CGS2 algorithm can be represented as shown in Algorithm 4. Hereafter, this implementation of the CGS2 algorithm is referred to as the OMP-CGS2 algorithm. It should be noted that the OMP-CGS2 algorithm is based on the same idea as the parallel implementation of the CGS algorithm proposed in Ref. [9], which is parallelized for distributed memory systems by using the MPI (Message Passing Interface) and is adopted to the reorthogonalization process of the inverse iteration algorithm.

Algorithm 4 OMP-CGS2 algorithm

```
 1: function OMP-CGS2$X_{i-1}(:= [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i-1}])$, $\boldsymbol{a}_i$
 2:     #omp parallel private($I$, $s$)
 3:     do $I := 1, 2$
 4:         #omp single
 5:         $\boldsymbol{w} := \boldsymbol{a}_i$                        ▹ Serially performed
 6:         #omp end single
 7:         #omp do reduction(+:$\boldsymbol{a}_i$)
 8:         do $j := 1$ to $i - 1$
 9:             $s := -\langle \boldsymbol{x}_j, \boldsymbol{w} \rangle$
10:             $\boldsymbol{a}_i := \boldsymbol{a}_i + s\boldsymbol{x}_j$                 ▹ Array reduction
11:         end do
12:         #omp end do
13:     end do
14:     #omp end parallel
15:     return $\tilde{\boldsymbol{x}}_i := \boldsymbol{a}_i$
16: end function
```

Table 1 Comparison of reorthogonalization algorithms [18].

	CGS2	MGS	cWY	OMP-CGS2
Computation	$4mk^2$	$2mk^2$	$4mk^2 - k^3$	$4mk^2$
Orthogonality	$O(\epsilon)^\dagger$	$O(\epsilon\kappa(A))$	$O(\epsilon)$	$O(\epsilon)^\dagger$
BLAS	Level 2	Level 1	Level 2	Level 1

†: Realized if the condition $O(\epsilon\kappa(A)) < 1$ is satisfied.

As shown in line 7, the parallelization of the summation is represented as that of **do**-loop by the OpenMP directive. In this case, the inner-dot product (line 9) and the AXPY operations (line 10) in terms of the different index j is performed serially on each computing thread. In addition, the array reduction must be implemented for the summation of $\boldsymbol{a}_i$ on line 10. Note that the array reduction in Fortran code is supported by using the `reduction` clause of OpenMP.

The advantage of this implementation is the high reusability of data. Since we compute $\boldsymbol{a}_i := \boldsymbol{a}_i + s\boldsymbol{x}_j$ (line 10) as soon as $s := -\langle \boldsymbol{x}_j, \boldsymbol{w} \rangle$ (line 9) is computed, the reusability of $\boldsymbol{w}$, $\boldsymbol{x}_j$, and $\boldsymbol{a}_i$ becomes higher on each thread computation. Thus, the OMP-CGS2 algorithm is expected to accelerate more effectively the reorthogonalization computation on shared-memory multi-core processors with large caches than other reorthogonalization algorithms if the vectors $\boldsymbol{w}$, $\boldsymbol{x}_j$, and $\boldsymbol{a}_i$ are stored in the L3 cache of each processor.

3.3 Comparison of Reorthogonalization Algorithms

As the summary of this section, the theoretical performance of the reorthogonalization algorithms is shown in **Table 1**, where *Computation* denotes the flops of the computational cost, *Orthogonality* indicates the bound of the norm $\|X^\top X - I\|$, and *BLAS* denotes the level of the BLAS routines of which each algorithm is mainly composed. In addition, ϵ is the machine epsilon and $\kappa(A)$

Table 2 Specifications of the experimental environment.

	1 node of Appro 2548X at ACCMS, Kyoto University
CPU	Intel Xeon E5-4650L@2.6 GHz, 32 cores (8 cores × 4) L3 cache: 20 MB × 4
RAM	DDR3-1066 1.5 TB, 136.4 GB/sec
Compiler	Intel C++/Fortran Compiler 14.0.2

Options	`-O3 -xHOST -ipo -no-prec-div`
	`-openmp -mcmodel=medium -shared-intel`

Run Command	`numactl --localalloc`
Software	Intel Math Kernel Library 11.1.2

denotes the condition number of $A = \left[\boldsymbol{a}_1 \cdots \boldsymbol{a}_k\right]$.

4. Performance Evaluation

In this section, we report experimental results in order to evaluate the performance of the OpenMP-based parallel implementation of the CGS2 algorithm.

4.1 Configurations of Numerical Experiments

In the numerical experiments, we compare the execution time for computing the ℓ largest singular triplets of the same target matrix using a code of the GKLR algorithm with different ℓ. Here, ℓ is the number of desired singular triplets; $\ell = 100, 200, 400, 800$.

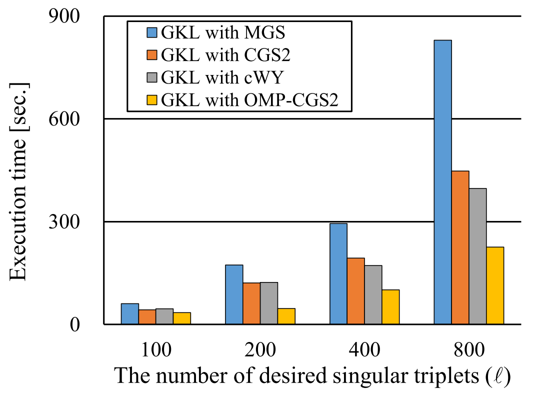
We compare the numerical results for computing subsets of singular triplets using four different codes of the GKLR algorithm. Each GKLR code is implemented with the following four reorthogonalization algorithms shown in Section 3. **GKL with MGS** is implemented with the MGS algorithm. **GKL with CGS2** is implemented with the CGS2 algorithm shown in Algorithm 3. **GKL with cWY** is implemented with the cWY algorithm. The reorthogonalization algorithms of these three code are parallelized in terms of the BLAS routines. **GKL with OMP-CGS2** is implemented with the OMP-CGS2 algorithm shown in Algorithm 4.

In the experiments, we used four $m \times n$ real matrices A_1, A_2, A_3, and A_4. All of A_1, A_2, and A_3 are sparse matrices having in each row 256 non-zero elements, which are set to be uniform random numbers in the range $(0, 1)$ and are randomly allocated. A_1, A_2, and A_3 are only different in both m and n from each other as follows: $m = 16,000$ and $n = 8,000$ for A_1. $m = 32,000$ and $n = 16,000$ for A_2. $m = 64,000$ and $n = 32,000$ for A_3. Note that the condition number is 4.803×10^1 for A_1, 4.754×10^1 for A_2, and 4.757×10^1 for A_3, respectively. In addition, the Frank matrix with $m = 32,000$, $n = 32,000$, and the condition number 1.600×10^9 was used as A_4.

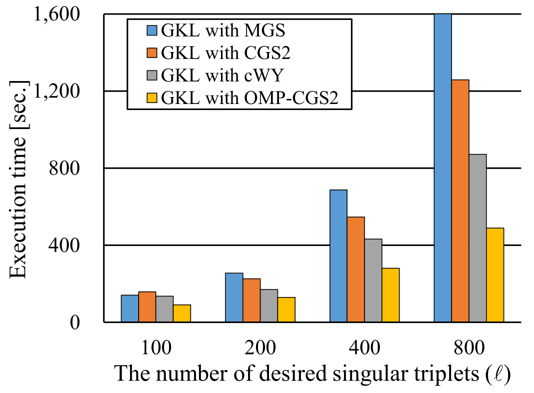
All the experiments were performed with 32 threads on one node of Appro 2548X at ACCMS, Kyoto University, whose specification is listed in **Table 2**. We used the Intel Math Kernel Library (MKL) [7] for parallelizing the BLAS routines. To control the memory allocation, all the experiments were run with `numactl --localalloc` command.

4.2 Experimental Results

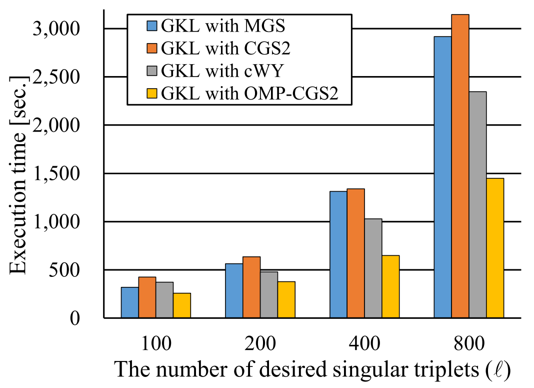
Figure 1 illustrates the experimental results showing the number of desired singular triplets and the execution time for computing singular triplets of each target matrix using the four code of the GKLR algorithm. Figure 1 (a), 1 (b), 1 (c), and 1 (d) cor-



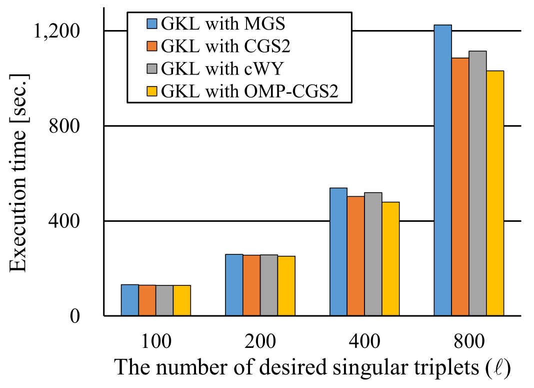
(a) Cases of A_1



(b) Cases of A_2



(c) Cases of A_3



(d) Cases of A_4

Fig. 1 The number of desired singular triplets and the execution time for computing the ℓ largest singular triplets of the target matrix using four GKLR codes where the GKL algorithms with different reorthogonalization process are implemented.

Table 4 The orthogonality of the singular vectors of the target matrix computed by four GKLR codes where the GKL algorithms with different reorthogonalization process are implemented. "Left" denotes the error of orthogonality of the left singular vectors: $\|U_\ell^\top U_\ell - I\|_F / \sqrt{\ell}$. "Right" denotes the error of orthogonality of the right singular vectors: $\|V^\top V - I\|_F / \sqrt{\ell}$.

(a) Cases of A_1

ℓ: # of desired singular triplets	100		200		400		800	
	Left	Right	Left	Right	Left	Right	Left	Right
GKL with MGS	1.40E-15	1.34E-15	1.53E-15	1.56E-15	1.81E-15	1.99E-15	2.39E-15	2.35E-15
GKL with CGS2	1.81E-15	1.47E-15	2.08E-15	1.84E-15	2.45E-15	2.15E-15	3.05E-15	2.66E-15
GKL with cWY	3.54E-15	2.59E-15	3.92E-15	2.93E-15	4.60E-15	3.67E-15	5.41E-15	4.61E-15
GKL with OMP-CGS2	1.75E-15	1.56E-15	2.13E-15	1.85E-15	2.38E-15	2.15E-15	2.95E-15	2.59E-15

(b) Cases of A_2

ℓ: # of desired singular triplets	100		200		400		800	
	Left	Right	Left	Right	Left	Right	Left	Right
GKL with MGS	1.41E-15	1.60E-15	1.74E-15	1.64E-15	2.10E-15	2.03E-15	2.66E-15	2.61E-15
GKL with CGS2	2.13E-15	1.74E-15	2.31E-15	2.16E-15	2.78E-15	2.54E-15	3.43E-15	3.06E-15
GKL with cWY	4.24E-15	3.38E-15	4.74E-15	3.58E-15	5.40E-15	4.25E-15	6.33E-15	5.01E-15
GKL with OMP-CGS2	1.98E-15	1.81E-15	2.30E-15	2.20E-15	2.84E-15	2.54E-15	3.31E-15	2.92E-15

(c) Cases of A_3

ℓ: # of desired singular triplets	100		200		400		800	
	Left	Right	Left	Right	Left	Right	Left	Right
GKL with MGS	1.89E-15	1.60E-15	2.24E-15	1.82E-15	2.50E-15	2.37E-15	3.05E-15	2.97E-15
GKL with CGS2	3.09E-15	1.98E-15	2.97E-15	2.37E-15	3.46E-15	2.84E-15	3.95E-15	3.47E-15
GKL with cWY	5.33E-15	3.95E-15	5.90E-15	4.31E-15	6.61E-15	4.94E-15	7.57E-15	5.92E-15
GKL with OMP-CGS2	2.89E-15	2.10E-15	3.08E-15	2.59E-15	3.60E-15	2.86E-15	3.99E-15	3.47E-15

(d) Cases of A_4

ℓ: # of desired singular triplets	100		200		400		800	
	Left	Right	Left	Right	Left	Right	Left	Right
GKL with MGS	8.09E-15	3.95E-14	9.84E-15	3.97E-14	2.19E-14	4.16E-14	8.19E-14	4.40E-14
GKL with CGS2	4.24E-15	4.39E-15	4.68E-15	4.82E-15	4.87E-15	5.00E-15	4.94E-15	4.99E-15
GKL with cWY	1.08E-14	1.08E-14	1.12E-14	1.07E-14	1.10E-14	1.07E-14	1.11E-14	1.11E-14
GKL with OMP-CGS2	4.42E-15	4.69E-15	4.81E-15	4.36E-15	4.88E-15	4.63E-15	4.85E-15	4.66E-15

Table 3 The iteration number of the GKLR codes in each experiment.

ℓ: # of desired singular triplets	100	200	400	800
Matrix A_1	1,000	1,600	2,400	4,000
Matrix A_2	1,300	2,000	3,200	4,800
Matrix A_3	1,600	2,400	3,600	5,600
Matrix A_4	200	400	800	1,600

responds to the cases of target matrices A_1, A_2, A_3, and A_4, respectively. From these figures, **GKL with OMP-CGS2** is faster than the other codes in all the cases. Thus, the OMP-CGS2 accelerates the computation of the GKLR algorithm more effectively than other reorthogonalization algorithms. The iteration number of the GKLR codes in each experiment is shown in **Table 3**. Note that the iteration number is not changed in Table 3 if we replace the reorthogonalization algorithm, for example, from the MGS to the CGS2 algorithm.

Table 4 show the orthogonality of the singular vectors computed by each GKLR code. Table 4 (a), 4 (b), 4 (c), and 4 (d) shows the cases of test matrices A_1, A_2, A_3, and A_4, respectively. Note that "Left" denotes $\|U_\ell^\top U_\ell - I\|_F / \sqrt{\ell}$, where $U_\ell = \left[\boldsymbol{u}_1 \cdots \boldsymbol{u}_\ell\right]$ and each of $\boldsymbol{u}_j$ is the left singular vector computed by each code. Similarly, "Right" denotes $\|V_\ell^\top V_\ell - I\|_F / \sqrt{\ell}$, where $V_\ell = \left[\boldsymbol{v}_1 \cdots \boldsymbol{v}_\ell\right]$ and each of $\boldsymbol{v}_j$ is the left singular vector computed by each code. From Table 4, we can observe that **GKL with OMP-CGS2** achieves as high orthogonality as other GKLR codes.

In addition, **Table 5** (a), 5 (b), 5 (c), and 5 (d) shows the number of desired singular triplets and the execution time spending for the reorthogonalization process in computing the singular triplets

Table 5 The number of desired singular triplets (ℓ) and the execution time (sec.) spending for the reorthogonalization process in computing the singular triplets of the target matrix using four GKLR codes.

(a) Cases of A_1

ℓ: # of desired singular triplets	100	200	400	800
GKL with MGS	46	158	272	780
GKL with CGS2	24	105	166	385
GKL with cWY	25	107	145	339
GKL with OMP-CGS2	11	32	74	168

(b) Cases of A_2

ℓ: # of desired singular triplets	100	200	400	800
GKL with MGS	106	219	622	1,490
GKL with CGS2	98	187	485	1,148
GKL with cWY	81	143	361	764
GKL with OMP-CGS2	37	89	215	379

(c) Cases of A_3

ℓ: # of desired singular triplets	100	200	400	800
GKL with MGS	205	469	1,177	2,668
GKL with CGS2	285	544	1,185	2,888
GKL with cWY	220	380	881	2,098
GKL with OMP-CGS2	106	256	514	1,143

(d) Cases of A_4

ℓ: # of desired singular triplets	100	200	400	800
GKL with MGS	4.9	19	64	245
GKL with CGS2	1.8	7.7	29	108
GKL with cWY	2.4	9.2	36	124
GKL with OMP-CGS2	2.3	5.4	17	50

of A_1, A_2, A_3, and A_4 using each GKLR code, respectively. The tables show that, in the case of A_4 with $\ell = 100$, the execution time for the OMP-CGS2, the reorthogonalization in **GKL with OMP-CGS2**, is almost the same as that for the CGS2 algorithm, the reorthogonalization in **GKL with CGS2**, but the OMP-CGS2 is faster than the CGS2 algorithm in the other cases.

5. Cache utilization in OMP-CGS2

As mentioned in Section 3.2, the high performance of the OMP-CGS2 algorithm arises from the high reusability of processors' cache. However, the CGS2 algorithm may achieve higher performance than that of the OMP-CGS2 algorithm if the capacity of cache is not sufficient. Then, it is desirable that we know which the CGS2 algorithm or the OMP-CGS2 algorithm achieves a higher performance before adopting the GKLR code to actual applications. In the followings, we discuss the cache utilization in the OMP-CGS2 algorithm and a condition under which the OMP-CGS2 algorithm achieves higher performance than the CGS2 algorithm.

5.1 Discussion

Recalling Algorithm 4, the vectors $\boldsymbol{w}$, $\boldsymbol{a}_i$, and $\boldsymbol{x}_j$ appear at each of **do**-loop in terms of j in lines 7–11. From this fact, whether the OMP-CGS2 algorithm achieves higher performance than the CGS2 algorithm depends on m, which is the size of all these vectors, and let m_{cache} be the maximum size of the vectors at the time when the OMP-CGS2 algorithm achieves higher performance than the CGS2 algorithm does.

If all these vectors are stored in the L3 cache of the processors within a computer, such superiority of the OMP-CGS2 over the CGS2 is guaranteed. However, $\boldsymbol{x}_j$ is not shared by different threads while $\boldsymbol{w}$ is accessed by all computing threads. In addition, each thread should access the copy of $\boldsymbol{a}_i$ before reducing arrays. As a result, the number of the vectors which should be stored in the cache is $(T \times 2 + 1)$ where T is the number of threads in each processor.

From the discussion above, assuming that neither any other software works nor any other data is stored in the cache, the following inequality must be satisfied:

$$m \times (T \times 2 + 1) \times 8 \leq C \times 1024 \times 1024, \tag{13}$$

where C is the capacity of processor's L3 cache in MB and a one data of the elements needs 8 bytes when we use double-precision floating-point numbers. If m satisfies inequality (13), the OMP-CGS2 algorithm is theoretically guaranteed to achieve higher performance than the CGS2 algorithm. Hereafter, let $m_{\text{cache}}^{(\text{theory})}$ be the maximum value of m at which inequality (13) is satisfied. Even if $m > m_{\text{cache}}^{(\text{theory})}$, however, $\boldsymbol{x}_j$ on each thread possibly be stored in cache while the computation of line 10 in Algorithm 4 are executed. Thus, the CGS2 algorithm is not always guaranteed to achieve higher performance than the OMP-CGS2 algorithm in such a case. From these discussions, numerical experiments in the case when $m > m_{\text{cache}}^{(\text{theory})}$ helps us to know a true value of m_{cache}.

5.2 Verification

According to inequality (13), $m_{\text{cache}}^{(\text{theory})}$ for a machine in Table 2 is computed as follows: Since $T = 8$ and $C = 20$ from Table 2,

$$m_{\text{cache}}^{(\text{theory})} = 154202. \tag{14}$$

In fact, **GKL with OMP-CGS2** is faster than **GKL with CGS2** in all the experiments shown in Section 4 because both $m \leq m_{\text{cache}}^{(\text{theory})}$ and $n \leq m_{\text{cache}}^{(\text{theory})}$ are satisfied.

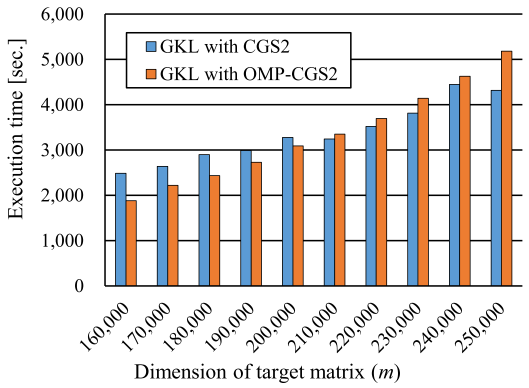


Fig. 2 Comparison of the execution times for computing the desired singular triplets of the $m \times m$ target matrix using **GKL with CGS2** and **GKL with OMP-CGS2**.

To examine a true value of m_{cache} for a machine shown in Table 2, the numerical results is shown as follows: **Fig. 2** shows the execution times for computing 100 singular triplets of $m \times m$ target matrices by **GKL with CGS2** and **GKL with OMP-CGS2**. Note that the iteration number of both GKLR codes is $2,500$ for $m = 160,000$, and $m = 170,000$, $2,600$ for $m = 180,000$ and $m = 190,000$, $2,700$ for $m = 200,000$, $m = 210,000$ and $m = 220,000$, $2,800$ for $m = 230,000$, and $2,900$ for $m = 240,000$ and $m = 250,000$, respectively. As can be seen in Fig. 2, m_{cache} for the experimental environment in this paper is ranged from $200,000$ to $210,000$.

6. Conclusions and Future Work

In this paper, we first introduce the GKLR algorithm for computing a subset of singular triplets of target matrices. To more effectively accelerate the reorthogonalization of the GKLR algorithm on shared-memory multi-core processors, we then present the OMP-CGS2 algorithm, which is parallelized by the OpenMP directives and moreover has the advantage of the data reusability. We performed numerical experiments on shared-memory multi-core processors to evaluate the performance of the GKL algorithm with the OMP-CGS2 algorithm. The experimental results show that the OMP-CGS2 algorithm accelerates more effectively the subset computations of singular triplets of some target matrices by the GKLR algorithm than other reorthogonalization algorithms. In addition, the cache utilization in the OMP-CGS2 algorithm is discussed and the condition that the OMP-CGS2 algorithm achieves higher performance than the CGS2 algorithm is examined through both theoretical approach and numerical experiments.

Future work is to apply the OMP-CGS2 algorithm to other algorithms, such as the inverse iteration method, GMRES algorithm [14], and implicitly restarted Arnoldi and Lanczos methods [3], [17] to accelerate their reorthogonalization processes.

Acknowledgments The authors would like to express their gratitude to reviewers of this paper for their helpful comments. This work was supported in part by JSPS KAKENHI Grant Numbers 13J02820 and 24360038. In this work, we used the supercomputer of ACCMS, Kyoto University.

References

[1] Barlow, J.L.: Reorthogonalization for the Golub-Kahan-Lanczos bidiagonal reduction, *Numer. Math.*, pp.1–42 (2013).

[2] Blackford, L.S., Demmel, J.W., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K. and Whaley, R.C.: An updated set of basic linear algebra subprograms (BLAS), *ACM Trans. Math. Softw.*, Vol.28, No.2, pp.135–151 (2002).

[3] Calvetti, D., Reichel, L. and Sorensen, D.C.: An implicitly restarted Lanczos method for large symmetric eigenvalue problems, *ETNA*, Vol.2, pp.1–21 (1994).

[4] Daniel, J.W., Gragg, W.B., Kaufman, L. and Stewart, G.W.: Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization, *Math. Comput.*, Vol.30, No.136, pp.772–795 (1976).

[5] Golub, G. and Kahan, W.: Calculating the singular values and pseudo-inverse of a matrix, *SIAM J. Numer. Anal.*, Vol.2, No.2, pp.205–224 (1965).

[6] Golub, G.H. and van Loan, C.F.: *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, USA (1996).

[7] Intel Math Kernel Library: available from ⟨https://software.intel.com/en-us/intel-mkl/⟩ (2003).

[8] Ishigami, H., Kimura, K. and Nakamura, Y.: On implementation and evaluation of inverse iteration algorithm with compact WY orthogonalization, *IPSJ Transactions on Mathematical Modeling and Its Applications*, Vol.6, No.2, pp.25–35 (2013).

[9] Katagiri, T.: Performance evaluation of parallel Gram-Schmidt re-orthogonalization methods, *High Performance Computing for Computational Science — VECPAR 2002*, Lecture Notes in Computer Science, Vol.2565, Springer Berlin Heidelberg, pp.302–314 (2003).

[10] Lehoucq, R.B., Sorensen, D.C. and Yang, C.: *ARPACK Users's Guide*, SIAM, Philadelphia, PA, USA (1998).

[11] OpenMP: available from ⟨http://openmp.org/wp/⟩ (1997).

[12] Parlett, B.N.: *The Symmetric Eigenvalue Problem*, SIAM, Philadelphia, PA, USA (1998).

[13] Saad, Y.: On the rates of convergence of the Lanczos and the block-Lanczos methods, *SIAM J Numer. Anal.*, Vol.17, No.5, pp.687–706 (1980).

[14] Saad, Y. and Schultz, M.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Stat. Comput.*, Vol.7, No.3, pp.856–869 (1986).

[15] Schreiber, R. and van Loan, C.: A storage-efficient WY representation for products of Householder transformations, *SIAM J. Sci. Stat. Comput.*, Vol.10, No.1, pp.53–57 (1989).

[16] Simon, H.D. and Zha, H.: Low-Rank Matrix Approximation Using the Lanczos Bidiagonalization Process with Applications, *SIAM J Sci. Comput.*, Vol.21, No.6, pp.2257–2274 (2000).

[17] Sorensen, D.C.: Implicit application of polynomial filters in a k-step Arnoldi method, *SIAM J. Matrix Anal. Appl.*, Vol.13, No.1, pp.357–385 (1992).

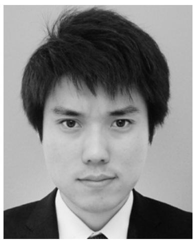
[18] Yamamoto, Y. and Hirota, Y.: A parallel algorithm for incremental orthogonalization based on the compact WY representation, *JSIAM Letters*, Vol.3, pp.89–92 (2011).

Masami Takata is a lecturer of the Research Group of Information and Communication Technology for Life at Nara Women's University. She received her Ph.D. degree from Nara Women's University in 2004. Her research interests include parallel algorithms for distributed memory systems and numerical algebra.

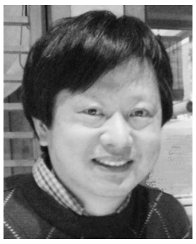
Hiroyuki Ishigami received his Ph.D. degree from Kyoto University in 2016. His research interests include parallel algorithms for eigenvalue and singular value decomposition. He is an IPSJ member.

Kinji Kimura received his Ph.D. degree from Kobe University in 2004. He became a PRESTO, COE, and CREST researcher in 2004 and 2005. He became an assistant professor at Kyoto University in 2006, an assistant professor at Niigata University in 2007, a lecturer at Kyoto University in 2008, and has been a program-specific associate professor at Kyoto University since 2009. He is an IPSJ member.

Yuki Fujii received his B.E. and M.I. degrees from Kyoto University in 2013 and 2015. His research interests include the parallel computation of the partial eigenvalue decomposition for sparse matrices.



Hiroki Tanaka received his B.E. and M.I. degrees from Kyoto University in 2013 and 2015. Since 2015, he is an application developer of a private enterprise.



Yoshimasa Nakamura has been a professor of Graduate School of Informatics, Kyoto University from 2001. His research interests includes integrable dynamical systems which originally appear in classical mechanics. But integrable systems have a rich mathematical structure. His recent subject is to design new numerical algorithms such as the mdLVs and I-SVD for singular value decomposition by using discrete-time integrable systems. He is a member of JSIAM, SIAM, MSJ and AMS.