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,d)] Hiroki Tanaka[2,e)]
Yoshimasa Nakamura[2,f)]

Abstract: The Golub-Kahan-Lanczos algorithm with reorthogonalization (GKLR algorithm) is an algorithm for computing a subset of singular triplets for large-scale sparse matrices. The reorthogonalization tends to become a bottleneck of elapsed 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 for a sparse matrix than the conventional reorthogonalization algorithms.

1. Introduction

Let A be a real $m \times n$ matrix and $\mathrm{rank}(A) = r$ ($r \leq \min(m,\ n)$). Then A has the r singular values $\sigma_1,\ \ldots,\ \sigma_r \in \mathbb{R}$, which satisfies $\sigma_1 \geq \cdots \geq \sigma_r > 0$, and their corresponding left and right singular vectors $\boldsymbol{u}_i \in \mathbb{R}^m$, $\boldsymbol{v}_i \in \mathbb{R}^n$ ($1 \leq i \leq r$). A subset of singular triplets, i.e. the l largest singular values $\sigma_1,\ \ldots,\ \sigma_l$ and their corresponding singular vectors, is often required in low-rank matrix approximation and statistical processings 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 perform the computation of singular triplets directly from a large-scale sparse matrix because of the computational cost and need for large amounts of memory.

The Golub-Kahan-Lanczos (GKL) algorithm [3], [4] 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, let us incorporate a reorthogonalization process into the GKL algorithm. Such an algorithm is referred to as the GKL algorithm with reorthogonalization (GKLR algorithm) [1]. 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 elapsed time as the iteration number increases. However, since the reorthogonalization of the GKLR algorithm is mainly implemented using the matrix-vector multiplications, even in parallel computing, the reorthogonalization is not effectively accelerated and then the overall elapsed time of the GKLR algorithm is not effectively reduced.

In this paper, to accelerate the reorthogonalization of the GKLR algorithm more effectively in parallel computing, we present a parallel implementation of the classical Gram-Schmidt algorithm with reorthogonalization (CGS2 algorithm) [2], which is parallelized using the OpenMP [7]. Hereafter, this implementation of the CGS2 is referred to as the OMP-CGS2 algorithm. This parallelization technique enables to use the cache of CPUs effectively and then the computation is expected to be accelerated more effectively than the conventional reorthogonalization algorithms, which are parallelized in terms of the BLAS operations.

The rest of this paper is organized as follows. In Section 2, we describe the GKLR algorithm. In Section 3, a conventional reorthogonalization algorithms and the OMP-CGS2 algorithm are presented. Section 4 provides performance evaluations of the OMP-CGS2 algorithm on multi-core processors. We end with conclusions and future works in Section 5.

2. GKLR algorithm

The Golub-Kahan-Lanczos [3] (GKL) algorithm generates new bases $\boldsymbol{p}_k \in \mathbb{R}^n$ and $\boldsymbol{q}_k \in \mathbb{R}^m$, iteratively ($k = 1,\ 2,\ \ldots$). 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 sub-

[1] Research Group of Information and Communication Technology for Life, Nara Women's University, Nara, JAPAN
[2] Graduate School of Informatics, Kyoto University, Kyoto, 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) yuki.fujii@amp.i.kyoto-u.ac.jp
e) hirotnk@amp.i.kyoto-u.ac.jp
f) ynaka@i.kyoto-u.ac.jp

Algorithm 1 GKLR algorithm

1: Set an n-dimensional unit vector $\boldsymbol{p}_1$
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: $\quad \boldsymbol{p} = A^\top \boldsymbol{q}_k$
6: $\quad \tilde{\boldsymbol{p}} =$ Reorthogonalization$(P_k,\ \boldsymbol{p})$
7: $\quad \beta_k = \pm\|\tilde{\boldsymbol{p}}\|_2,\ \boldsymbol{p}_{k+1} = \tilde{\boldsymbol{p}}/\beta_k$
8: $\quad \boldsymbol{q} = A\boldsymbol{p}_{k+1}$
9: $\quad \tilde{\boldsymbol{q}} =$ Reorthogonalization$(Q_k,\ \boldsymbol{q})$
10: $\quad \alpha_{k+1} = \pm\|\tilde{\boldsymbol{q}}\|_2,\ \boldsymbol{q}_{k+1} = \tilde{\boldsymbol{q}}/\alpha_{k+1}$
11: $\quad P_{k+1} = \begin{bmatrix} P_k & \boldsymbol{p}_{k+1} \end{bmatrix}, Q_{k+1} = \begin{bmatrix} Q_k & \boldsymbol{q}_{k+1} \end{bmatrix}$
12: **end do**

space $\mathcal{K}(AA^\top,\ A\boldsymbol{p}_1,\ k)$. In the GKLR algorithm [1], each time a new basis is added with the expansion of the Krylov subspace, the new basis are reorthogonalized against the existing bases.

Algorithm 1 shows the pseudocode of the GKLR algorithm. Lines 6 and 9 show the reorthogonalization process, respectively. At the beginning of the k-th iteration for $k = 1, 2, \ldots$ in Algorithm 1, the $k \times k$ approximate matrices

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

are obtained and the following equations hold

$$AP_k = Q_k B_k, \tag{2}$$

$$A^\top Q_k = P_k B_k^\top + \beta_k \boldsymbol{p}_{k+1} \boldsymbol{e}_k^\top, \tag{3}$$

where $\boldsymbol{e}_k$ is the k-th column 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. 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 value of B_k, the left singular vector, and the right singular vector corresponding to $\sigma_j^{(k)}$, respectively. If $\sigma_j^{(k)}$ approximates σ_j well, then $\boldsymbol{u}_j$ and $\boldsymbol{v}_j$ corresponds to $\boldsymbol{u}_j^{(k)}$ and $\boldsymbol{v}_j^{(k)}$ defined as the following equations, respectively:

$$\boldsymbol{u}_j^{(k)} = Q_k \boldsymbol{s}_j^{(k)},\ \boldsymbol{v}_j^{(k)} = P_k \boldsymbol{t}_j^{(k)}. \tag{4}$$

In order to improve the accuracy of singular vectors, Eq. (4) is implemented to the combination with the QR factorization [6].

As seen in Algorithm 1, the GKLR algorithm must be parallelized in terms of the computations on each line. In general, we parallelize them in terms of each the BLAS operations.

To improve the orthogonality of the basis 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 larger than the other processes of the GKLR, as the iteration number increases. Thus, it is important to accelerate the reorthogonalization in the GKLR.

3. Reorthogonalization algorithms

In this section, at first, we consider three conventional reorthogonalization algorithms for the GKLR algorithm. These algorithms are parallelized in terms of the BLAS operations in recent days. Secondly, we present the OpenMP-based parallel implementation of the CGS2 algorithm for shared-memory multicore processors and describe the advantage of this implementation with respect to the data usability.

In the followings, we discuss the computation of $\boldsymbol{x}_i \in \mathbb{R}^m$, the reorthogonalized vector of $\boldsymbol{a}_i \in \mathbb{R}^m$ $(2 \le i \le n)$, where satisfies $\langle \boldsymbol{x}_i,\ \boldsymbol{x}_k \rangle = 0$ for $j \ne k$. In addition, let X_{i-1} be $X_{i-1} = \begin{bmatrix} \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_{i-1} \end{bmatrix}$ $(2 \le i \le n)$. Note that X_{i-1}, $\boldsymbol{x}_i$, and $\boldsymbol{a}_i$ correspond to P_k, $\tilde{\boldsymbol{p}}$, and $\boldsymbol{p}$ on line 6 in Algorithm 1, and also correspond to Q_k, $\tilde{\boldsymbol{q}}$, and $\boldsymbol{q}$ on line 9 in Algorithm 1.

3.1 BLAS-based parallel implementation algorithms

3.1.1 CGS2 algorithm

The classical Gram-Schmidt (CGS) algorithm [4] is a well-known reorthogonalization algorithm. The reorthogonalization of $\boldsymbol{a}_i$ using the CGS algorithm is formulated as follows:

$$\boldsymbol{x}_i = \boldsymbol{a}_i - \sum_{k=1}^{i-1} \langle \boldsymbol{x}_k,\ \boldsymbol{a}_i \rangle \boldsymbol{x}_k. \tag{5}$$

Eq. (5) is composed of the level 1 BLAS operations, such as inner-dot products and AXPY operations. Using the matrix-vector multiplications, Eq. (5) is also replaced as

$$\boldsymbol{x}_i = \boldsymbol{a}_i - X_{i-1} X_{i-1}^\top \boldsymbol{a}_i. \tag{6}$$

In general, the level 2 BLAS operations, such as the matrix-vector multiplications, achieve the higher performance in parallel computing than the level 1 BLAS operations. Thus, the CGS algorithm 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, CGS algorithm with reorthogonalization (CGS2 algorithm) [2] is proposed, which repeats the CGS algorithm twice.

3.1.2 MGS algorithm

Another variant of the CGS algorithm is the modified Gram-Schmidt (MGS) algorithm. The MGS algorithm is composed of the level 1 BLAS operations, such as inner-dot product and AXPY operations. However, compared with the CGS, the MGS improves the orthogonality. Furthermore, the computational cost of the MGS is half as high as that of the CGS2.

3.1.3 Compact WY algorithm

The Householder transformations [4] are also used for the reorthogonalization. Yamamoto et al. [9] proposed a reorthogonalization algorithm using the Householder transformations in terms of the compact WY representation [8]. Hereafter, this algorithm is referred to as the cWY algorithm. In this algorithm, we can rewrite the product of the Householder matrices in a simple block matrix form. Hence, the cWY can be performed mainly using the level 2 BLAS operations. This algorithm can achieve the high orthogonality theoretically and high scalability in parallel computing.

Algorithm 2 OpenMP-based parallel implementation of CGS2 algorithm

```
1: function OMP-CGS2(X_{i-1}(= [x_1, ..., x_{i-1}]), a_i)
2:     #omp parallel private(j, s)
3:     do j = 1, 2
4:         #omp single
5:         w = a_i                                   ▷ Perform serially
6:         #omp end single
7:         #omp do reduction(+:a_i)
8:         do k = 1 to i − 1
9:             s = −⟨x_k, w⟩
10:            a_i = a_i + s x_k                     ▷ Array reduction
11:        end do
12:        #omp end do
13:    end do
14:    #omp end parallel
15:    return x_i = a_i
16: end function
```

3.2 OpenMP-based parallel implementation of CGS2 algorithm

Recalling Eq. (5), 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 2. Note that where $\boldsymbol{w}$ is a vector where preserves the original vector of $\boldsymbol{a}_i$. Hereafter, this implementation of the CGS2 algorithm is referred to as the OMP-CGS2 algorithm.

The parallel computation in terms of the summation is represented as the parallelism of do-loop as shown in line 7. As the result, the inner-dot product (line 9) and the AXPY operations (line 10) in terms of the different index k are performed on each thread. In addition, the array reduction must be implemented for the summation of $\boldsymbol{a}_i$ on line 10. 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 $\boldsymbol{a}_i = \boldsymbol{a}_i + s\boldsymbol{x}_k$ (line 10) as soon as $s = -\langle \boldsymbol{x}_k, \boldsymbol{w} \rangle$ (line 9) is computed, the reusability of $\boldsymbol{w}$, $\boldsymbol{x}_k$, 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 $\boldsymbol{w}$, $\boldsymbol{x}_k$, and $\boldsymbol{a}_i$ are stored in the L3 cache of each CPU.

4. Numerical experiments

In this section, we report results of numerical experiments 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 elapsed 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 required singular triplets; $l = 100,\ 200,\ 400,\ 800$.

Table 1: 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: 20MB × 4
RAM	DDR3-1066 1.5 TB, 136.4GB/sec
Compiler	Intel C++/Fortran Compiler 14.0.2
Options	`-O3 -xHOST -ipo -no-prec-div`
	`-openmp -mcmodel=medium -shared-intel`
Software	Intel Math Kernel Library 11.1.2

Table 2: The number of iterations at the point (k_{end}), where the GKLR algorithm stops, needed in each of the experiments. l denotes the number of required singular triplets.

l	100	200	400	800
Matrix T_1	1,000	1,600	2,400	4,000
Matrix T_2	1,300	2,000	3,200	4,800
Matrix T_3	1,600	2,400	3,600	5,600

We compare the elapsed time for computing subsets of singular triplets using the following four codes of the GKLR algorithms. **GKLR with MGS** is implemented with the MGS algorithm. **GKLR with CGS2** is implemented with the CGS2 algorithm. **GKLR with cWY** is implemented with the cWY algorithm. The reorthogonalization algorithms of the above three code are parallelized in terms of the BLAS routines. **GKLR with OMP-CGS2** is implemented with the OpenMP-based parallel implementation of the CGS2 algorithm.

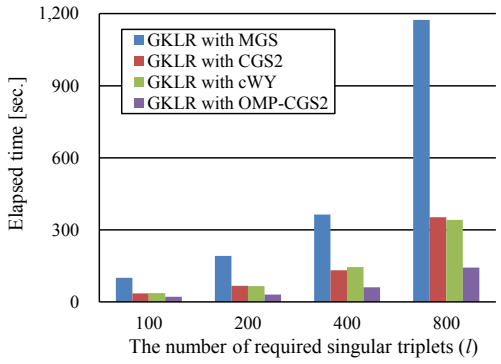
In the experiments, we use three $m \times n$ real sparse matrices T_1, T_2, and T_3. All of T_1, T_2, and T_3 are set to be 256 non-zero elements, which are set to be random numbers in the range $(0, 1)$ and are randomly allocated, in each row. T_1, T_2, and T_3 are only different in the size of m and n from each other as follows: $m = 16,000$ and $n = 8,000$ for T_1. $m = 32,000$ and $n = 16,000$ for T_2. $m = 64,000$ and $n = 32,000$ for T_3. In addition, the condition number is 4.803×10^1 for T_1, 4.754×10^1 for T_2, and 4.757×10^1 for T_3, respectively.

Finally, all the experiments are run with 32 threads on a machine shown in Table 1. We use the Intel Math Kernel Library (MKL) [5] for parallelizing the level 2 and level 3 BLAS routines. The Intel MKL also provides the level 1 BLAS routines, but the implementation depends on the dimension of the target vectors and the performance of them is unstable. Thus, we use the hand-made level 1 BLAS routines, which is parallelized by using OpenMP, in the experiments.

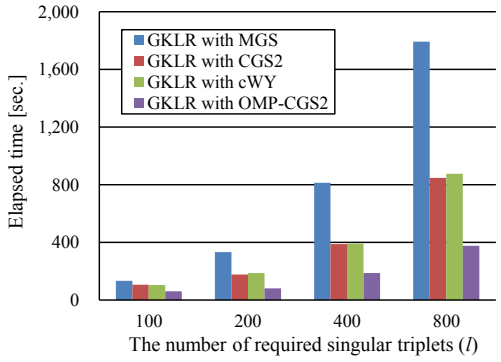
4.2 Results of performance evaluation

Figs. 1a, 1b, and 1c graph the experimental results and shows the number of required singular triplets and the elapsed time for computing singular triplets of each target matrix T_1, T_2, or T_3 using the four code of the GKLR algorithm, respectively. From the figures, **GKLR with OMP-CGS2** is faster than the other code in all the cases. Thus, the OMP-CGS2 accelerates the computation of the GKLR algorithm more effectively than the other reorthogonalization algorithms.

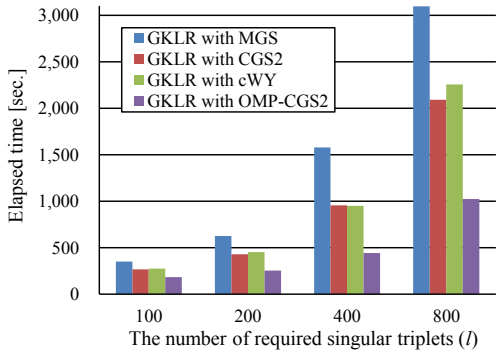
Note that the number of iterations at the point (k_{end}), where the GKLR algorithm stops, is the same regardless to the reorthogonalization algorithms in each of the experiments. Table 2 summarizes k_{end} needed in each of the experiments.



(a) Cases of T_1



(b) Cases of T_2



(c) Cases of T_3

Fig. 1: The number of required singular triplets and the elapsed time spending in the computation of the l largest singular triplets for each of the target matrices using the GKLR algorithm with different reorthogonalization implementation.

4.3 Discussion about cache use in OMP-CGS2

As mentioned in Sec. 3.2, the high performance of OMP-CGS2 arises from the higher reusability of cache in CPU. Here, we discuss the limit size of the vectors when we perform the reorthogonalization by using OMP-CGS2. Let the number of threads in a CPU be T and the capacity of L3 cache in the CPU be C MB. The one data of the elements needs 8 bytes when we use a double-precision floating-point number.

Recalling Algorithm 2, the vectors $\boldsymbol{w}$, $\boldsymbol{a}_i$, and $\boldsymbol{x}_k$ appear at each of **do**-loop in terms of k. If all these vectors are stored in the L3 cache of CPU, we can achieve the higher performance of the reorthogonalization by using OMP-CGS2. However, $\boldsymbol{x}_k$ 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 the results, the number of the vectors which should be stored in the cache is $(T \times 2 + 1)$.

From the above discussion, the dimension of the matrix which achieves better performance in this environment is determined by the following inequality:

$$m \times (T \times 2 + 1) \times 8 \leq C \times 1024 \times 1024, \qquad (7)$$

where m is the size of the vectors $\boldsymbol{w}$, $\boldsymbol{a}_i$, and $\boldsymbol{x}_k$. Then, since $T = 8$ and $C = 20$ from the specification of the CPUs used for the performance evaluation in this paper, the following inequality holds:

$$m \leq 154202. \qquad (8)$$

Thus, under the condition (8) of the performance evaluation in the experimental environment in Table 1, the OMP-CGS2 algorithm is guaranteed to achieve the higher performance than the other reorthogonalization algorithms.

5. Conclusions and future work

In this paper, we first introduce the GKLR algorithm for computing a subset of singular triplets for target matrices. To accelerate the reorthogonalization of the GKLR algorithm on shared-memory multi-core processors more effectively, we then present the OpenMP-based parallel implementation of the CGS2 algorithm. The OpenMP-based implementation of the CGS2 algorithm has the advantage of the data reusability. Experimental results on shared-memory multi-core processors show that the OpenMP-based implementation of the CGS2 algorithm accelerates the GKLR algorithm more effectively for computing a subset of singular triplets for a sparse matrix than other reorthogonalization algorithms.

Future work is to evaluate the performance of the GKLR algorithms for larger target matrices than those we used in the performance evaluation and to extend and confirm the validity of the modeling inequality (7) depending on CPUs.

Acknowledgment

In this work, we used the supercomputer of ACCMS, Kyoto University. This work was supported by JSPS KAKENHI Grant Numbers 13J02820 and 24360038.

References

[1] Barlow, J. L.: Reorthogonalization for the Golub-Kahan-Lanczos bidiagonal reduction, *Numer. Math.*, pp. 1–42 (2013).
[2] 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).
[3] 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).
[4] Golub, G. H. and van Loan, C. F.: *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, USA (1996).
[5] Intel Math Kernel Library: Available electronically at https://software.intel.com/en-us/intel-mkl/ (2003).
[6] Lehoucq, R. B., Sorensen, D. C. and Yang, C.: *ARPACK Users's Guide*, SIAM, Philadelphia, PA, USA (1998).
[7] OpenMP: Available electronically at http://openmp.org/wp/ (1997).
[8] 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).
[9] 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).