Doctor's Thesis

Studies on Parallel Solvers Based on Bisection and Inverse Iteration for Subsets of Eigenpairs and Singular Triplets

Hiroyuki ISHIGAMI

Department of Applied Mathematics and Physics

Graduate School of Informatics

Kyoto University

March 2016

Contents

1 Introduction

An eigenpair of an $n \times n$ real symmetric matrix A is the pair of a real constant $\lambda_k \in \mathbb{R}$ such that $\lambda_1 \geq \cdots \geq \lambda_n$ and a non-zero vector $\boldsymbol{q}_k \in \mathbb{R}^n$ for $k = 1, \ldots, n$, which satisfies

$$A\boldsymbol{q}_k = \lambda_k \boldsymbol{q}_k, \tag{1.1}$$

where λ_k is referred to as an eigenvalue of A, and $\boldsymbol{v}_k$ is referred to as the corresponding eigenvector to λ_k. A subset of eigenpairs of the real symmetric matrix is required in several applications, such as the vibration analysis, the molecular orbital computation in quantum chemistry, and the kernel principal component analysis [1] for multivariate data.

In addition, a subset computation of singular triplets, i.e., singular values and the corresponding singular vectors, of an $m \times n$ real rectangular matrix M such that $\mathrm{rank}(M) = r$ ($r \leq \min(m, n)$) is also one of the most important linear algebra operations. Assuming that $\sigma_k \in \mathbb{R}$ is one of the singular values of M such that $\sigma_1 \geq \cdots \geq \sigma_r > 0$, and $\boldsymbol{u}_k \in \mathbb{R}^m$ and $\boldsymbol{v}_k \in \mathbb{R}^n$ are the corresponding left and right singular vectors to σ_k, respectively, the following relationship is satisfied:

$$M\boldsymbol{v}_k = \sigma_k \boldsymbol{u}_k, \tag{1.2}$$

$$M^\top \boldsymbol{u}_k = \sigma_k \boldsymbol{v}_k. \tag{1.3}$$

It is noted that a subset of the singular triplets can be computed through the computation of a subset of eigenpairs as shown in a previous study [2]. In particular, a subset of singular triplets associated with the ℓ largest singular values $\sigma_1, \ldots, \sigma_\ell$ is often required in low-rank matrix approximation [3] and multivariate data analysis such as principal component analysis [4] and least-squares regression. In such applications, the target matrix M is often large and sparse, and $\ell \ll \min(m, n)$ [5].

As the size of the target matrices becomes significantly larger, the parallel processing becomes crucial for reducing the overall execution time. In fact, the performance of recent processors is enhanced by increasing the number of cores per processor and not the clock rates of each processor. Recent high-performance supercomputer systems listed in TOP500 [6] have more than 100,000 computing units. In addition, multi-core processors are usually equipped with the latest server systems regardless of their scale. The computation on accelerators such as Intel Xeon Phi Co-processor and GPUs released by NVIDIA and AMD, which also have many computing units, has been attracting attention. These facts have led to an increase in the demand for high-performance parallel solvers for the subset computation of both eigenpairs and singular triplets.

The *Basic Linear Algebra Subprograms (BLAS)* [7, 8] are one of the fundamental frameworks in the numerical linear algebra. The performance of the BLAS routines depends on their implementation, and their optimal implementation depends on the

computing platforms. Thus, to overcome this difficulty, the well-tuned BLAS routines are provided in several numerical libraries and commonly used for achieving a high-performance computation. The parallel BLAS routines are also provided in Intel Math Kernel Library [9], GotoBLAS2 [10], and so on. The BLAS routines are categorized into three sets of routines: BLAS 1, BLAS 2, and BLAS 3, and these routines contain vector-vector operations, matrix-vector operations, and matrix-matrix operations, respectively. It is well known that the BLAS 2 routines are faster than the BLAS 1 routines if the numbers of floating-point operations in both routines are equal. Similarly, the BLAS 3 routines are faster than the others based on the same assumption. The parallel implementation of the BLAS routines also has the same properties. It should be noted that these properties are held if and only if all the following three conditions are true: 1) The size of the matrices and vectors appearing in the computation is not too small; 2) The speed of floating-point operations on processors is sufficiently higher than memory speed as is the case with the scalar computers; and 3) Sufficiently large caches are equipped with processors performing the computations.

Let us consider computing a subset of eigenpairs of a real symmetric full matrix A on the basis of the orthogonal similarity transformations using the Householder transformations or the Givens rotations [11]. In general cases, the eigenpairs of A are computed through the following three phases: The first phase is *tridiagonalization*, which involves the reduction of A to a symmetric tridiagonal form T by orthogonal similarity transformations. In the second phase, eigenpairs of T are computed by some suitable solvers. The last phase is *back-transformation*, which involves the determination of eigenvectors of A from the computed eigenvectors of T by using the orthogonal transformation generated in the first phase. Since the orthogonal similarity transformations are used in the first phase, the computed eigenvalues of T correspond to that of A. It should be noted that several algorithms for the tridiagonalization and back-transformation have been proposed for both serial and parallel computations and will be mentioned later in this chapter.

In the above-mentioned second phase, the following eigensolvers for T and their parallel implementation have been proposed: *QR iteration (QR)* algorithm [12], *divide-and-conquer (DC)* algorithm [13, 14], *bisection and inverse iteration (BI)* algorithm [15–17], and *multiple relatively robust representations (MRRR)* algorithm [18, 19]. The serial implementation of these algorithms is provided in *Linear Algebra PACKage (LAPACK)* [20, 21] library as `dsteqr`, `dstedc`, `dstevx`, and `dstemr` for double precision arithmetic, respectively.

The QR algorithm [12] is known to be reliable in terms of stability and accuracy. The DC algorithm [13, 14] divides an eigenproblem recursively into two smaller sub-problems and computes eigenpairs of the sub-problem through a conquer process. Then the eigenpairs of T are obtained from the computed eigenpairs of each sub-problem. Since the DC algorithm is mainly composed of the BLAS 3 routines, it achieves a highly scalable performance in parallel processing [22] and is much faster than the QR algorithm. Unfortunately, the QR and DC algorithms always compute all the eigenpairs even when we need a small subset of them. Their computational cost is $O(n^3)$, where n is the size of T. From the viewpoint of the computational cost, these two algorithms may not be suitable if the number of the desired eigenpairs is much smaller than n. Therefore,

the BI and MRRR algorithms must be used to compute a small subset of the eigenpairs.

The BI algorithm [17] can compute the desired eigenvalues by using the bisection method [15] and the corresponding eigenvectors by using the inverse iteration [16]. The bisection algorithm can be massively parallelized and its computational cost is $O(\ell n)$ if the ℓ eigenpairs of T are desired. In addition, compared to the QR and DC algorithms, the bisection algorithm is guaranteed to always compute the eigenvalues of T with a high relative accuracy. The inverse iteration algorithm computes the eigenvectors of T by repeatedly solving linear equations with $O(\ell n)$ computational cost. However, the inverse iteration algorithm requires reorthogonalization to improve the orthogonality of the resulting eigenvectors, and thus, the computational cost increases to $O(\ell^2 n)$ in the worst case scenario. Note that the *modified Gram-Schmidt (MGS)* algorithm [11] is widely used for the reorthogonalization process of the inverse iteration algorithm as implemented in the `dstein` routine [17], which is provided in LAPACK.

The MRRR algorithm [18,19] is a variant of the inverse iteration algorithm. Owing to the avoidance of the reorthogonalization process, the computational cost of the MRRR algorithm is $O(\ell n)$ and less than that for the inverse iteration algorithm. In addition, the high-performance parallel implementation of the MRRR algorithm is also proposed [23, 24]. However, for the matrices that have extremely clustered eigenvalues such as the glued-Wilkinson matrix [25,26], the execution time for the MRRR algorithm is reported to be longer than that for the BI algorithm, as shown previously [25], even though the MRRR is superior with respect to the computational cost. In addition, `dstemr` [27], a LAPACK routine of the MRRR algorithm, can lose its accuracy, as shown in a previous study [28]. To solve this problem, the different formulation [28, 29] and the mixed precision approach [30] are proposed.

Due to the above-mentioned problems in the MRRR algorithm, it is important to work on the improvement of the BI algorithm for achieving the fast subset computation of eigenpairs of T with a high accuracy.

Moreover, recent studies on parallel symmetric eigensolvers have pointed out that the tridiagonalization phase is a bottleneck of the execution time in parallel processing from the viewpoint of both the communication cost and the computational cost. The tridiagonalization algorithm was first proposed in Wilkinson's study [31]. This algorithm transforms A into symmetric tridiagonal form T by using the Householder transformations and is, thus, referred to as *Householder tridiagonalization* algorithm. To improve the cache hit ratio and reduce the execution time for the tridiagonalization phase, the block version of the Householder tridiagonalization algorithm is proposed in [32].

As another approach to reducing the execution time for the tridiagonalization phase, the following two-step framework is proposed in [33, 34]: the first step is to reduce the full matrix A to a symmetric band form B and the second step is to reduce the band matrix B to a symmetric tridiagonal form T. Studies have proposed several efficient parallel algorithms for this framework [33, 35–38], and many of these algorithms are faster than the block Householder tridiagonalization algorithm. However, it is pointed out in [37] that the band reduction in the second step and its corresponding back-transformation remain to be the bottleneck of the execution time in massively parallel processing if the eigenvectors are required. Thus, recent studies have also proposed a highly scalable parallel algorithm based on the Householder tridiagonalization algorithm [39]

for massively parallel processing.

To overcome the problem in the tridiagonalization and the corresponding back-transformation, symmetric band eigensolvers without both of them have been proposed in [40–42]. The symmetric band eigensolver in [41] computes not only a subset of the eigenpairs but also all the eigenpairs because it employs the DC algorithm for B. However, the symmetric band eigensolvers in [40, 42] can compute only a subset of the eigenpairs of B since they employ the BI algorithm for B. In these eigensolvers, the implementations of the bisection algorithm for B are different from each other. In the followings, the bisection algorithms for B in [40] and [42] are referred to as *Gupta's bisection* algorithm and *Murata's bisection* algorithm, respectively. Although the computational cost of both of the bisection algorithms is $O(\ell w^2 n)$ where w is the bandwidth of B, Murata's bisection algorithm is expected to be faster than Gupta's bisection algorithm since Murata's bisection is better than Gupta's bisection in terms of the cache hit ratio. On the other hand, in these studies, the inverse iteration algorithm with reorthogonalization for B is commonly used for computing the corresponding eigenvectors. It should be noted that the inverse iteration algorithm with reorthogonalization for B, as well as that for T, includes the following two processes: the process to solve linear equations and that to reorthogonalize eigenvectors. Thus, the acceleration techniques of the inverse iteration algorithm for T can also be applied for accelerating the eigenvector computation by the inverse iteration algorithm for B. The computational cost for solving linear equations is $O(\ell w^2 n)$ and that for the reorthogonalization process is $O(\ell^2 n)$ in the worst cases. In other words, compared to the inverse iteration algorithm with reorthogonalization for T, solving linear equations may be a bottleneck in computing the eigenvectors of B unless ℓ is much larger than w. Consequently, a parallel algorithm for computing eigenvectors of B based on the inverse iteration is required to be efficient not only in its reorthogonalization but also in its linear equation solver.

As an extension of the eigenpair exploration discussed above, let us consider the subset computation of singular triplets associated with the ℓ largest singular values of an $m \times n$ real sparse matrix M on the basis of the *Golub-Kahan-Lanczos (GKL)* algorithm [2, 11]. The GKL algorithm is an iterative algorithm that uses two Krylov subspaces. At the k-th iteration, the GKL algorithm generates a $k \times k$ bidiagonal matrix B_k whose singular values approximate those of the target matrix M. If the singular values of B_k sufficiently approximate the desired singular values of M, the iteration of the GKL algorithm is terminated. For this criterion of the termination, it is required to find a subset of singular triplets of B_k. In addition, the corresponding singular vectors of M are obtained by singular vectors of B_k and the Krylov subspaces generated through the GKL algorithm. As per a previous study [2], algorithms for computing a particular subset of eigenpairs of real symmetric tridiagonal matrices, including the BI algorithm, are applicable for the above-mentioned subset computation of singular triplets of B_k. However, the GKL algorithm loses the orthogonality of the Krylov subspace because of computational errors. The *GKL algorithm with reorthogonalization (GKLR algorithm)* [43] employs the reorthogonalization process to improve the orthogonality of the vectors of the Krylov subspaces generated through the GKL algorithm. Although the GKLR algorithm is stable because of the reorthogonalization, the reorthogonalization tends to become a bottleneck in terms of computational cost and execution time even in parallel processing

as the number of iterations increases.

With this research background, this thesis focuses on the following three topics of efficient parallel algorithms to compute subsets of eigenpairs and singular triplets and their implementation on shared-memory multi-core processors:

1) The first topic is to develop a parallel solver based on the bisection and inverse iteration algorithms to compute a subset of eigenpairs of a given real symmetric tridiagonal matrix T. As mentioned earlier, the bottleneck of this computation in parallel processing is the computation of eigenvectors of T by using the inverse iteration algorithm with reorthogonalization. In particular, its reorthogonalization process is a dominant part in terms of computational cost, and thus, the efficient parallelization of this process is crucial for reducing the execution time. The conventional inverse iteration algorithm employs the MGS algorithm for the reorthogonalization process [16, 17] and is difficult to achieve a high performance in parallel processing. One of the reasons is that the MGS algorithm is composed of the BLAS 1 routines such as inner-dot product and scaled vector addition. Thus, as an approach to accelerating the inverse iteration algorithm in parallel processing, it is considered to reformulate this algorithm into that with higher-level BLAS routines. According to this approach, the following two algorithms for computing eigenvectors of T are proposed in this thesis:

The first algorithm is the inverse iteration algorithm with an alternative reorthogonalization [44], which is based on the Householder transformations with the compact WY representation [45]. In the followings, this reorthogonalization algorithm is referred to as the *compact WY reorthogonalization*. Since the compact WY reorthogonalization algorithm is mainly composed of the BLAS 2 routines such as the matrix-vector multiplications, the inverse iteration algorithm with the compact WY reorthogonalization is expected to achieve higher performance than the conventional inverse iteration algorithm. In this thesis, an alternative implementation of the compact WY reorthogonalization is also proposed, which employs suitable BLAS routines from the viewpoint of a mathematical structure appearing in the compact WY representation. In addition, both the computational cost and memory use for the proposed implementation is less than those for a naive BLAS implementation of the compact WY reorthogonalization. The second algorithm is the *block inverse iteration algorithm with reorthogonalization (BIR algorithm)*. The BIR algorithm is introduced as an improvement of the *simultaneous inverse iteration (or block inverse iteration) algorithm* [46], which is a variant of the inverse iteration algorithm, with a block parameter. Owing to the introduction of the block parameter, the BIR algorithm is mainly composed of the matrix multiplications and is expected to achieve the highly scalable performance. Thus, the BIR algorithm is also expected to accelerate the computation of eigenvectors especially in parallel processing.

2) The second topic is to propose a parallel solver for computing a subset of eigenpairs of real symmetric band matrices B. The proposed eigensolver for B includes the parallel implementation of Murata's bisection algorithm [42] and the BIR algorithm, which is proposed primarily for T in this thesis and then modified for the application to B. As a result, the BIR algorithm for B is parallelized with a lower communication cost than the conventional inverse iteration algorithms with reorthogonalization. Thus, the proposed eigensolver for B is expected to achieve higher parallel performance than the conventional eigensolvers for B.

3) The third topic is to develop a parallel solver for computing a subset of singular triplets of sparse matrices based on the combination of the GKLR and BI algorithms. As mentioned earlier, the reorthogonalization process of the GKLR algorithm remains to be a bottleneck in terms of the execution time in parallel processing. To overcome this difficulty, an alternative parallel implementation of the classical Gram-Schmidt algorithm with reorthogonalization (CGS2 algorithm) [47] is proposed in this thesis. The conventional parallel implementation of the CGS2 algorithm is composed of the BLAS 2 routines and is parallelized by employing the parallel BLAS routines. To the contrary, the proposed parallel implementation of the CGS2 algorithm is composed of the BLAS 1 routines and is parallelized by employing the OpenMP [48], and is referred to as the OMP-CGS2 algorithm. Since this parallelization effectively improves the cache hit ratio of CPUs, the OMP-CGS2 algorithm is expected to achieve the higher performance in parallel processing than the conventional reorthogonalization algorithms whose parallelization relies on that of the BLAS routines.

The rest of this thesis is organized as follows:

Chapters 2, 3, and 4 focus on the parallel solver based on the bisection and inverse iteration algorithms for computing a subset of eigenpairs of a given real symmetric tridiagonal matrix. Chapter 2 gives the introduction of the bisection and inverse iteration algorithms for real symmetric tridiagonal matrices and shows their parallel implementation. Chapter 3 reviews the compact WY reorthogonalization algorithm and discusses its implementation using the BLAS routines. In addition, an alternative implementation of the compact WY reorthogonalization is proposed by employing suitable BLAS routines, which is more economical than a naive implementation of it in terms of both the computational cost and the memory use. Then the performance of the inverse iteration algorithm with the compact WY reorthogonalization is evaluated through numerical experiments on shared-memory multi-core processors. Chapter 4 gives firstly a review of the simultaneous inverse iteration algorithm and then presents the BIR algorithm. The performance of both the BIR algorithm and the proposed symmetric tridiagonal solver, including the parallel bisection and BIR algorithms, is evaluated through numerical experiments on shared-memory multi-core processors.

Chapter 5 focuses on a parallel solver based on the bisection and inverse iteration algorithms for computing a subset of eigenpairs of a given real symmetric band matrix. This chapter gives firstly a review of the bisection and inverse iteration algorithms, as proposed in previous studies [40, 42], for computing a subset of eigenpairs of a real symmetric band matrix. Based on these algorithm, the proposed eigensolver computes the eigenvalues by using the parallel Murata's bisection algorithm and the corresponding eigenvectors by using the BIR algorithm for symmetric band matrices. Then the performance of the proposed eigensolver is evaluated through numerical experiments on shared-memory multi-core processors.

Chapter 6 focuses on a parallel solver for computing a subset of singular triplets of a given sparse matrix based on the combination of the GKLR and BI algorithms. This chapter reviews the GKLR algorithm and shows how to apply the bisection and inverse iteration algorithms to the GKLR. Then, the OMP-CGS2 algorithm is proposed for accelerating the reorthogonalization process of the GKLR algorithm in parallel processing. Finally, the performance of the proposed solver for computing singular

triplet subsets of real sparse matrices is evaluated through numerical experiments on shared-memory multi-core processors.

Chapter 7 is devoted to conclusions of this thesis.

2 Subset Computation of Eigenpairs of Real Symmetric Tridiagonal Matrices

Let T be an $n \times n$ real symmetric tridiagonal matrix with diagonals $d_1, \ldots, d_n$ and subdiagonals $e_1, \ldots, e_{n-1}$, $\lambda_k \in \mathbb{R}$ be an eigenvalue of T such that $\lambda_1 > \lambda_2 > \cdots > \lambda_n$, and $\boldsymbol{q}_k \in \mathbb{R}^n$ be the corresponding eigenvector to λ_k, respectively. In addition, let ℓ be the number of desired eigenpairs of T ($\ell \leq n$).

2.1 Bisection Algorithm and Its Parallel Implementation

This section gives firstly an introduction to the bisection algorithm for T based on the `dstebz` routine and then presents the parallel bisection algorithm for shared-memory multi-core processors.

It is noted that the bisection algorithm [49] is to compute all or a subset of the eigenvalues of T and is provided as the `dstebz` routine [17] in LAPACK. Although the Intel Math Kernel Library (MKL) [9] provides the many parallel implementations of the LAPACK routines for shared-memory multi-core processors, the parallel `dstebz` routine is not provided even by the Intel MKL.

2.1.1 Bisection Algorithm

The bisection algorithm [15, 17] is to find approximated eigenvalues $\tilde{\lambda}_1 > \cdots > \tilde{\lambda}_\ell$ of T by repeatedly updating half-open intervals $(\mu^L, \mu^R]$ of eigenvalues. The computation of $\nu_T(\mu)$ is required for updating the intervals, where μ is a value in the intervals $(\mu^L, \mu^R]$ and $\nu_T(\mu)$ is the number of eigenvalues of T that are less than μ. Considering the $LDL^\top$ factorization of $T - \mu I$, $\nu_T(\mu)$ corresponds to the number of negative values in elements of D on the basis of Sylvester's law of inertia [50].

Algorithm 2.1 shows a pseudocode of the serial bisection algorithm and is based on the `dstebz` routine [17] and its subroutine `dlaebz` provided in LAPACK, where c_k corresponds to $\nu_T(\mu_k)$ and is stored in the k-th elements of the integer array $\boldsymbol{c}$ for the implementation. Lines 8-12 is to compute c_k by the recurrence derived from the $LDL^\top$ factorization of $T - \mu I$. Note that we let $e_0 = 0$ for convenience. For more details, see the `dstebz` and `dlaebz` routines in LAPACK.

The Initialization of μ_1^L and μ_1^R in line 2 is described as follows: At first, μ_1^L and μ_1^R are given by lower and upper bounds of eigenvalues of T based on the Gerschgorin theorem [11]. Then, μ_1^L and μ_1^R are iteratively refined by the update of $\nu_T(\mu_1^L)$ and $\nu_T(\mu_1^R)$ in the way similar to that shown in lines 7-13. After these computations, we hold that $\nu_T(\mu_1^L) = 0$ and $\nu_T(\mu_1^R) = \ell$.

Algorithm 2.1 Serial Bisection Algorithm for A Real Symmetric Tridiagonal Matrix T

```
1:  function SerialTriBisection(T, ℓ)
2:      Set μ_1^L, μ_1^R                          ▷ Use the Gerschgorin theorem, etc.
3:      k_b^next := 1, k_e^next := 1
4:      repeat
5:          k_b := k_b^next, k_e := k_e^next
6:          do k := k_b to k_e                                          ▷ k_e ≤ ℓ
7:              μ_k := (μ_k^L + μ_k^R)/2
8:              c_k := 0, r := 1
9:              do j := 1 to n
10:                 r := d_j − e_{j−1}^2/r − μ_k (e_0 := 0)
11:                 if r ≤ 0 then c_k := c_k + 1
12:             end do
13:             Update μ_k^L, μ_k^R, and k_e^next (See Table 2.1)
14:         end do
15:         do k := k_b to k_e
16:             if (μ_k^L, μ_k^R] contains only one eigenvalue of T and |μ_k^L − μ_k^R| < δ, then
17:                 μ_k^L ↔ μ_{k_b^next}^L, μ_k^R ↔ μ_{k_b^next}^R, and k_b^next := k_b^next + 1
18:             end if
19:         end do
20:     until k_b^next > k_e^next
21:     μ_k := (μ_k^L + μ_k^R)/2 for k = 1, ..., ℓ
22:     Sort μ_k for k = 1, ..., ℓ in descending order
23:     return λ̃_k := μ_k for k = 1, ..., ℓ
24: end function
```

Line 13 corresponds to update μ_k^L and μ_k^R for the next **repeat**-**until** iteration of lines 4-20. The update rule is based on the magnitude relation among $\nu_T(\mu_k^L)$, $\nu_T(\mu_k)$, and $\nu_T(\mu_k^R)$ and is shown in Table 2.1. This update rule is based on the following ideas: If $\nu_T(\mu_k) = \nu_T(\mu_k^L)$, any eigenvalue of T does not exist in $(\mu_k^L, \mu_k]$ and thus we set to $\mu_k^L := \mu_k$ for the next iteration. Similarly, if $\nu_T(\mu_k) = \nu_T(\mu_k^R)$, any eigenvalue of T does not exist in $(\mu_k, \mu_k^R]$ and thus we set to $\mu_k^R := \mu_k$ for the next iteration. In these cases, the number of the intervals to be searched is unchanged. To the contrary, if $\nu_T(\mu_k) \neq \nu_T(\mu_k^L)$ and $\nu_T(\mu_k) \neq \nu_T(\mu_k^R)$, some of the eigenvalues exist in both $(\mu_k^L, \mu_k]$ and $(\mu_k, \mu_k^R]$ and the number of the intervals to be searched in the next iteration increases. Since $\nu_T(\mu_k^L) \neq \nu_T(\mu_k^R)$ are always guaranteed by both the update rule shown in Table 2.1 and the initialization of μ_1^L and μ_1^R described earlier, we need not consider the case when $\nu_T(\mu_k) = \nu_T(\mu_k^L) = \nu_T(\mu_k^R)$. It should be noted that we must store all of $\nu_T(\mu_k^L)$ and $\nu_T(\mu_k^R)$ in the working memory for reducing the computational cost and update them in conjunction with the update of μ_k^L and μ_k^R.

Lines 15-19 shows operations for terminating the **repeat**-**until** iteration of lines 4-20. To terminate this iteration, the update of the intervals $(\mu_k^L, \mu_k^R)]$ for $k = k_b, \ldots, k_e$ are also terminated. For this purpose, we check whether both of the following two conditions

Table 2.1: The update rule of μ_k^L and μ^R in the bisection algorithm for computing eigenvalues of a real symmetric matrix T [17], where $\nu_T(\mu)$ denotes the number of eigenvalues of T that are less than μ and k_e^{next} denotes the largest index of μ_k in the next iteration of the bisection algorithm. Note that $\nu_T(\mu_k)$ corresponds to c_k in Algorithm 2.1.

Condition	Operations
$\nu_T(\mu_k) = \nu_T(\mu_k^L)$	$\mu_k^L := \mu_k$
$\nu_T(\mu_k) = \nu_T(\mu_k^R)$	$\mu_k^R := \mu_k$
$\nu_T(\mu_k) \neq \nu_T(\mu_k^L)$ and $\nu_T(\mu_k) \neq \nu_T(\mu_k^R)$	1. $k_e^{\text{next}} := k_e^{\text{next}} + 1$ 2. $\mu_{k_e^{\text{next}}}^L := \mu_k$, $\mu_{k_e^{\text{next}}}^L := \mu_k^R$ 3. $\mu_k^R := \mu_k$

in terms of the intervals $(\mu_k^L, \mu_k^R)]$ are satisfied or not in line 16: 1) $(\mu_k^L, \mu_k^R]$ contains only one eigenvalue of T; 2) $\mu_k^L - \mu_k^R|$ is sufficiently small. In the `dlaebz` routine, δ in line 16 is set to a small value such as $2\epsilon \max(|\mu_k^L|, |\mu_k^R|)$, where ϵ is the machine epsilon; for more details, see the `dlaebz` routine. If these conditions are satisfied, we exchange the pair μ_k^L and μ_k^R for the pair $\mu_{k_b^{\text{next}}}^L$ and $\mu_{k_b^{\text{next}}}^R$ and update k_b^{next} as shown in line 17. Through this process, we exclude the interval $(\mu_k^L, \mu_k^R]$ from the search in the next **repeat**-**until** iteration. Finally, as shown in line 20, we terminate the **repeat**-**until** iteration if $k_b^{\text{next}} > k_e^{\text{next}}$ is satisfied to mean all intervals have become sufficiently small in terms of and thus have been excluded.

2.1.2 Parallel Implementation of Bisection Algorithm

Several parallel implementations of the bisection algorithm are proposed: The one is a method proposed in [51] and is provided by the `pdstebz` routine in ScaLAPACK (Scalable Linear Algebra PACKage) [52]. The `pdstebz` routine is parallelized in terms of the indices of eigenvalues since the computation of each eigenvalue is independent. This implementation requires $O(1)$ of synchronization, and thus is expected to achieve highly scalable performance of distributed memory parallel systems. However, the computational cost increases from the baseline serial computation because in general the computation for an eigenvalue overlaps with others to make the overlapped portions redundant. For parallel computation on shared-memory multi-core processors, in particular, this increase of the computational cost is likely to degrade parallel performance more severely than that of the synchronization cost which is in general much smaller than that in distributed memory systems. Therefore, the parallel implementation of `pdstebz` is not suitable for shared-memory computing environments such as multi-core processors.

Now let us consider the parallelization of the most time-consuming part of the bisection algorithm, i.e., the computations of lines 4-20 in Algorithm 2.1, to accelerate effectively the bisection algorithm in parallel processing. In Algorithm 2.1, the iterations of **do**-loop

Algorithm 2.2 Parallel Bisection Algorithm for A Real Symmetric Tridiagonal Matrix T

```
 1: function ParallelTriBisection(T, ℓ)
 2:     Set μ_1^L, μ_1^R                          ▷ Use the Gerschgorin theorem, etc.
 3:     k_b := 1, k_e := 1
 4:     repeat
 5:         !$omp parallel do private(j, μ_k, r, c_t)
 6:         do k := k_b to k_e                                          ▷ k_e ≤ ℓ
 7:             μ_k := (μ_k^L + μ_k^R)/2
 8:             c_t := 0, r := 1
 9:             do j := 1 to n
10:                 r := d_j − e_{j−1}^2/r − μ_k (e_0 := 0)
11:                 if r ≤ 0 then c_t := c_t + 1
12:             end do
13:             c_k := c_t
14:         end do
15:         do k := k_b to k_e
16:             Update μ_k^L, μ_k^R, and k_e^next (See Table 2.1)
17:         end do
18:         Check μ_k^L and μ_k^R for k = k_b, ..., k_e & Update μ_k^L, μ_k^R, and k_b^next if necessary
            (See lines 15-19 in Algorithm 2.1)
19:     until k_b^next > k_e^next
20:     μ_k := (μ_k^L + μ_k^R)/2 for k = 1, ..., ℓ
21:     Sort μ_k for k = 1, ..., ℓ in descending order
22:     return λ̃_k := μ_k for k = 1, ..., ℓ
23: end function
```

of lines 6-14 are mutually independent in terms of k. Thus, this computation is easily parallelized using the OpenMP directive. It should be noted that the false sharing may occur with respect to the array $\boldsymbol{c}$ if we parallelize the loop straightforwardly. To avoid the false sharing, we introduce a temporary variable c_t private to each thread. In addition, the atomic execution would be needed for the update of μ_k^L, μ_k^R, k_b^{next}, and k_e^{next} if we parallelized the operation of lines 13 and the loop of 15-19. Since their computational cost is significantly less than that for the computation of the **do**-loop for k of lines 6-14, the operation and loop are performed serially.

As a result, the pseudocode of the parallel bisection algorithm in this thesis is obtained as shown in Algorithm 2.2. It should be noted that its computational cost is almost equal to that for Algorithm 2.1.

2.2 Inverse Iteration Algorithm and Its Parallelization

This section gives an introduction to the inverse iteration algorithm with reorthogonalization and then discusses the parallel implementation of it.

Algorithm 2.3 Conventional Inverse Iteration Algorithm for A Real Symmetric Tridiagonal Matrix T

```
1: function TriInv(T, ℓ, λ̃_1, ..., λ̃_ℓ)
2:     do k := 1 to ℓ
3:         i := 0
4:         Generate an initial random vector v_k^(0)
5:         T − λ̃_k := P_k L_k U_k                          ▷ Call dlagtf
6:         repeat
7:             i := i + 1
8:             Normalize v_k^(i−1) to q_k^(i−1)
9:             Solve P_k L_k U_k v_k^(i) = q_k^(i−1)       ▷ Call dlagts
10:            if k > 1 and |λ̃_{k−1} − λ̃_k| ≤ 10^{−3}‖T‖_1, then
11:                do j := k_1 to k − 1
12:                    v_k^(i) := v_k^(i) − ⟨v_k^(i), q_j⟩ q_j
13:                end do
14:            else
15:                k_1 := k
16:            end if
17:        until some condition is met.
18:        Normalize v_k^(i) to q_k^(i)
19:        Q_k := [Q_{k−1}, q_k^(i)]
20:    end do
21:    return Q_ℓ = [q_1, ..., q_ℓ]
22: end function
```

2.2.1 Inverse Iteration Algorithm with Reorthogonalization

Let $\tilde{\lambda}_k$ be an approximate value of λ_k, which is computed in advance by some algorithm such as the bisection algorithm. Given a starting vector $\boldsymbol{v}_k^{(0)}$ whose elements are set by random real numbers, the inverse iteration for T generates a sequence of vectors $\boldsymbol{v}_k^{(i)}$ as follows:

$$\left(T - \tilde{\lambda}_k I\right) \boldsymbol{v}_k^{(i)} = \boldsymbol{v}_k^{(i-1)}, \quad i = 1, 2, \ldots, \tag{2.1}$$

where I is the $n \times n$ identity matrix. For solving numerically the linear equation (2.1), we first factorize $T - \tilde{\lambda}_k I$ by the *LU factorization with the partial pivoting (PLU factorization)*, and then compute $\boldsymbol{v}_k^{(i)}$ by using the resulting elements. In addition, we should normalize the vectors $\boldsymbol{v}_k^{(i)}$ to avoid overflow. Since the result of the *PLU* factorization is invariant for all series of computations of $\boldsymbol{v}_k^{(}i)$ for a certain $\tilde{\lambda}_k$, we keep the factorization form in a working store so as to reuse it. For this purpose, we have to store them in the working memory.

If the eigenvalues of T are well separated, $\boldsymbol{v}_k^{(i)}$ quickly converges to $\boldsymbol{q}_k$ by a few iterations. As a result, the computational cost to solve (2.1) for ℓ eigenpairs of T is $O(\ell n)$ ($\ell \leq n$).

By contrast, if some of the eigenvalues of T are close to one another, the eigenvectors obtained from Eq. (2.1) sometimes loses their orthogonality. Hereafter, such eigenvalues are referred to as a *clustered eigenvalues*. To improve the orthogonality of the corresponding eigenvectors to these eigenvalues, the reorthogonalization process is added to the inverse iteration algorithm [49]. For dividing eigenvalues into clusters, *Peters-Wilkinson's criterion* [53] is widely used. In Peters-Wilkinson's criterion, λ_{k-1} and λ_k are regarded as belonging to the same cluster ($2 \leq k \leq \ell$) if the following condition is satisfied:

$$|\tilde{\lambda}_{k-1} - \tilde{\lambda}_k| \leq 10^{-3}\|T\|_1. \tag{2.2}$$

Several algorithms have been proposed for the reorthogonalization process, but their computational cost is $O(\hat{\ell}^2 n)$, where $\hat{\ell}$ is the number of vectors to be reorthogonalized. The MGS algorithm is widely used for the reorthogonalization process of the inverse iteration algorithm. The inverse iteration algorithm with reorthogonalization which employs Peters-Wilkinson's criterion and the MGS algorithm is widely used, and its implementation is provided in LAPACK as the `dstein` routine [17].

Algorithm 2.3 shows a pseudocode of the conventional inverse iteration algorithm for T, where k_1 denotes the index of the largest eigenvalue of a certain cluster. As described before, the process to solve (2.1) is divided into the two computations of lines 5 and 9. Line 5 corresponds to the computation of the PLU factorization and employs the `dlagtf` routine. Line 9 corresponds to the computation of $\boldsymbol{v}_k^{(i)}$ by using the result of the PLU factorization and employs the `dlagts` routine. Both of these routines are employed in the `dstein` and are provided in LAPACK. In addition, Peters-Wilkinson's criterion is given in line 10. The reorthogonalization of eigenvectors by the MGS algorithm corresponds to lines 11-13. As can be seen in line 12, the MGS algorithm is composed of the BLAS 1 routines such as inner-dot product and scaled vector addition.

2.2.2 Parallelism of Inverse Iteration Algorithm

If the reorthogonalization is not required, the inverse iteration, i.e., solving linear equation (2.1), can be parallelized in terms of the computation of the corresponding eigenvector to each eigenvalue of T since linear equation (2.1) is independent in terms of $\tilde{\lambda}_k$. Otherwise, the inverse iteration algorithm with reorthogonalization is considered to be parallelized by assigning the eigenvector computation for each cluster to a CPU core if T has many clusters of eigenvalues.

However, it has been pointed out that most of the eigenvalues are regarded as belonging to the same cluster if we adopt Peters-Wilkinson's criterion to the inverse iteration algorithm with reorthogonalization and the size of T is greater than 1,000 [18]. In other words, the number of eigenvalues of T belonging to the largest cluster becomes close to ℓ if the size of T becomes large. In such a situation, the inverse iteration algorithm with reorthogonalization should be parallelized not in terms of clusters but rather in terms of computation loop of lines 6-17 in Algorithm 2.3, which includes the inverse iteration (2.1) and the reorthogonalization by the MGS algorithm. More accurately, the inverse iteration algorithm with reorthogonalization should be parallelized in terms of the computations of each line in Algorithm 2.3.

However, it is well known that the conventional inverse iteration algorithm shown in Algorithm 2.3 is difficult to be accelerated effectively by the above-mentioned parallelization since the reorthogonalization process becomes a bottleneck in parallel processing. This is in accordance with the discussion on the computational cost in Section 2.2.1. Thus, a key to reduce the overall execution time for the inverse iteration algorithm in parallel processing is to accelerate efficiently the computation of the reorthogonalization process. As one of the approaches for this purpose, we can consider the implementation of alternative reorthogonalization algorithms, which achieve better parallel performance than the MGS algorithm, to the inverse iteration algorithm. Recalling the discussion on the parallel BLAS routines in Chapter 1, the reorthogonalization algorithms composed of the BLAS 2 or 3 routines may achieve the higher performance than the MGS algorithm.

3 Inverse Iteration Algorithm with Compact WY Reorthogonalization

In this chapter, an alternative inverse iteration algorithm is proposed for accelerating the subset computation of eigenvectors of T in parallel processing. The proposed inverse iteration algorithm employs a reorthogonalization algorithm [44], which is based on the Householder transformations in terms of the *compact WY representation* [45]. Hereafter, let us name this reorthogonalization algorithm *compact WY reorthogonalization*. Compared to the MGS algorithm, which is employed in the conventional inverse iteration algorithm shown in Algorithm 2.3, the compact WY reorthogonalization is mainly composed of the BLAS 2 routines such as the matrix-vector multiplications. According to this fact and the discussion on the parallel BLAS routines in Chapter 1, the proposed inverse iteration algorithm is expected to achieve higher performance than the conventional inverse iteration algorithm.

In addition, an alternative BLAS-based implementation of the compact WY reorthogonalization is proposed in this chapter. From the viewpoint of the computational cost and the memory use, the proposed implementation is better than a naive BLAS-based implementation of the compact WY reorthogonalization.

The rest of this chapter is organized as follows: Section 3.1 gives a review of the reorthogonalization algorithm based on the Householder transformations and the compact WY reorthogonalization algorithm. Section 3.2 discusses the implementations of the compact WY reorthogonalization on the basis of the BLAS routines, and then presents its proposed implementation. Section 3.3 shows the implementation of the inverse iteration algorithm with the compact WY reorthogonalization. Section 3.4 provides experimental results on shared-memory multi-core processors to evaluate the performance of the proposed inverse iteration algorithm.

3.1 Compact WY Reorthogonalization

This section gives the introduction of the compact WY reorthogonalization, i.e., the reorthogonalization algorithm on the basis of the Householder transformations in terms of the compact WY representation [45], which is proposed in [44].

In the followings, we discuss the computation of $\boldsymbol{q}_j \in \mathbb{R}^n$, which is the orthogonalized vector of $\boldsymbol{v}_j \in \mathbb{R}^n$ $(2 \leq j \leq k,\ k \leq n)$ and satisfies $\langle \boldsymbol{q}_j,\ \boldsymbol{q}_i \rangle = 0$ for $i \neq j$. In addition, let $\mathbf{0}_i$ be an i-dimensional zero vector.

Algorithm 3.1 Householder Reorthogonalization

```
1:  function HOUSEHOLDER(v_j, y_1, ..., y_{j-1}, t_1, ..., t_{j-1})
2:      u_j := (I - t_1 y_1 y_1^T) v_j
3:      do i := 2 to j - 1
4:          u_j := (I - t_i y_i y_i^T) u_j
5:      end do
6:      Compute y_j by using u_j and t_j := 2/||y_j||_2^2
7:      q_j := (I - t_j y_j y_j^T) e_j
8:      do i := j - 1 to 1
9:          q_j := (I - t_i y_i y_i^T) q_j
10:     end do
11:     return q_j
12: end function
```

3.1.1 Householder Reorthogonalization

As preliminaries, the reorthogonalization algorithm on the basis of the Householder transformations [11] is shown below. Hereafter, let us name this reorthogonalization algorithm *Householder reorthogonalization*.

The Householder transformations is a transformation given by the Householder matrices $H_j \in \mathbb{R}^{n\times n}$ $(j = 1, \ldots, k)$, which is defined as

$$H_j = I - t_j \boldsymbol{y}_j \boldsymbol{y}_j^\top, \tag{3.1}$$

where $t_j = 2/\|\boldsymbol{y}_j\|_2^2 \in \mathbb{R}$ and $\boldsymbol{y}_j = \boldsymbol{u}_j - \boldsymbol{w}_j \in \mathbb{R}^n$ such that $\|\boldsymbol{u}_j\|_2 = \|\boldsymbol{w}_j\|_2$, and satisfies $H_j H_j^\top = H_j^\top H_j = I$, $H_j \boldsymbol{u}_j = \boldsymbol{w}_j$.

The Householder reorthogonalization is shown in Algorithm 3.2, where $\boldsymbol{e}_j$ is the j-th column vector of the $n \times n$ identity matrix. It is noted that the computation of $\boldsymbol{y}_j$ in line 6 is composed as follows: Let us define the vectors $\boldsymbol{u}_j$ and $\boldsymbol{w}_j$ as

$$\boldsymbol{u}_j := \begin{bmatrix} u_{1,j} & \cdots & u_{j-1,j} & u_{j,j} & u_{j+1,j} & \cdots & u_{n,j} \end{bmatrix}^\top \tag{3.2}$$

$$\boldsymbol{w}_j := \begin{bmatrix} u_{1,j} & \cdots & u_{j-1,j} & c_j & \boldsymbol{0}_{n-j}^\top \end{bmatrix}^\top, \tag{3.3}$$

where $u_{i,j}$ $(i = 1, \ldots, n)$ is the i-th element of $\boldsymbol{u}_j$ and

$$c_j := -\operatorname{sgn}(u_{j,j}) \sqrt{\sum_{i=j}^{n} u_{i,j}^2}. \tag{3.4}$$

Hence, $\boldsymbol{y}_j$ is computed as

$$\boldsymbol{y}_j = \begin{bmatrix} \boldsymbol{0}_{j-1}^\top & u_{j,j} - c_j & u_{j+1,j} & \cdots & u_{n,j} \end{bmatrix}^\top. \tag{3.5}$$

As can be seen in Algorithm 3.1, the Householder reorthogonalization is mainly composed of the BLAS 1 routines as well as the MGS algorithm, which is implemented to the reorthogonalization of the conventional inverse iteration mentioned in Section 2.2.1. By contrast, the computational cost of the Householder reorthogonalization is about twice higher than that of the MGS algorithm.

Algorithm 3.2 Compact WY Reorthogonalization

1: **function** COMPACTWY($\boldsymbol{v}_j, Y_{j-1}, T_{j-1}$)
2: $\quad \boldsymbol{u}_j := \left(I - Y_{j-1}T_{j-1}^\top Y_{j-1}^\top\right)\boldsymbol{v}_j$
3: $\quad$ Compute $\boldsymbol{y}_j$ by using $\boldsymbol{u}_j$
4: $\quad t_j := 2/\|\boldsymbol{y}_j\|_2^2$
5: $\quad Y_j := \begin{bmatrix} Y_{j-1} & \boldsymbol{y}_j \end{bmatrix}, \quad T_j := \begin{bmatrix} T_{j-1} & -t_j T_{j-1} Y_{j-1}^\top \boldsymbol{y}_j \\ \boldsymbol{0}_{j-1}^\top & t_j \end{bmatrix}$
6: $\quad \boldsymbol{q}_j := \left(I - Y_j T_j Y_j^\top\right)\boldsymbol{e}_j$
7: $\quad$ **return** $\boldsymbol{q}_j$
8: **end function**

3.1.2 Compact WY Reorthogonalization

The implementation of the Householder reorthogonalization on the basis of the BLAS 2 routines is proposed in [44] by introducing the compact WY representation [45]. Hereafter let us name this implementation of the Householder reorthogonalization the *compact WY reorthogonalization*. The compact WY reorthogonalization is also theoretically demonstrated in [44] to achieve high orthogonality and high scalability in parallel processing.

Recalling the definition of the Householder transformations, the compact WY representation is introduced as follows: First, we define matrices Y_1 and T_1 as $Y_1 := [\boldsymbol{y}_1] \in \mathbb{R}^{n\times 1}$ and $T_1 := [t_1] \in \mathbb{R}^{1\times 1}$. Then we define recursively a rectangular matrix $Y_j \in \mathbb{R}^{n\times j}$ and an upper triangular matrix $T_j \in \mathbb{R}^{j\times j}$ as

$$Y_j := \begin{bmatrix} Y_{j-1} & \boldsymbol{y}_j \end{bmatrix}, \; T_j := \begin{bmatrix} T_{j-1} & -t_j T_{j-1} Y_{j-1}^\top \boldsymbol{y}_j \\ \boldsymbol{0}_{j-1}^\top & t_j \end{bmatrix}. \tag{3.6}$$

Using Eq (3.6), the product $H_1 H_2 \cdots H_j$ of the Householder matrices is represented in a simple block form as follows:

$$H_1 H_2 \cdots H_j = I - Y_j T_j Y_j^\top. \tag{3.7}$$

Here $I - Y_j T_j Y_j^\top$ is referred to as the compact WY representation of the product $H_1 H_2 \cdots H_j$.

By introducing the compact WY representation (3.7), the Householder reorthogonalization in Algorithm 3.1 can be written into the compact WY reorthogonalization shown in Algorithm 3.2. As can be seen in Algorithm 3.2, the compact WY reorthogonalization is mainly composed of the BLAS 2 routines such as the matrix-vector multiplications.

3.2 Implementation of Compact WY Reorthogonalization

In this section, we show a naive implementation of the compact WY reorthogonalization using the BLAS routines. We also discuss the mathematical structure of this algorithm and present a new implementation of the compact WY reorthogonalization that reduces computational cost.

3.2.1 Naive Implementation Based on BLAS Routines

Let us consider a naive BLAS-based implementation of the compact WY reorthogonalization in lines 2-6 of Algorithm 3.2. To use the BLAS routines here, we need to reformulate line 2 as follows:

$$\begin{aligned} \boldsymbol{u}_j &:= \left(I - Y_{j-1}T_{j-1}^\top Y_{j-1}^\top\right)\boldsymbol{v}_j && (3.8)\\ &= \boldsymbol{v}_j - Y_{j-1}T_{j-1}^\top Y_{j-1}^\top \boldsymbol{v}_j. && (3.9) \end{aligned}$$

We can now implement this formula by using the BLAS routines as follows:

$$\begin{cases} \boldsymbol{u}_j := \boldsymbol{v}_j & \texttt{(dcopy)} \\ \boldsymbol{x}_{j-1} := Y_{j-1}^\top \boldsymbol{u}_j + 0 \cdot \boldsymbol{x}_{j-1} & \texttt{(dgemv)} \\ \boldsymbol{x}_{j-1} := T_{j-1}^\top \boldsymbol{x}_{j-1} & \texttt{(dtrmv)} \\ \boldsymbol{u}_j := (-1) \cdot Y_{j-1}\boldsymbol{x}_{j-1} + \boldsymbol{u}_j & \texttt{(dgemv)} \end{cases}, \tag{3.10}$$

where $\boldsymbol{x}_{j-1} \in \mathbb{R}^{j-1}$. We set the initial memory address of $\boldsymbol{x}_{j-1}$ to correspond to that of $\boldsymbol{v}_j$. `dcopy` is a routine for copying a vector $\boldsymbol{x}$ to a vector $\boldsymbol{y}$: $\boldsymbol{y} := \boldsymbol{x}$. `dgemv` is a routine for the matrix-vector operation $\boldsymbol{y} := \alpha A\boldsymbol{x} + \beta\boldsymbol{y}$, where A is a general rectangular matrix and α, β are real constants. `dtrmv` is a routine for the multiplications $\boldsymbol{x} := T\boldsymbol{x}$ of a triangular matrix T and a vector.

Next, in line 3, we compute $\boldsymbol{y}_j$ in Eq. (3.5) and t_j. These computations are mostly performed using the BLAS 1 routines, so their cost is relatively low. We implement the computation of $\boldsymbol{y}_j$ and t_j as follows:

$$\begin{cases} y_{i,j} := 0, \quad (i = 1, \ \ldots, \ j-1) & \\ y_{i,j} := u_{i,j}, \quad (i = j, \ \ldots, \ n) & \texttt{(dcopy)} \\ c_j := -\operatorname{sgn}(u_{j,j})\sqrt{\sum_{i=j}^{n} u_{i,j}^2} & \texttt{(dnrm2)} \\ y_{j,j} := u_{j,j} - c_j & \\ t_j := 2/\|\boldsymbol{y}_j\|_2^2 & \texttt{(dnrm2)} \end{cases}, \tag{3.11}$$

where $y_{i,j}$ $(i = 1, \ldots, n)$ is the i-th column element of $\boldsymbol{y}_j$, and `dnrm2` is a routine for computing of the 2-norm of a vector.

In line 5, updating Y_j and t_j is easily accomplished. Let $\hat{\boldsymbol{t}}_j \in \mathbb{R}^{j-1}$ be $\hat{\boldsymbol{t}}_j = -t_j T_{j-1} Y_{j-1}^\top \boldsymbol{y}_j$. Note that $\hat{\boldsymbol{t}}_j$ is implemented by using the BLAS routines as follows:

$$\begin{cases} \hat{\boldsymbol{t}}_j := (-t_j) Y_{j-1}^\top \boldsymbol{y}_j + 0 \cdot \hat{\boldsymbol{t}}_j & \texttt{(dgemv)} \\ \hat{\boldsymbol{t}}_j := T_{j-1}\hat{\boldsymbol{t}}_j & \texttt{(dtrmv)} \end{cases}. \tag{3.12}$$

Finally, the computation of line 6 can be reformulated as follows:

$$\begin{aligned} \boldsymbol{q}_j &:= \left(I - Y_j T_j Y_j^\top\right)\boldsymbol{e}_j && (3.13)\\ &= \boldsymbol{e}_j - Y_j T_j Y_j^\top \boldsymbol{e}_j, && (3.14) \end{aligned}$$

where the matrix-vector product $Y_j^\top \boldsymbol{e}_j$ can be simplified by

$$Y_j^\top \boldsymbol{e}_j = \begin{bmatrix} y_{j,1} \\ \vdots \\ y_{j,j} \end{bmatrix}. \tag{3.15}$$

This computation can be performed only by copying the j-th column of Y_j to some vector. Thus, using the BLAS routines, we implement the formula of line 6 as

$$\begin{cases} \boldsymbol{q}_j \coloneqq \boldsymbol{e}_j & \texttt{(dcopy)} \\ \boldsymbol{x}_j \coloneqq \begin{bmatrix} y_{j,1} & \cdots & y_{j,j} \end{bmatrix}^\top & \texttt{(dcopy)} \\ \boldsymbol{x}_j \coloneqq T_j \boldsymbol{x}_j & \texttt{(dtrmv)} \\ \boldsymbol{q}_j \coloneqq (-1) \cdot Y_j \boldsymbol{x}_j + \boldsymbol{q}_j & \texttt{(dgemv)} \end{cases}, \tag{3.16}$$

where $\boldsymbol{x}_j \in \mathbb{R}^j$, $\boldsymbol{q}_j \in \mathbb{R}^n$. We set the initial memory address of $\boldsymbol{x}_j$ and $\boldsymbol{q}_j$ to correspond to that of $\boldsymbol{u}_j$ and $\boldsymbol{v}_j$, respectively.

The computational cost of the compact WY reorthogonalization algorithm shown above is almost $4k^2n + k^3$. In the worst case, i.e., when $k = n$, the computational cost is $5n^3$. Worse yet, for this implementation, we have to use almost $kn + k^2$ memory, since Y_k uses kn and T_k uses k^2.

3.2.2 New Implementation Based on BLAS Routines

Now we discuss the mathematical structure of the compact WY reorthogonalization algorithm and then present a new implementation of it.

We discuss the formula of line 2 before moving on to that of line 3 of Algorithm 3.2. Since

$$c_j \coloneqq -\operatorname{sgn}\left(u_{j,j}\right) \sqrt{\sum_{i=j}^{n} u_{i,j}^2}, \tag{3.17}$$

we have

$$\|\boldsymbol{y}_j\|_2^2 = \left(u_{j,j} - c_j\right)^2 + \sum_{i=j+1}^{n} u_{i,j}^2 \tag{3.18}$$

$$= \sum_{i=j}^{n} u_{i,j}^2 - 2u_{j,j}c_j + c_j^2 \tag{3.19}$$

$$= 2(c_j^2 - u_{j,j}c_j). \tag{3.20}$$

Hence, we have

$$t_j \coloneqq \frac{2}{\|\boldsymbol{y}_j\|_2^2} = \frac{1}{c_j^2 - u_{j,j}c_j}. \tag{3.21}$$

From this fact and the definition of $\boldsymbol{y}_j$ and c_j, we observe that we do not need to compute the leading $(j-1)$ elements of $\boldsymbol{u}_j$ in actual. Restricting the computation to the j-th to the n-th elements of $\boldsymbol{u}_j$, the formula of line 2 is reduced to

$$\hat{\boldsymbol{u}}_j \coloneqq \hat{\boldsymbol{v}}_j - \hat{Y}_{j-1} T_{j-1}^\top Y_{j-1}^\top \boldsymbol{v}_j, \tag{3.22}$$

where $\hat{\boldsymbol{u}}_j \in \mathbb{R}^{n-(j-1)}$ is $\hat{\boldsymbol{u}}_j \coloneqq \begin{bmatrix} u_{j,j} & \cdots & u_{n,j} \end{bmatrix}^\top$.

Now, considering the structure of $\boldsymbol{y}_j$, we can represent $\boldsymbol{y}_j$ $(j = 2, \ldots, k)$ from Eq. (3.5) as the block vector of the form

$$\boldsymbol{y}_j \coloneqq \begin{bmatrix} \boldsymbol{0}_{j-1} \\ \hat{\boldsymbol{y}}_j \end{bmatrix}, \tag{3.23}$$

where $\hat{\boldsymbol{y}}_j \in \mathbb{R}^{n-(j-1)}$ is the vector of non-zero elements of $\boldsymbol{y}_j$. From this fact, Y_j can be represented as the following block matrix:

$$Y_j \coloneqq \begin{bmatrix} L_j \\ \hat{Y}_j \end{bmatrix}, \tag{3.24}$$

where $L_j \in \mathbb{R}^{j \times j}$ is a lower triangular matrix and $\hat{Y}_j \in \mathbb{R}^{(n-j)\times j}$ is generally a dense rectangular matrix. Let us also consider $\boldsymbol{v}_j$ as the block vector of the form

$$\boldsymbol{v}_j \coloneqq \begin{bmatrix} \check{\boldsymbol{v}}_j \\ \hat{\boldsymbol{v}}_j \end{bmatrix}, \tag{3.25}$$

where $\check{\boldsymbol{v}}_j \in \mathbb{R}^{j-1}$ and $\hat{\boldsymbol{v}}_j \in \mathbb{R}^{n-(j-1)}$. Using these block forms of $\boldsymbol{v}_j$ and Y_j, we can reduce the computational cost of the matrix-vector product $Y_{j-1}^\top \boldsymbol{v}_j$ through

$$Y_{j-1}^\top \boldsymbol{v}_j = \begin{bmatrix} L_{j-1} \\ \hat{Y}_{j-1} \end{bmatrix}^\top \begin{bmatrix} \check{\boldsymbol{v}}_j \\ \hat{\boldsymbol{v}}_j \end{bmatrix} = L_{j-1}^\top \check{\boldsymbol{v}}_j + \hat{Y}_{j-1}^\top \hat{\boldsymbol{v}}_j. \tag{3.26}$$

Hence, the formula of $\hat{\boldsymbol{u}}_j$ can be simplified as follows:

$$\hat{\boldsymbol{u}}_j \coloneqq \hat{\boldsymbol{v}}_j - \hat{Y}_{j-1} T_{j-1}^\top \left(L_{j-1}^\top \check{\boldsymbol{v}}_j + \hat{Y}_{j-1}^\top \hat{\boldsymbol{v}}_j \right). \tag{3.27}$$

This formula can then be implemented using the BLAS routines as follows:

$$\begin{cases} \hat{\boldsymbol{u}}_j \coloneqq \hat{\boldsymbol{v}}_j & \texttt{(dcopy)} \\ \check{\boldsymbol{v}}_j \coloneqq L_{j-1}^\top \check{\boldsymbol{v}}_j & \texttt{(dtrmv)} \\ \check{\boldsymbol{v}}_j \coloneqq \hat{Y}_{j-1}^\top \hat{\boldsymbol{v}}_j + \check{\boldsymbol{v}}_j & \texttt{(dgemv)} \\ \check{\boldsymbol{v}}_j \coloneqq T_{j-1}^\top \check{\boldsymbol{v}}_j & \texttt{(dtrmv)} \\ \hat{\boldsymbol{u}}_j \coloneqq (-1) \cdot \hat{Y}_{j-1} \check{\boldsymbol{v}}_j + \hat{\boldsymbol{u}}_j & \texttt{(dgemv)} \end{cases}. \tag{3.28}$$

With the formulation given above, we can now implement the computation of line 3 using the BLAS routines as follows:

$$\begin{cases} y_{i,j} \coloneqq u_{i,j},\ (i = j, \ldots, n) & \texttt{(dcopy)} \\ c_j \coloneqq -\operatorname{sgn}(u_{j,j}) \sqrt{\sum_{i=j}^n u_{i,j}^2}, & \texttt{(dnrm2)} \\ y_{j,j} \coloneqq u_{j,j} - c_j & \\ t_j \coloneqq 1 / \left(c_j^2 - u_{j,j} c_j \right) & \end{cases}. \tag{3.29}$$

For line 5, we can further reduce the computational cost for $\hat{\boldsymbol{t}}_j$ through

$$\hat{\boldsymbol{t}}_j := -t_j T_{j-1} Y_{j-1}^\top \boldsymbol{y}_j \tag{3.30}$$

$$= -t_j T_{j-1} \begin{bmatrix} L_{j-1} \\ \hat{Y}_{j-1} \end{bmatrix}^\top \begin{bmatrix} \boldsymbol{0}_{j-1} \\ \hat{\boldsymbol{y}}_j \end{bmatrix} \tag{3.31}$$

$$= -t_j T_{j-1} \left(L_{j-1}^\top \boldsymbol{0}_{j-1} + \hat{Y}_{j-1}^\top \hat{\boldsymbol{y}}_j \right) \tag{3.32}$$

$$= -t_j T_{j-1} \hat{Y}_{j-1}^\top \hat{\boldsymbol{y}}_j. \tag{3.33}$$

This formula can be implemented using the BLAS routines as follows:

$$\begin{cases} \hat{\boldsymbol{t}}_j := (-t_j) \hat{Y}_{j-1}^\top \hat{\boldsymbol{y}}_j + 0 \cdot \hat{\boldsymbol{t}}_j & \texttt{(dgemv)} \\ \hat{\boldsymbol{t}}_j := T_{j-1} \hat{\boldsymbol{t}}_j & \texttt{(dtrmv)} \end{cases}. \tag{3.34}$$

Finally, we may exploit the fact that the orthogonality of the vectors $\boldsymbol{q}_j$ computed by line 6 with respect to other vectors is preserved regardless of the sign of $\boldsymbol{q}_j$. Thus, we can reformulate $\boldsymbol{q}_j$ as $\boldsymbol{q}_j = \left(Y_j T_j Y_j^\top - I \right) \boldsymbol{e}_j$. Furthermore, let us consider $\boldsymbol{q}_j$ as the following block vector:

$$\boldsymbol{q}_j := \begin{bmatrix} \check{\boldsymbol{q}}_j \\ \hat{\boldsymbol{q}}_j \end{bmatrix}, \tag{3.35}$$

where $\check{\boldsymbol{q}}_j \in \mathbb{R}^j$, $\hat{\boldsymbol{q}}_j \in \mathbb{R}^{n-j}$. With this representation, the equation for $\boldsymbol{q}_j$ can be reformulated as follows:

$$\begin{bmatrix} \check{\boldsymbol{q}}_j \\ \hat{\boldsymbol{q}}_j \end{bmatrix} := \begin{bmatrix} L_j T_j Y_j^\top \boldsymbol{e}_j \\ \hat{Y}_j T_j Y_j^\top \boldsymbol{e}_j \end{bmatrix} - \begin{bmatrix} \check{\boldsymbol{e}}_j \\ \boldsymbol{0}_{n-j} \end{bmatrix}, \tag{3.36}$$

where $\check{\boldsymbol{e}}_j$ is the j-th column vector of the $j \times j$ identity matrix. This formula can be implemented by using the BLAS routines as follows:

$$\begin{cases} \boldsymbol{x}_j := \begin{bmatrix} y_{j,1} & \cdots & y_{j,j} \end{bmatrix}^\top & \texttt{(dcopy)} \\ \boldsymbol{x}_j := T_j \boldsymbol{x}_j & \texttt{(dtrmv)} \\ \check{\boldsymbol{q}}_j := \boldsymbol{x}_j & \texttt{(dcopy)} \\ \check{\boldsymbol{q}}_j := L_j \check{\boldsymbol{q}}_j & \texttt{(dtrmv)} \\ \hat{\boldsymbol{q}}_j := \hat{Y}_j \boldsymbol{x}_j + 0 \cdot \hat{\boldsymbol{q}}_j & \texttt{(dgemv)} \\ q_{j,j} := q_{j,j} - 1 \end{cases}, \tag{3.37}$$

where $\boldsymbol{x}_j \in \mathbb{R}^j$.

As the result, the improved implementation of the compact WY reorthogonalization is shown as Algorithm 3.3. By this implementation, the highest order of computational cost of the compact WY algorithm is reduced from $4k^2n + k^3$ to $4k^2n - k^3$. Even in the worst case, i.e., when $k = n$, the computational cost of the new implementation of the compact WY algorithm is no more than $3n^3$. In addition, the proposed implementation does not have to refer to any zero elements of Y_j and T_j. Thus, the amount of the working memory for the compact WY reorthogonalization can be reduced to almost $n(m + 1)$ by assigning portions of a two-dimensional working array to Y_j and T_j as shown in Figure 3.1.

Algorithm 3.3 Suitable Implementation of Compact WY Reorthogonalization

1: **function** NEW_COMPACTWY($\boldsymbol{v}_j, Y_{j-1}, T_{j-1}$)
2: $\quad \hat{\boldsymbol{u}}_j := \hat{\boldsymbol{v}}_j - \hat{Y}_{j-1} T_{j-1}^\top \left(L_{j-1}^\top \check{\boldsymbol{v}}_j + \hat{Y}_{j-1}^\top \hat{\boldsymbol{v}}_j \right)$
3: $\quad$ Compute $\hat{\boldsymbol{y}}_j$ by using $\hat{\boldsymbol{u}}_j$
4: $\quad t_j := 1/(c_j^2 - u_{j,j} c_j)$
5: $\quad Y_j := \begin{bmatrix} L_{j-1} & \boldsymbol{0}_j \\ \hat{Y}_{j-1} & \hat{\boldsymbol{y}}_j \end{bmatrix}$, $T_j := \begin{bmatrix} T_{j-1} & -t_j T_{j-1} \hat{Y}_{j-1}^\top \hat{\boldsymbol{y}}_j \\ \boldsymbol{0}_{j-1}^\top & t_j \end{bmatrix}$
6: $\quad \boldsymbol{q}_j := \begin{bmatrix} L_j T_j Y_j^\top \boldsymbol{e}_j \\ \hat{Y}_j T_j Y_j^\top \boldsymbol{e}_j \end{bmatrix}$
7: $\quad q_{j,j} := q_{j,j} - 1$
8: $\quad$ **return** $\boldsymbol{q}_j$
9: **end function**

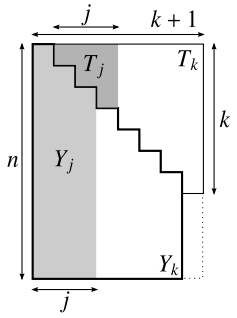


Figure 3.1: Assignment model of Y_j and T_j in the working memory.

Table 3.1: Comparison of the reorthogonalization algorithms [54, 55]

	Computation	*Synchronization*	*Orthogonality*
MGS	$2k^2n$	$O(k^2)$	$O(\epsilon\kappa(V))$
Householder	$4k^2n$	$O(k^2)$	$O(\epsilon)$
compact WY	$4k^2n + k^3$	$O(k)$	$O(\epsilon)$
new compact WY	$4k^2n - k^3$	$O(k)$	$O(\epsilon)$

3.2.3 Comparison of Reorthogonalization Algorithms

The compact WY reorthogonalization has a stable orthogonality originating in the Householder transformations, and its numerical computation is mainly performed by the BLAS 2 routines. As a result, it is better suited to parallel processing than MGS. Table 3.1 shows the differences in performance among the orthogonalization algorithms mentioned above. *Computation* denotes the order of the computational cost, *Synchronization* indicates the order of the number of synchronizations, and *Orthogonality* indicates the norm $\|Q^\top Q - I\|$, where Q is defined as an $n \times k$ matrix $Q = [\boldsymbol{q}_1 \ \cdots \ \boldsymbol{q}_k]$ and $\boldsymbol{q}_j$ ($j = 1, \ldots, k$) is the orthogonalized vector of $\boldsymbol{v}_j$ by each algorithm. Note that ϵ denotes the machine epsilon and $\kappa(V)$ denotes the condition number of $V = [\boldsymbol{v}_1 \ \cdots \ \boldsymbol{v}_k]$.

Algorithm 3.4 Inverse Iteration Algorithm with Compact WY Reorthogonalization

1: **function** TRIINVCWY($T,\ \ell,\ \tilde{\lambda}_1,\ \ldots,\ \tilde{\lambda}_\ell$)
2: **do** $k := 1$ to ℓ
3: $i := 0$
4: Generate an initial random vector $\boldsymbol{v}_k^{(0)}$
5: $T - \tilde{\lambda}_k := P_k L_k U_k$ ▷ Call `dlagtf`
6: **repeat**
7: $i := i + 1$
8: Normalize $\boldsymbol{v}_k^{(i-1)}$ to $\boldsymbol{q}_k^{(i-1)}$
9: Solve $P_k L_k U_k \boldsymbol{v}_k^{(i)} = \boldsymbol{q}_k^{(i-1)}$ ▷ Call `dlagts`
10: **if** $k > 1$ and $|\tilde{\lambda}_{k-1} - \tilde{\lambda}_k| \leq 10^{-3}\|T\|_1$, **then**
11: $j_c := j - k_1$
12: **if** $j_c := 1$ and $k := 1$, **then**
13: Compute $Y_1 := [\boldsymbol{y}_1]$ and $T_1 := [t_1]$ by using $\boldsymbol{v}_{k_1}$
14: **end if**
15: $\boldsymbol{u}_{j_c+1} := \left(I - Y_{j_c} T_{j_c}^\top Y_{j_c}^\top\right) \boldsymbol{v}_j^{(k)}$
16: Compute $\boldsymbol{y}_{j_c+1}$ and t_{j_c+1} by using $\boldsymbol{u}_{j_c+1}$
17: $Y_{j_c+1} := \begin{bmatrix} Y_{j_c} & \boldsymbol{y}_{j_c+1} \end{bmatrix}, T_{j_c+1} := \begin{bmatrix} T_{j_c} & -t_{j_c+1} T_{j_c} Y_{j_c}^\top \boldsymbol{y}_{j_c+1} \\ \boldsymbol{0}_{j_c}^\top & t_{j_c+1} \end{bmatrix}$
18: $\boldsymbol{v}_j^{(k)} := \left(I - Y_{j_c+1} T_{j_c+1} Y_{j_c+1}^\top\right) \boldsymbol{e}_{j_c+1}$
19: **else**
20: $k_1 := k$
21: **end if**
22: **until** some condition is met.
23: Normalize $\boldsymbol{v}_k^{(i)}$ to $\boldsymbol{q}_k^{(i)}$
24: $Q_k := [Q_{k-1},\ \boldsymbol{q}_k^{(i)}]$
25: **end do**
26: **return** $Q_\ell = [\boldsymbol{q}_1,\ \ldots,\ \boldsymbol{q}_\ell]$
27: **end function**

3.3 Inverse Iteration Algorithm with Compact WY Reorthogonalization

Algorithm 3.4 shows a pseudocode of the inverse iteration algorithm with compact WY reorthogonalization, where ℓ is the number of the desired eigenpairs and k_1 denotes the index of the largest eigenvalue of a certain cluster. It is noted that Algorithm 3.4 is based on the `dstein` routine provided in LAPACK as well as Algorithm 2.3.

The implementation of the compact WY reorthogonalization in Algorithm 3.4 corresponds to that described in Section 3.2.1. The application of the proposed implementation in Algorithm 3.3 is done by replacing lines 15-18 in Algorithm 3.4 with Algorithm 3.3.

Table 3.2: Specifications of The Experimental Environments

	Environment 1	Environment 2	Environment 3
CPU	AMD Opteron 6128@2.0GHz	Intel Xeon X5570@2.93GHz	
	32 cores (8 cores×4)	8 cores (4 cores×2)	
RAM	DDR3-1066 256GB	DDR3-1066 32GB	
Compiler	Gfortran 4.4.5	Gfortran 4.4.5	Intel Fortran 13.0.1
Software	LAPACK 3.3.0	LAPACK 3.3.0	Intel MKL 11.0.1
	GotoBLAS2 1.13	GotoBLAS2 1.13	

3.4 Performance Evaluation

This section gives the results of the numerical experiments performed to evaluate the performance of the inverse iteration algorithm with the compact WY reorthogonalization described in Section 3.3.

In the experiments, the performance of three codes **DSTEIN**, **DSTEIN-cWY**, and **DSTEIN-ncWY** is evaluated through the computation of eigenvectors of real symmetric tridiagonal matrices. Here, **DSTEIN** is the `dstein` routine provided in LAPACK, which is a code of the inverse iteration algorithm with the MGS algorithm shown in Algorithm 2.3. **DSTEIN-cWY** and **DSTEIN-ncWY** are codes of the inverse iteration algorithm with the compact WY reorthogonalization shown in Algorithm 3.4. Both codes are also implemented on the basis of the `dstein` routine. The naive implementation of the compact WY reorthogonalization described in Section 3.2.1 is applied to the **DSTEIN-cWY** code. The proposed implementation of the compact WY reorthogonalization in Section 3.2.2, i.e., Algorithm 3.3, is applied to the **DSTEIN-ncWY** code.

Table 3.2 shows the specifications of all three experimental environments. Note that Environments 2 and 3 comprise the same computer but with different compiler and software. As mentioned in Section 2.2.2, the inverse iteration codes described above are parallelized in terms of the BLAS routines by which all CPU cores in each Environment are used for parallel executions. For running the parallel BLAS routines, GotoBLAS2 [10] is employed in Environments 1 and 2, and the Intel Math Kernel Library [9] is employed in Environment 3.

In the experiments, $n \times n$ symmetric tridiagonal matrices T_1, T_2, and T_3 were used for target matrices. T_1 is the glued-Wilkinson matrix [55], which is defined as

$$T_1 := \left[\begin{array}{c|c|c|c} W_{21}^{\dagger} & \delta & & \\ \hline \delta & W_{21}^{\dagger} & \delta & \\ \hline & \delta & \ddots \quad \ddots & \\ & & \ddots \quad \ddots & \delta \\ \hline & & \delta & W_{21}^{\dagger} \end{array}\right], \tag{3.38}$$

where each block diagonal element $W_{21}^{\dagger}$ is the 21×21 Wilkinson matrix such that

$$W_{21}^{\dagger} = \begin{bmatrix} 10 & 1 & & & & & \\ 1 & 9 & 1 & & & & \\ & 1 & \ddots & \ddots & & & \\ & & \ddots & 0 & \ddots & & \\ & & & \ddots & \ddots & 1 & \\ & & & & 1 & 10 \end{bmatrix}, \tag{3.39}$$

and the real scalar parameter δ such that $0 < \delta < 1$ is to determine the closeness of eigenvalues in each cluster. In the experiments of this thesis, we set $\delta = 10^{-4}$. The eigenvalues of T_1 are divided into 14 clusters of eigenvalues in terms of Peters-Wilkinson's criterion. The size of seven of these clusters is $n/21$ and the size of the remaining clusters is $2n/21$. Note that the distance between the minimum and maximum eigenvalues of each cluster becomes smaller as δ becomes smaller. The computation of eigenpairs of the glued-Wilkinson matrix T_1 is one of the benchmark problems for evaluating the robustness of real symmetric tridiagonal eigensolvers since the eigenvalues belonging in each cluster are extremely close to one another. In fact, the glued-Wilkinson matrix has also been used in the previous studies [25,26,56]. T_2 is a symmetric tridiagonal matrix, all the elements of which are set to 1. All the eigenvalues of this matrix belong to only one cluster except for the case of $n = 1,050$ in the experiments of this section. Thus, the computation of eigenpairs of T_2 is the worst case for the inverse iteration algorithms from the viewpoint of the computational cost. T_3 is a symmetric tridiagonal matrix, all the elements of which are set to uniform random numbers in the range $(0, 1)$. The eigenvalue distribution of random symmetric matrices such as T_3 is known to be similar to that of the matrices appearing in many applications. Almost all the eigenvalues of T_3 belongs to a significantly large cluster in terms of Peters-Wilkinson's criterion [53] if the size of T_3 is sufficiently large [18].

Note that the maximum number of the **repeat**-**until** iterations in Algorithms 2.3 and 3.4 is commonly set to 5 as done in the `dstein` routine, but in all the experiments with all codes the required number of the iterations was 3 to satisfy the termination condition given in the `dstein`, which we applied to other codes as well. Also note that the input of the inverse iteration, i.e., the set of candidate eigenvalues $\tilde{\lambda}_1, \ldots, \tilde{\lambda}_n$ are computed by the `dstebz` routine of LAPACK.

3.4.1 Execution Time

Figures 3.2, 3.3, and 3.4 show the execution time by using each code of the inverse iteration algorithms for computing all the eigenvectors of T_1, T_2, and T_3, respectively.

Figures 3.2 and 3.3 show that both **DSTEIN-ncWY** and **DSTEIN-cWY** are faster than **DSTEIN** for all cases except that for T_3 of $n = 1,050$ in Environments 1 and 2. On the other hand, Figure 3.4 shows that **DSTEIN-ncWY** is faster than **DSTEIN** for all matrix types and **DSTEIN-cWY** is faster than **DSTEIN** for T_1 and T_3 matrices as n becomes larger in Environment 3. From these results, the superiority of the compact

WY reorthogonalization over the MGS algorithm in terms of parallel performance is confirmed.

DSTEIN-ncWY is faster than **DSTEIN-cWY** as n becomes larger for all the target matrices. As can be seen in Figures 3.2b and 3.3b, in particular, **DSTEIN-ncWY** is about 1.6 times faster than **DSTEIN-cWY** (using GotoBLAS2) in the cases of T_2, the size of the eigenvalue cluster of which is equivalent to n. These results are in accordance with the difference between the computational cost of the naive implementation of the compact WY reorthogonalization and that of the proposed implementation of the compact WY reorthogonalization. Thus, the reduction of the computational cost of the compact WY reorthogonalization effectively accelerates the reorthogonalization process of the inverse iteration algorithm. However, **DSTEIN-cWY** is faster than **DSTEIN-ncWY** in some cases when n is small. This can result from bottlenecks in synchronization. Since **DSTEIN-ncWY** is composed of more BLAS routines than **DSTEIN-cWY**, the execution time of **DSTEIN-ncWY** is more delayed by synchronization in such cases than the other.

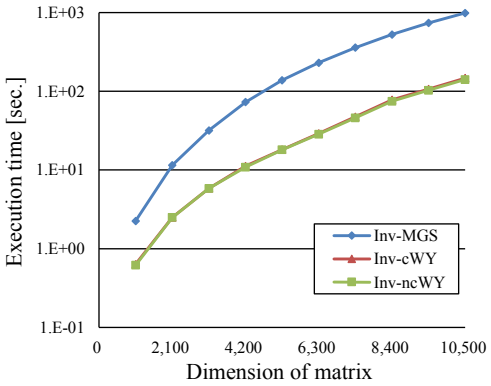
It should be noted that **DSTEIN** in Environment 3 is faster than that in Environment 2, as shown in Figures 3.3 and 3.4. In other words, on the processors equipped in Environments 2 and 3, the combination of Intel compiler and Intel MKL is able to get better performance of **DSTEIN** than that of GNU compiler and GotoBLAS2. In addition, **DSTEIN** runs on Environment 1 much slower than Environments 2 and 3, though its peak performance is higher than the Environments 2 and 3 owing to the larger number of cores. On the other hand, **DSTEIN-cWY** and **DSTEIN-ncWY** runs on Environment 1 faster than Environments 2 and 3 reflecting the superiority of peak performance. This observation implies that the superiority of the compact WY reorthogonalization over the MGS will be more significant as the semiconductor technology progresses to increase the number of cores of a microprocessor.

3.4.2 Orthogonality

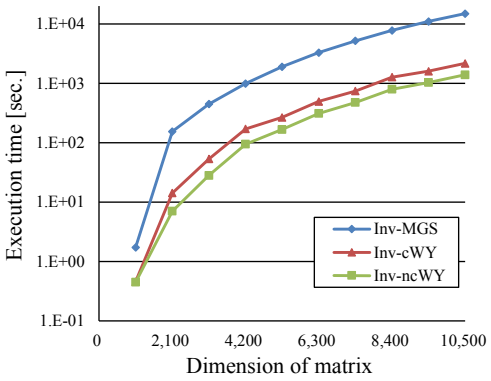
Figure 3.5 shows the orthogonality $\|Q^\top Q - I\|_\infty / n$ using each code of the inverse iteration algorithms in Environment 1, where $Q = [\boldsymbol{q}_1 \ \cdots \ \boldsymbol{q}_n]$ and $\boldsymbol{q}_j$ $(j = 1, \ldots, n)$ is an eigenvector computed by each code.

We observe from Figure 3.5 that the orthogonality of eigenvectors computed by **DSTEIN-ncWY** is better than those by **DSTEIN-cWY**. However, contrary to theoretical analysis, the orthogonality of the eigenvectors computed by **DSTEIN** is the best. Since the reorthogonalization in the inverse iteration algorithm is performed on each inverse iteration, the condition number of vectors to be orthogonalized may change on each inverse iteration and reorthogonalization. In addition, the number of arithmetic operations in the compact WY reorthogonalization is larger than that of the MGS algorithm as shown in Table 3.1. Since the computation errors increase as the number of operations becomes larger, the compact WY reorthogonalization is more affected by the computation errors than the MGS algorithm.

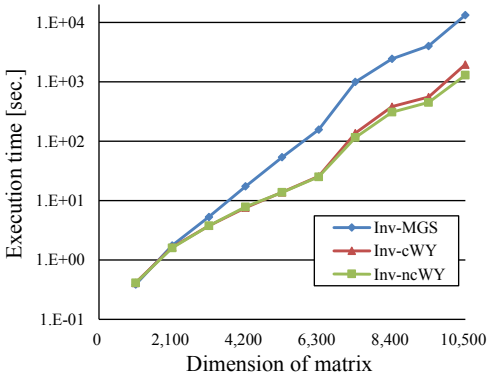
All these results show the orthogonality of the eigenvectors computed by the inverse iteration algorithm computes with the compact WY reorthogonalization is the same level as that by the conventional inverse iteration algorithm even in the cases of T_1, the



(a) Cases of T_1

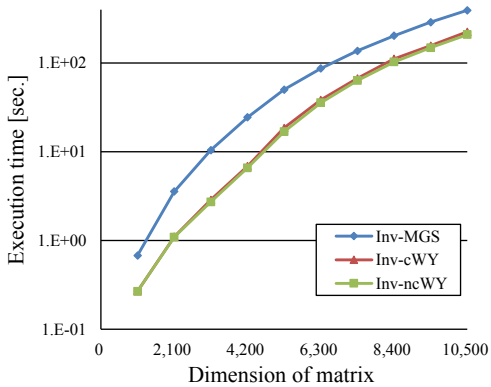


(b) Cases of T_2

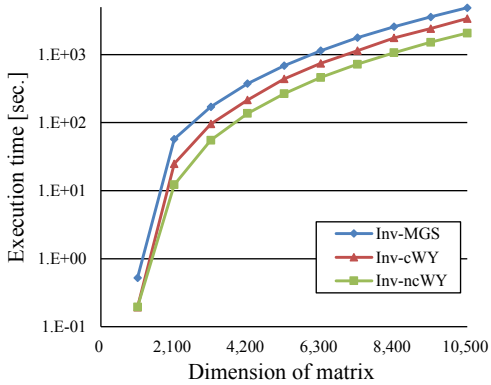


(c) Cases of T_3

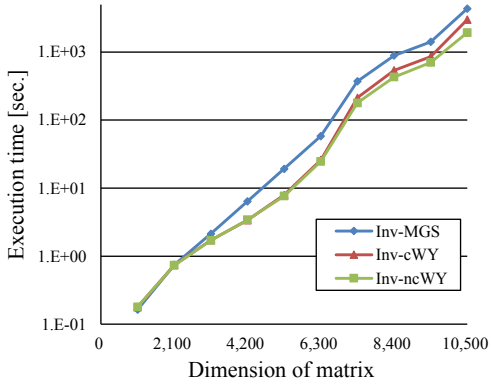
Figure 3.2: Execution times for computing the eigenvectors of target matrices by using each code in Environment 1.



(a) Cases of T_1



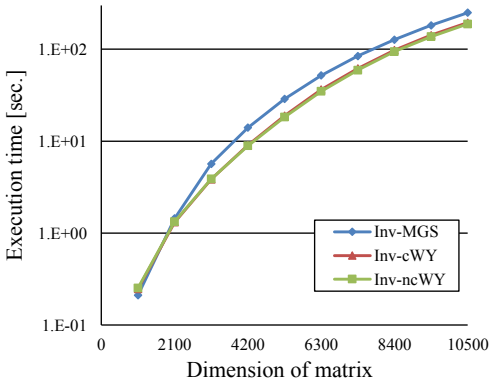
(b) Cases of T_2



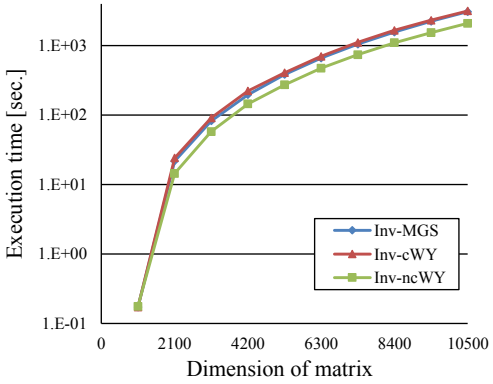
(c) Cases of T_3

Figure 3.3: Execution times for computing the eigenvectors of target matrices by using each code in Environment 2.

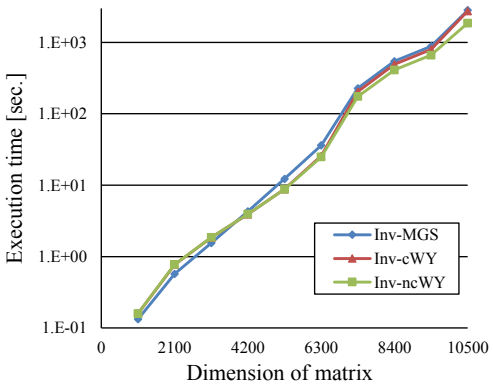
glued-Wilkinson matrix.



(a) Cases of T_1

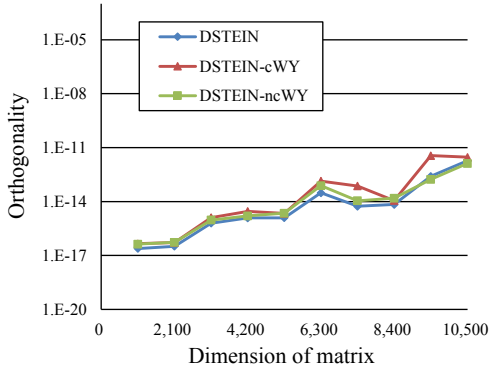


(b) Cases of T_2

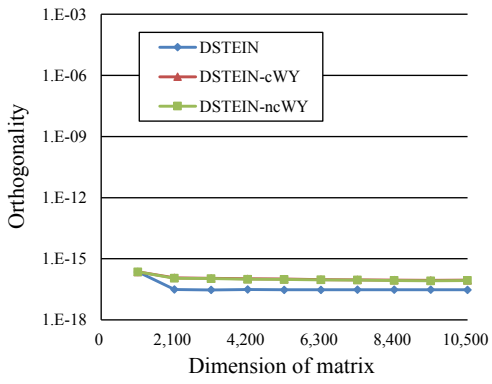


(c) Cases of T_3

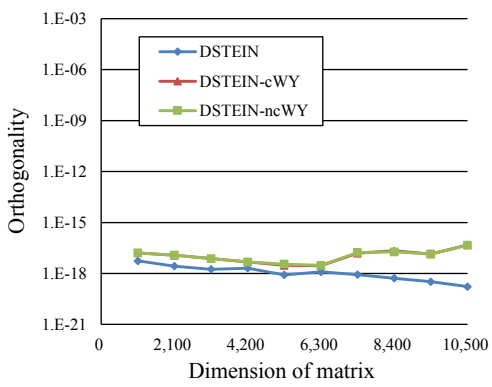
Figure 3.4: Execution times for computing the eigenvectors of target matrices by using each code in Environment 3.



(a) Cases of T_1



(b) Cases of T_2



(c) Cases of T_3

Figure 3.5: Orthogonality $\|Q^\top Q - I\|_\infty / n$ of the computed eigenvectors of target matrices by using each code in Environment 1.

4 Block Inverse Iteration Algorithm with Reorthogonalization

In this chapter, the *block inverse iteration algorithm with reorthogonalization (BIR algorithm)* is proposed for accelerating the subset computation of eigenvectors of T in parallel processing. The BIR algorithm is introduced as an improvement of the *simultaneous inverse iteration (or block inverse iteration) algorithm* [46], which is a variant of the inverse iteration algorithm, with a block parameter. Owing to the introduction of the block parameter, the BIR algorithm requires the *block reorthogonalization* process of computing eigenvectors, which is mainly composed of the BLAS 3 routines such as the matrix multiplications. From this fact and the discussion on the parallel BLAS routines in Chapter 1, the BIR algorithm is expected to be more scalable than the inverse iteration algorithms, as shown in Sections 2.1.2 and 3.3. In addition, the BIR algorithm is expected to accelerate efficiently the computation of eigenvectors in parallel processing.

The rest of this chapter is organized as follows: Section 4.1 gives an introduction of the simultaneous inverse iteration algorithm. Section 4.2 presents the BIR algorithm and then discusses its convergence. Section 4.3 provides experimental results on shared-memory multi-core processors to evaluate the performance of the BIR algorithm. In addition, experimental results are also provided to evaluate the performance of a parallel solver for computing eigenpairs of T, which includes the parallel bisection algorithm presented in Section 2.1.2 and the BIR algorithm.

4.1 Simultaneous Inverse Iteration Algorithm for A Real Symmetric Tridiagonal Matrix T

The simultaneous inverse iteration (SI) algorithm [46] is a variant of the inverse iteration algorithm. In [46], the SI algorithm is defined for multiple eigenvalues of a matrix. In this thesis, we define the simultaneous inverse iteration algorithm for clustered eigenvalues of a real $n \times n$ symmetric tridiagonal matrix T. Algorithm 4.1 shows the pseudocode of the SI algorithm, where $\hat{\ell}$ is the number of eigenvalues belonging to a certain cluster.

Different from the inverse iteration algorithm, the SI algorithm computes simultaneously all the desired eigenvectors on each iteration. For this purpose, we solve $\hat{\ell}$ linear equations simultaneously in lines 11-12 and perform the QR factorization in line 14 instead of the reorthogonalization in the inverse iteration algorithm. On the other hand, as well as the inverse iteration algorithm, we perform the PLU factorization in line 6 for $k = 1, \ldots, \hat{\ell}$ before performing the iterative computations. The resulting elements of these computations must be stored in memory and the amount of them for the SI algorithm is $\hat{\ell}$ times as large as that for the inverse iteration algorithm. It should be noted

Algorithm 4.1 Simultaneous Inverse Iteration Algorithm

```
1:  function TriSI(T, ℓ̂, λ̃_1, ..., λ̃_ℓ̂)
2:      i := 0
3:      Set Q^(0) := [q_1^(0) ... q_ℓ̂^(0)]
4:      !$omp parallel do
5:      do k := 1 to ℓ̂
6:          T − λ̃_k I := P_k L_k U_k
7:      end do
8:      repeat
9:          i := i + 1
10:         !$omp parallel do
11:         do k := 1, ..., ℓ̂
12:             Solve P_k L_k U_k v_k^(i) = q_k^(i−1)
13:         end do
14:         QR factorization: V^(i) = Q^(i) R^(i) (V^(i) := [v_1^(i) ... v_ℓ̂^(i)])
15:     until converge
16:     return Q_ℓ̂ := [Q^(i)]
17: end function
```

that the computational cost of the SI algorithm is almost the same as that of the inverse iteration.

The parallelization of the SI algorithm is as follows: Since the computations for solving the linear equations (lines 5-6 and lines 11-12) are independent in terms of k from each other, these computations can be parallelized without any data synchronization. Thus, these computations are parallelized by using the OpenMP directives as shown in lines 4 and 10. On the other hand, the QR factorization of line 14 is parallelized in terms of the BLAS routines, such as the matrix multiplications.

The recursive implementation of the *classical Gram-Schmidt (CGS)* algorithm [11] is proposed for the QR factorization, which is mainly composed of the matrix multiplications, and is shown to achieve the highly scalable performance [57]. In the followings, this implementation of the CGS algorithm is referred to as the *RCGS algorithm*. However, the orthogonality of the matrix computed by the QR factorization using the CGS algorithm sometimes deteriorates if the condition number of the original matrix is large. To improve the orthogonality, the *Classical Gram-Schmidt algorithm with reorthogonalization (CGS2 algorithm)*, which repeats the CGS algorithm twice, is proposed [47]. Similarly, the RCGS algorithm had better be repeated twice for improving the orthogonality of the resulting matrix and we refer to this implementation as the *RCGS2 algorithm*. In this thesis, we adopt the RCGS2 algorithm to the QR factorization of line 14 and we parallelize the RCGS2 algorithm by using the parallel implementation of `dgemm`, the matrix multiplication routine in the BLAS.

4.2 Block Inverse Iteration Algorithm with Reorthogonalization

Now we propose the block inverse iteration algorithm with reorthogonalization (BIR algorithm). As well as the SI algorithm, we define the BIR algorithm for computing the eigenvectors $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_{\hat{\ell}}$ corresponding to $\lambda_1, \ldots, \lambda_{\hat{\ell}}$, which belong to a cluster of eigenvalues of T. In the followings, let $Q_{\hat{\ell}}$ be $Q_{\hat{\ell}} = \begin{bmatrix} \boldsymbol{q}_1 & \cdots & \boldsymbol{q}_{\hat{\ell}} \end{bmatrix}$.

The BIR algorithm is introduced as an improvement of the SI algorithm with a block parameter $r \in \mathbb{Z}$ $(1 \leq r \leq \hat{\ell})$. Note that r is arbitrarily determined by users and, for convenience, let r define $r = \hat{\ell}/l$ where $l \in \mathbb{Z}$ satisfy $1 \leq r \leq \hat{\ell}$.

The basic idea of the BIR algorithm is to apply the SI algorithm to each of computing $Q_{j,r}$, which is the j-th block matrix of eigenvectors for T and $Q_{j,r} = \begin{bmatrix} \boldsymbol{q}_{(j-1)r+1} & \cdots & \boldsymbol{q}_{jr} \end{bmatrix}$ $(j = 1, \ldots, \hat{\ell}/r)$. Now let us consider the computation of $Q_{j,r}^{(i)} = \begin{bmatrix} \boldsymbol{q}_{(j-1)r+1}^{(i)} & \cdots & \boldsymbol{q}_{jr}^{(i)} \end{bmatrix}$ using the SI algorithm for $i = 1, 2, \ldots$. In this computation, it is not guaranteed that the vectors of $Q_{j,r}^{(i)}$ are orthogonal against the computed eigenvectors $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_{(j-1)r}$. To solve the problem, we introduce the BIR algorithm to the reorthogonalization of $Q_{j,r}^{(i)}$ against the computed eigenvectors $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_{(j-1)r}$ before the QR factorization is performed. Summarizing the discussions above, the i-th iteration of the BIR algorithm for computing $Q_{j,r}$ is composed of the following three computation and $Q_{j,r}^{(i)}$ converges to $Q_{j,r}$ as i increases:

1. Compute $\boldsymbol{v}_{(j-1)r+k}^{(i)}$ from $\boldsymbol{q}_{(j-1)r+k}^{(i-1)}$ $(k = 1, \ldots, r)$:

$$\left(T - \tilde{\lambda}_k I\right) \boldsymbol{v}_{(j-1)r+k}^{(i)} = \boldsymbol{q}_{(j-1)r+k}^{(i-1)}. \tag{4.1}$$

2. Reorthogonalize $V_{j,r}^{(i)} = \begin{bmatrix} \boldsymbol{v}_{(j-1)r+1}^{(i)} & \cdots & \boldsymbol{v}_{jr}^{(i)} \end{bmatrix}$ against $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_{(j-1)r}$

3. Perform the QR factorization:

$$\overline{V}_{j,r}^{(i)} = Q_{j,r}^{(i)} R_{j,r}^{(i)}, \tag{4.2}$$

   where $\overline{V}_{j,r}^{(i)}$ is the resulting matrix of 2).

As well as the SI algorithm, $Q_{j,r}^{(0)} = \begin{bmatrix} \boldsymbol{q}_{(j-1)r+1}^{(0)} & \cdots & \boldsymbol{q}_{jr}^{(0)} \end{bmatrix}$ is set an $n \times r$ real orthonormal matrix whose elements are generated from random numbers. Since the BIR algorithm solves r linear equations simultaneously as shown in 1), the amount of memory usage for this computation needs r times as large as the inverse iteration algorithm. If $r \ll \hat{\ell}$, however, the amount of memory usage is significantly less than that for the SI algorithm. Now the combination of 2) and 3) is referred to as the *block reorthogonalization* process. For the purpose of the block reorthogonalization, the *block CGS (BCGS) algorithm* [58], which is also a variant of the CGS algorithm, is proposed. To improve the orthogonality of the resulting vectors, the *BCGS algorithm with reorthogonalization (BCGS2 algorithm)* [59] is preferable. The BCGS2 algorithm including the QR factorization can be implemented using mainly the matrix multiplications.

Algorithm 4.2 Block Inverse Iteration Algorithm with Reorthogonalization for A Real Symmetric Tridiagonal Matrix T

```
1:  function TriBIR(T, r, ℓ̂, λ̃_1, ..., λ̃_ℓ̂)
2:      Set an n × r matrix Q_0 be Q_0 := O
3:      do j := 1 to ℓ̂/r
4:          i := 0
5:          Generate Q^(0)_{j,r} := [q^(0)_{(j-1)r+1}  ···  q^(0)_{jr}]
6:          !$omp parallel do
7:          do k := 1 to r
8:              T − λ̃_{jr+k} := P_k L_k U_k
9:          end do
10:         repeat
11:             i := i + 1
12:             !$omp parallel do
13:             do k := 1 to r
14:                 Solve P_k L_k U_k v^(i)_k := q^(i-1)_{(j-1)r+k}
15:             end do
16:             V^(i)_{j,r} := V^(i)_{j,r} − Q_{(j-1)r} Q^⊤_{(j-1)r} V_{j,r}
17:             QR factorization: V^(i)_{j,r} = Q̄^(i)_{j,r} R^(i)_{j,r}
18:             Q̄^(i)_{j,r} := Q̄^(i)_{j,r} − Q_{(j-1)r} Q^⊤_{(j-1)r} Q̄^(i)_{j,r}
19:             QR factorization: Q̄^(i)_{j,r} = Q^(i)_{j,r} R^(i)_{j,r}
20:         until converge
21:         Q_{jr} := [Q_{(j-1)r}  Q^(i)_{j,r}]  (Q_r := [Q^(i)_{1,r}])
22:     end do
23:     return Q_ℓ̂ = [q_1  ···  q_ℓ̂]
24: end function
```

Algorithm 4.2 shows the pseudocode of the BIR algorithm. Lines 8 and 14 corresponds to 1). In addition, lines 16-19 shows the process of the BCGS2 algorithm. In this thesis, the QR factorization in lines 17 and 19 is implemented using the RCGS algorithm described in Section 4.1. Note that the total computational cost of the BIR algorithm is the same order as both that of the SI algorithm and that of the inverse iteration algorithm.

The parallelization of the BIR algorithm is as follows: As well as the SI algorithm, the r linear equations are solved in parallel by using OpenMP directives as shown on lines 6 and 12. Since almost all the computations of the BCGS2 algorithm in lines 16-19 are composed of the matrix multiplications, we parallelize the BCGS2 algorithm by using the parallel implementation of `dgemm`. Thus, the BIR algorithm is expected to achieve the highly scalable performance even on processors of high core-counts.

Note that the BIR algorithm is equivalent to the inverse iteration algorithm with reorthogonalization when $r = 1$, and is equivalent to the simultaneously inverse iteration algorithm when $r = \hat{\ell}$.

Table 4.1: Specifications of the experimental environment: one node of Appro Green Blade 8000 at ACCMS, Kyoto University.

CPU	Intel Xeon E5-2670@2.6GHz, 16 cores (8 cores × 2)
	L3 cache: 20MB × 2
RAM	DDR3-1600 64GB, 136.4GB/sec
Compiler	Intel C++/Fortran Compiler 15.0.2
Options	`-O3 -xHOST -ipo -no-prec-div`
	`-openmp -mcmodel=medium -shared-intel`
Software	Intel Math Kernel Library 11.2.2

4.2.1 Convergence of BIR Algorithm

The convergence of the BIR algorithm can be derived from that of the SI algorithm as follows: Section 5.9.2 in [46] shows the convergence of the SI algorithm is guaranteed when we compute all the eigenvectors corresponding to the multiple eigenvalues. Since the computation of the corresponding eigenvectors to the multiple eigenvalues is the most difficult case for the SI algorithm, the convergence of the SI algorithm as defined in this thesis is naturally guaranteed. In addition, although the reorthogonalization of the vectors against the computed eigenvectors is introduced into the BIR algorithm for improving the orthogonality of the resulting vectors, the reorthogonalization process never disturb the convergence to the eigenvectors. Thus, the convergence of the BIR algorithm is also guaranteed.

Note that the SI algorithm is pointed out in Corollary 5.9.2 in [46] to fail if eigenvalues of the target matrix are not computed accurately. Thus, the BIR algorithm is not always guaranteed to be robust in such a case.

4.3 Performance Evaluation

This section gives the results of the numerical experiments performed to evaluate the performance of the proposed eigensolver, which computes eigenvalues of target matrices using the parallel bisection algorithm given in Section 2.1.2, and computes the corresponding eigenvectors using the BIR algorithm described in Section 4.2.

All the experiments were performed with 16 threads on one node of Appro Green Blade 8000 at ACCMS, Kyoto University, whose specification is listed in Table 4.1. The Intel Math Kernel Library (MKL) is used for the parallel execution of the BLAS and LAPACK routines and the OpenMP directives are also used for the parallelization as shown in Algorithms 2.2, 4.1, and 4.2.

$n \times n$ symmetric tridiagonal matrices T_1, T_2, and T_3 were used as target matrices in the experiments. T_1 is the glued-Wilkinson matrix [25, 26]. T_2 is a symmetric tridiagonal matrix, all the elements of which are set to 1. T_3 is a symmetric tridiagonal matrix, into which we tridiagonalize a dense symmetric matrix, whose elements are uniform random numbers in the range $(0, 1)$, by using the `dsytrd` routine provided in LAPACK. All of these target matrices was selected by the reasons discussed in Section 3.4.

The maximum number of iterations to make $Q_{j,r}^{(i)}$ in the BIR algorithm or $Q^{(i)}$ in the SI algorithm convergent, i.e., the **repeat**-**until** iterations in Algorithms 4.1 and 4.2, was set to 5 as well as the experiments shown in Section 3.4. In fact, the required number of the iterations was 3 in all the experiments of this section.

It should be noted that the performance of the BIR algorithm has been evaluated through numerical experiments on shared-memory multi-core processors in [A2], besides the experiments shown in this section. This study has shown that the BIR algorithm is faster than two conventional inverse iteration algorithms if the size of target matrices are large. One of the conventional inverse iteration algorithm has implemented the MGS algorithm to its reorthogonalization process. The other has implemented the CGS2 algorithm, which is composed of the BLAS 2 routines such as the matrix-vector multiplications as well as the compact WY reorthogonalization discussed in Chapter 3. In addition, the BIR algorithm has been also shown to achieve as high accuracy as the conventional inverse iteration algorithms.

4.3.1 Experiments I

We compared the execution time spent to compute all the eigenvectors of symmetric tridiagonal matrices using the BIR algorithm with different size of r and the SI algorithm, which corresponds to the BIR algorithm with $r = \hat{\ell}$ as mentioned in Section 4.2. Note that the eigenvalues are computed by the parallel bisection algorithm given in Section 2.1.2 and all the numerical experiments are run with 16 threads.

Figures 4.1 shows the execution time using the BIR algorithm with different size of r and the SI algorithm. Figure 4.1a, 4.1b, and 4.1c corresponds to the cases of T_1, T_2, and T_3, respectively. It should be noted that the SI algorithm for $n = 42,000$ of T_1 and $n = 40,000$ of T_2 and T_3 fails because of the memory overflow.

As can be seen in Figures 4.1, the BIR algorithm with relatively small r such as 16, 32, and 64 is slower than the other cases. It is because that the parallel matrix multiplication routine achieves the higher performance as the target matrix is larger. In addition, the BIR algorithm with $r = 2,048$ is not always the fastest case and the SI algorithm, which corresponds to the BIR algorithm with $r = \hat{\ell}$, is slower than some cases of the BIR algorithm. From these facts, it is not always guaranteed that the BIR algorithm becomes faster as the block parameter r is larger. It is because the best r, with which the BIR algorithm achieves the fastest performance, depends on the ratio of the computational cost of the reorthogonalization process to the total computational cost of the BIR algorithm. In fact, the best r is 128 or 256 for T_1 and $1,024$ or $2,048$ for T_2 and T_3.

In the following experiments, we set $r = 1,024$ as the block parameter of the BIR algorithm because the BIR ($r = 1,024$) averagely achieves the good performance for any matrix in Figures 4.1.

4.3.2 Experiments II

We evaluate speedup ratios of the proposed eigensolver, which is composed of the parallel bisection algorithm and the BIR algorithm. Results of the numerical experiments are

Table 4.2: Execution times and Speedup ratios in the proposed eigensolver for computing all the eigenpairs of $n \times n$ target matrices on the shared-memory 16-core computer shown in Table 4.1].

(a) Cases of T_1

	$n = 21,000$			$n = 42,000$		
	PBi	BIR	Total	PBi	BIR	Total
Serial [sec.]	1.78E+1	6.84E+2	7.02E+2	6.45E+1	4.60E+3	4.67E+3
Parallel [sec.]	1.13E+0	1.24E+2	1.25E+2	4.10E+0	5.98E+2	6.02E+2
Speedup [times]	15.7	5.5	5.6	15.7	7.7	7.8

(b) Cases of T_2

	$n = 20,000$			$n = 40,000$		
	PBi	BIR	Total	PBi	BIR	Total
Serial [sec.]	8.61E+1	5.18E+3	5.27E+3	3.35E+2	4.07E+4	4.11E+4
Parallel [sec.]	5.42E+0	4.02E+2	4.07E+2	2.11E+1	2.88E+3	2.90E+3
Speedup [times]	15.9	12.9	12.9	15.9	14.1	14.2

(c) Cases of T_3

	$n = 20,000$			$n = 40,000$		
	PBi	BIR	Total	PBi	BIR	Total
Serial [sec.]	8.75E+1	5.17E+3	5.26E+3	3.42E+2	4.08E+4	4.11E+4
Parallel [sec.]	5.51E+0	4.01E+2	4.07E+2	2.15E+1	2.88E+3	2.90E+3
Speedup [times]	15.9	12.9	12.9	15.9	14.1	14.2

shown in Table 4.2. In Table 4.2, "PBi" and "BIR" denotes a part of the parallel bisection algorithm and the BIR algorithm, respectively. Then "Total" denotes the whole of computing all the eigenpairs of the target matrix. "Serial" shows the execution time spent for each part using 1 thread and "Parallel" shows that using 16 threads.

The speedup ratios of the parallel bisection nearly equal to 16 times for all the case. However, the speedup ratios of the proposed eigensolver nearly equal to that of the BIR algorithm since the execution time for the BIR is significantly larger than that for the parallel bisection algorithm. Since the BIR algorithm is mainly composed of the matrix multiplications, the good speedup ratios are achieved in the cases of T_2 and T_3 with $n = 40,000$ as can be seen in Table 4.2.

For each type of the target matrices, the speedup ratios of the BIR algorithm become better as n is larger. In addition, the speedup ratios of the BIR algorithm for T_2 and T_3 are better than that for T_1. As well as the discussion in 4.3.1, these phenomena are caused by the ratio of the computational cost spent for the reorthogonalization process to the whole computational cost of the BIR algorithm.

4.3.3 Experiments III

We compare the performance of the proposed eigensolver, `PBi+BIR`, with that of the following three eigensolvers: the MRRR algorithm, the QR algorithm, and the DC

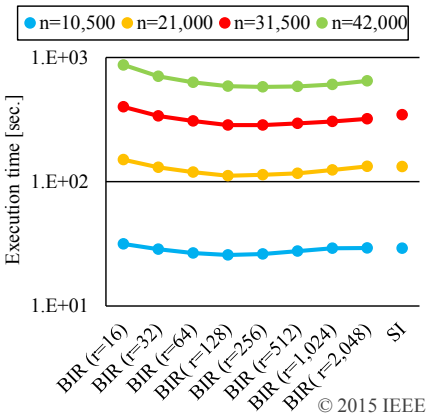
algorithm. These algorithms are provided by LAPACK routines as `dstevr`, `dstev`, and `dstedc`, respectively. In this thesis, we use the parallel implementation of them provided by Intel MKL. Note that all the numerical experiments are run with 16 threads.

Figure 4.2, 4.3, and 4.4 shows the performance of eigensolvers for computing all the eigenpairs of T_1, T_2, and T_3, respectively. Figures 4.2a, 4.3a, and 4.4a show the execution times for computing all the eigenpairs of each of these matrices. Figures 4.2b, 4.3b, and 4.4b show the orthogonality $\|Q^\top Q - I\|_\infty / n$ of all the computed eigenvectors $\boldsymbol{q}_k$ for $k = 1, \ldots, n$ by each eigensolvers, where $Q = \begin{bmatrix} \boldsymbol{q}_1 & \cdots & \boldsymbol{q}_n \end{bmatrix}$. Figures 4.2c, 4.3c, and 4.4c show the residual $\|TQ - QD\|_\infty / n$, where $D = \mathrm{diag}\{\tilde{\lambda}_1, \cdots, \tilde{\lambda}_n\}$.

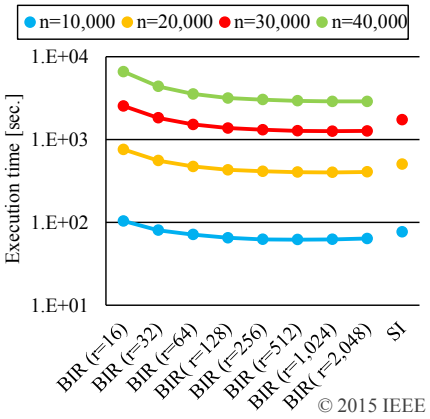
These graphs show that, in almost all the case of T_2 and T_3, the proposed eigensolver computes eigenpairs more accurately than the other algorithms while the proposed eigensolver are the slowest. The reason why the proposed eigensolver is so accurate is that eigenvalues are computed with highly relative accuracy by the bisection algorithm and thus make eigenvectors more accurate. On the other hand, in the case of T_1, the proposed eigensolver is faster than the MRRR algorithm and the QR algorithm but achieves the worse accuracy than the other algorithms. It should be noted the DC algorithm is the fastest in the almost all the case but fails in some cases of T_1. This is the fatal problem as a routine of the numerical library. In addition, in many case of T_2 and T_3, the MRRR algorithm is faster than the QR algorithm and the proposed eigensolver but the MRRR achieves the worst accuracy.

Now we consider the computational cost of each algorithm. The computational cost of the MRRR algorithm is always $O(n^2)$. The computational cost of the QR algorithm is $3bn^3$, where b denotes the average number of bulge chase per eigenvalue and is known to be usually 2 or 3; for more details, see [60]. The computational cost of the DC algorithm is $O(n^2)$ to $O(n^3)$. Almost all the computational cost of the proposed eigensolver is that required by the BIR algorithm and depends on the distribution of eigenvalues as follows: Since the number of iterations of the BIR algorithm is 3 in all the experiments, the computational cost of the BIR algorithm is $(20/21)n^3$ for large dimensional T_1 and is $12n^3$ for large dimensional T_2 and T_3. The execution times shown in Figures 4.2a, 4.3a, and 4.4a are in accordance with these cost analyses since the QR algorithm, the DC algorithm, and the BIR algorithm are mainly composed of the BLAS 3 routines such as the matrix multiplications and they are effectively accelerated by the parallelization. To the contrary, the MRRR algorithm is not fastest in spite of the superiority with respect to the computational cost since the MRRR is not mainly composed of the matrix multiplications resulting in the worse speedup ratios than the other algorithms.

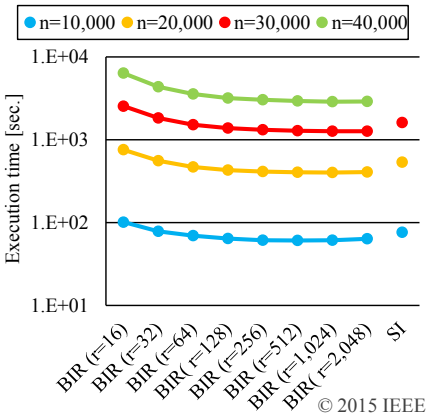
As mentioned earlier in Chapter 1, the QR and DC algorithms are not suitable in terms of the computational cost for computing a subset of eigenpairs, in particular, if the number of the desired eigenpairs are significantly smaller than the size of target matrices. Thus, from the results shown in this section, the proposed eigensolver is expected to be the best candidate for computing a subset of eigenpairs of matrices in actual applications with a high accuracy and reasonable execution time.



(a) Cases of T_1

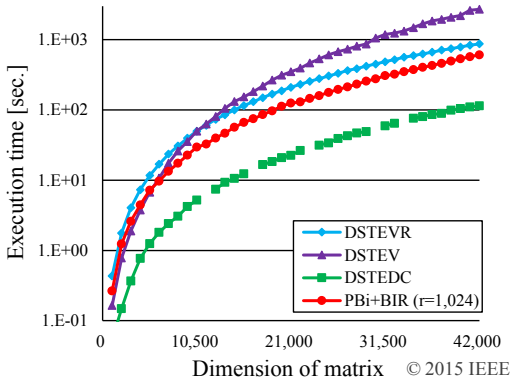


(b) Cases of T_2

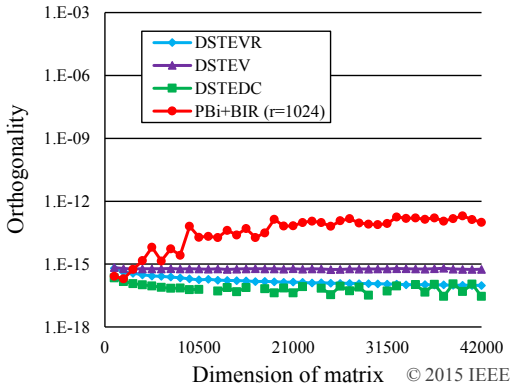


(c) Cases of T_3

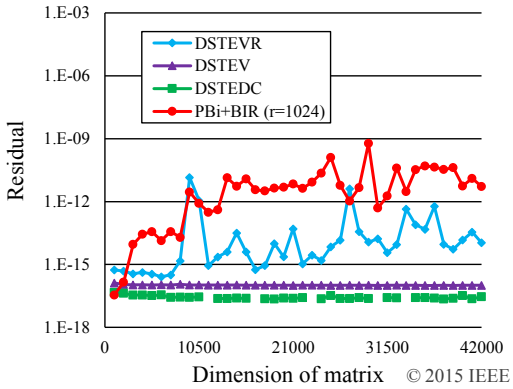
Figure 4.1: Execution times for computing all the eigenvectors using the BIR algorithm with different size of r and the SI algorithm [A6].



(a) Execution time

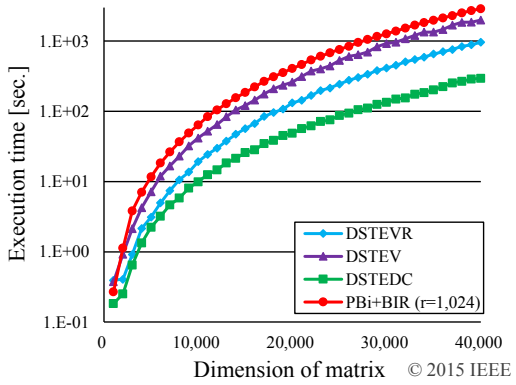


(b) Orthogonality $\|Q^\top Q - I\|_\infty / n$

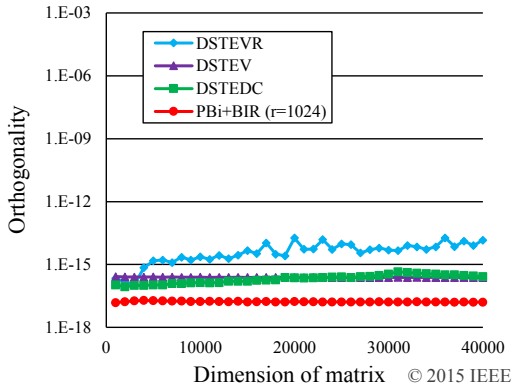


(c) Residual $\|TQ - QD\|_\infty / n$

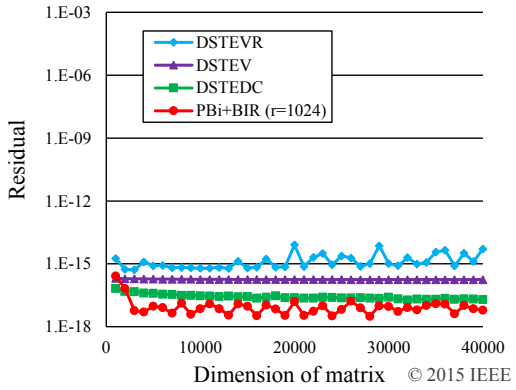
Figure 4.2: The performance of eigensolvers for computing all the eigenpairs of T_1, where n is a dimension of the target matrix [A6].



(a) Execution time

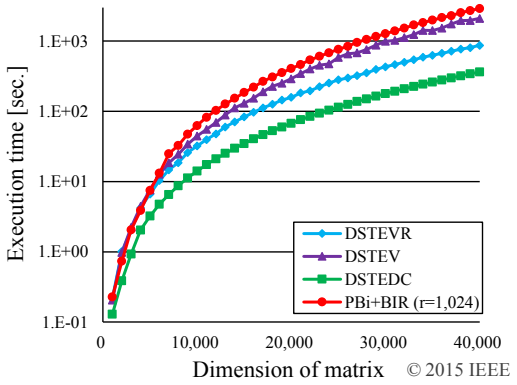


(b) Orthogonality $\|Q^\top Q - I\|_\infty / n$

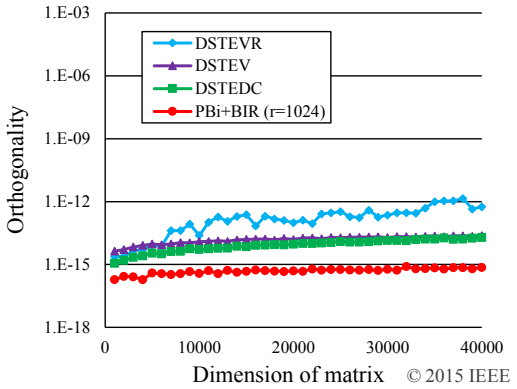


(c) Residual $\|TQ - QD\|_\infty / n$

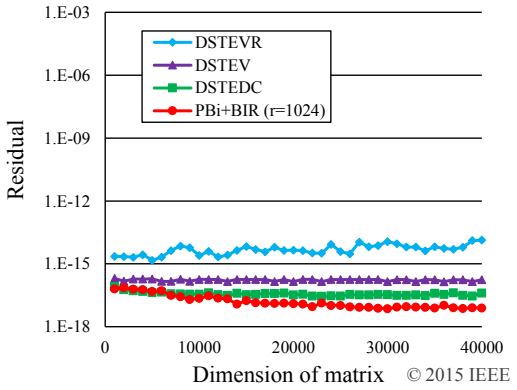
Figure 4.3: The performance of eigensolvers for computing all the eigenpairs of T_2, where n is a dimension of the target matrix [A6].



(a) Execution time



(b) Orthogonality $\|Q^\top Q - I\|_\infty / n$



(c) Residual $\|TQ - QD\|_\infty / n$

Figure 4.4: The performance of eigensolvers for computing all the eigenpairs of T_3, where n is a dimension of the target matrix [A6].

5 Subset Computation of Eigenpairs of Real Symmetric Band Matrices

In this chapter, the following parallel algorithm is proposed for the subset computation of eigenpairs of B, a real $n \times n$ symmetric band matrix with bandwidth w: The desired eigenvalues are computed by using the parallel implementation of Murata's bisection algorithm [42]. The corresponding eigenvectors are computed by using the block inverse iteration algorithm with reorthogonalization (BIR algorithm), whose implementation is modified for B from the implementation presented in 4.2. Then, the performance of the proposed methods is evaluated through numerical experiments on shared-memory multi-core processors.

The rest of this chapter is organized as follows: Section 5.1 gives a review of Gupta's bisection algorithm [40] and Murata's bisection algorithm and then gives their parallel implementations for shared-memory multi-core processors. Section 5.2 shows the inverse iteration algorithm with reorthogonalization and the BIR algorithm for computing eigenvectors of real symmetric band matrices and then gives their parallel implementations for shared-memory multi-core processors. Section 5.3 provides experimental results on shared-memory multi-core processors to evaluate the performance of the proposed symmetric band eigensolver.

5.1 Eigenvalue Computation of Symmetric Band Matrices

Similar to the cases of symmetric tridiagonal matrices shown in Section 2.1, both Gupta's and Murata's bisection algorithms compute the desired eigenvalues of B by repeatedly updating the half-open intervals $(\mu^L, \mu^R]$. The computation of $\nu_B(\mu)$ is also required for updating the intervals, where μ is a value in the intervals $(\mu^L, \mu^R]$ and $\nu_B(\mu)$ is the number of eigenvalues of B that are less than μ, and is the most time-consuming part of the bisection algorithms.

In this section, we introduce Gupta's bisection algorithm [40] and Murata's bisection algorithm [42] and then show parallel implementations of them. These two algorithms differ in ways to compute $\nu_B(\mu)$.

In the followings, let ℓ be the number of the desired eigenpairs of B.

5.1.1 Gupta's Bisection Algorithm

Gupta's bisection algorithm employs *Martin-Wilkinson's Gaussian elimination* [61] for computing $\nu_B(\mu)$.

The computation of $\nu_B(\mu)$ with Martin-Wilkinson's Gaussian elimination is based on Sturm's theorem [50], and thus require all the principal minor determinants of $B - \mu I$. Martin-Wilkinson's Gaussian elimination is adequate for this purpose, owing to its economical partial pivoting strategy, from the viewpoint of both the numerical stability and the computational cost. The computational cost of Martin-Wilkinson's Gaussian elimination is $O(w^2 n)$.

Martin-Wilkinson's Gaussian elimination is mainly composed of the BLAS 1 routines such as scaled vector additions. Whenever the partial pivoting occurs in Martin-Wilkinson's Gaussian elimination, the computational cost increases and, moreover, the pattern of data access changes. As a result, Gupta's bisection algorithm is difficult to achieve a high performance from the viewpoint of data reusability.

5.1.2 Murata's Bisection Algorithm

Murata's bisection algorithm employs not only Martin-Wilkinson's Gaussian elimination but also the LU factorization without pivoting for computing $\nu_B(\mu)$.

The computation of $\nu_B(\mu)$ with the LU factorization without pivoting is based on Sylvester's law of inertia [50]. That is, we perform the LU factorization without pivoting to have $B - \mu I = LU$, where L is an $n \times n$ lower triangular matrix with lower bandwidth w and U is an $n \times n$ unit upper triangular matrix with upper bandwidth w. On the basis of Sylvester's law of inertia, $\nu_B(\mu)$ is equivalent to the number of negative values in diagonal elements of L.

The computational cost of the LU factorization without pivoting is about one-third of Martin-Wilkinson's Gaussian elimination. In addition, the cache hit ratio of the LU factorization without pivoting is higher than that of Martin-Wilkinson's Gaussian elimination owing to absence of any pivoting. However, rounding errors and the absence of pivoting sometimes make the factorization failed or, even in the successful cases, give us incorrect elements in its result.

To cope with the inaccuracy of the factorization, Murata's bisection algorithm computes $\nu_B(\mu)$ in the following way: In the early stage of computing eigenvalues of B, Murata's bisection algorithm employs the LU factorization without pivoting for quick computation of $\nu_B(\mu)$. In addition, if μ is significantly close to a certain eigenvalue, Murata's bisection algorithm switches to Martin-Wilkinson's Gaussian elimination for accurate computation of $\nu_B(\mu)$. Consequently, Murata's bisection algorithm is expected to be faster than Gupta's algorithm for computing the desired eigenvalues of B.

The several acceleration techniques for the LU factorization algorithm has been proposed. For example, Hasegawa [62] proposed an implementation of Murata's bisection with an efficient factorization for vector processors. For recent scalar processors with large caches, on the other hand, it is known that the matrix blocking is effective for the LU factorization because the blocking achieves high cache hit ratio. Hence, in this chapter, the block LU factorization of real symmetric band matrices is introduced into Murata's bisection algorithm to have good performance on shared-memory multi-core processors.

Algorithm 5.1 Parallel Bisection Algorithm for A Real Symmetric Band Matrix B

1: **function** ParallelBandBisection(B, ℓ)
2: Set μ_1^L, μ_1^R ▷ Use Gerschgorin theorem, etc.
3: $k_b^{\text{next}} := 1,\ k_e^{\text{next}} := 1$
4: **repeat**
5: $k_b := k_b^{\text{next}},\ k_e := k_e^{\text{next}}$
6: `!$omp parallel do private(`μ_k`)`
7: **do** $k := k_b$ to k_e ▷ $k_e \leq \ell$
8: $\mu_k := (\mu_k^L + \mu_k^R)/2$
9: Compute $\nu_B(\mu_k)$
10: **end do**
11: Update μ_k^L, μ_k^R, and k_e^{next} for $k_b, \ldots, k_e$ (See Table 2.1)
12: Check μ_k^L and μ_k^R for $k = k_b, \ \ldots, \ k_e$ & Update μ_k^L, μ_k^R, and k_b^{next} if necessary (See lines 15-19 in Algorithm 2.1)
13: **until** $k_b^{\text{next}} > k_e^{\text{next}}$
14: $\mu_k := (\mu_k^L + \mu_k^R)/2$ for $k = 1, \ \ldots, \ \ell$
15: Sort μ_k for $k = 1, \ \ldots, \ \ell$ in descending order
16: **return** $\tilde{\lambda}_k := \mu_k$ for $k = 1, \ \ldots, \ \ell$
17: **end function**

5.1.3 Parallel Implementation of Bisection Algorithm

In this thesis, Murata's and Gupta's bisection algorithms are implemented in the way similar to LAPACK's `dstebz` and `dlaebz` [17] which we discussed in Section 2.1 for the case of tridiagonal matrices. A pseudocode of their parallel implementation is shown in Algorithm 5.1, which is similar to Algorithm 2.2.

The difference between Murata's and Gupta's algorithms is in the computation of $\nu_B(\mu_k)$ in line 9 as discussed in Section 5.1.1 and 5.1.2, respectively. In addition, the computation of $\nu_B(\mu_k)$ is performed in parallel with respect to each search point μ_k by employing the OpenMP directive shown in line 6.

The initialization of μ_1^L and μ_1^R is done by line 2. The update of μ_k^L, μ_k^R, k_b^{next}, and k_e^{next} is done in lines 11 and 12 to split intervals and to exclude some of them from the search if they are sufficiently small, as we did in Algorithm 2.2 based on the `dlaebz` routine. For more details, see Section 2.1, or, the `dstebz` and `dlaebz` routines provided in LAPACK.

Note that the above-mentioned parallel bisection algorithm requires the working memory for computing $\nu_B(\mu)$ independently on each computing thread. The amount of the working memory for Martin-Wilkinson's Gaussian elimination is $(3w + 1)n$ for each computing thread and is larger than that for the block LU factorization. Thus, we have to need $t(3w+1)n$ working memory for performing parallel Murata's and Gupta's bisection algorithm, where t is the number of the computing threads.

Algorithm 5.2 Inverse Iteration Algorithm with Reorthogonalization for A Real Symmetric Band Matrix B

function BandInv(B, ℓ, $\tilde{\lambda}_1$, $\ldots$, $\tilde{\lambda}_\ell$)
  do $k := 1$ to ℓ
    $i := 0$
    Generate an initial random vector: $\boldsymbol{v}_k^{(0)}$
    LU factorization with partial pivoting: $B - \tilde{\lambda}_k = P_k L_k U_k$ ▷ Call `dgbtrf`
    repeat
      $i := i + 1$
      Normalize $\boldsymbol{v}_k^{(i-1)}$ to $\boldsymbol{q}_k^{(i-1)}$
      Solve $P_k L_k U_k \boldsymbol{v}_k^{(i)} = \boldsymbol{q}_k^{(i-1)}$ ▷ Call `dgbtrs`
      if $k > 1$ and $|\tilde{\lambda}_{k-1} - \tilde{\lambda}_k| \leq 10^{-3} \times \max(|\tilde{\lambda}_1|, |\tilde{\lambda}_n|)$, **then**
        Reorthogonalize $\boldsymbol{v}_k^{(i)}$ to $\boldsymbol{q}_k^{(i)}$ by employing MGS algorithm
      else
        $k_1 := k$
      end if
    until some condition is met.
    Normalize $\boldsymbol{v}_k^{(i)}$ to $\boldsymbol{q}_k^{(i)}$
    $Q_k := \begin{bmatrix} Q_{k-1} & \boldsymbol{q}_k^{(i)} \end{bmatrix}$
  end do
  return $Q_\ell = \begin{bmatrix} \boldsymbol{q}_1 & \cdots & \boldsymbol{q}_\ell \end{bmatrix}$
end function

5.2 Eigenvector Computation of Symmetric Band Matrices

The inverse iteration algorithm with reorthogonalization is used in [40,42] for computing the eigenvectors of a real symmetric band matrix B. In this section, we consider applying the BIR algorithm in Section 4.2 for computing the eigenvectors of B, which is a variant of the inverse iteration algorithm with reorthogonalization.

5.2.1 Inverse Iteration Algorithm with Reorthogonalization

We first introduce the inverse iteration with reorthogonalization for computing the eigenvectors of B. In the followings, let λ_k be an eigenvalue of the target matrix such that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_\ell$ ($\ell \leq n$) and $\boldsymbol{q}_k$ be the eigenvector corresponding to λ_k, respectively. Besides them, let $\tilde{\lambda}_k$ be an approximate value of λ_k, obtained by some eigenvalue computation algorithm such as the bisection algorithm, and $\boldsymbol{v}_k^{(0)}$ be a starting vector for $k = 1, \ldots, \ell$. Then the inverse iteration for B generates a sequence of vectors $\boldsymbol{v}_k^{(i)}$ by iteratively solving the following linear equation:

$$\left(B - \tilde{\lambda}_k I\right) \boldsymbol{v}_k^{(i)} = \boldsymbol{v}_k^{(i-1)}, \quad i = 1, 2, \ldots, \tag{5.1}$$

where I is the $n \times n$ identity matrix. As well as the case of T described in Section 2.2.1, $\boldsymbol{v}_k^{(i)}$ quickly converges to $\boldsymbol{q}_k$ by a few iterations if the eigenvalues of B are well separated.

If some of the eigenvalues are very close to one another, we must reorthogonalize all the corresponding eigenvectors to these eigenvalues. *Peters-Wilkinson's criterion* [53] is applied in `dstein` [17], which is a LAPACK routine for computing eigenvectors of a real symmetric tridiagonal matrix T, as dividing eigenvalues of T into clusters. In Peters-Wilkinson's criterion, λ_{k-1} and λ_k are regarded as belonging to the same cluster if $|\tilde{\lambda}_{k-1} - \tilde{\lambda}_k| \leq 10^{-3}\|T\|_1$ is satisfied ($2 \leq k \leq \ell$). In the followings, we also use Peters-Wilkinson's criterion for dividing eigenvalues of a real symmetric band matrix B into clusters. However, $\|B\|_1 (= \|B\|_\infty)$ is not adequate to use in this criterion since $\|B\|_1$ becomes significantly large according to w, the bandwidth of B. To the contrary, since $\|B\|_2$ satisfies

$$\|B\|_2 = \sup_{\boldsymbol{x} \in \mathbb{R}^n} \frac{\|B\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} \geq \max_i |\lambda_i|, \tag{5.2}$$

a good lower bound of $\|B\|_2$ is given by $\max(\tilde{\lambda}_1, \tilde{\lambda}_n)$, where both $\tilde{\lambda}_1$ and $\tilde{\lambda}_n$ do not depend on w. Thus, in this chapter, Peters-Wilkinson's criterion for computing the eigenvectors of B is designed by using $\tilde{\lambda}_1$ and $\tilde{\lambda}_n$.

Algorithm 5.2 shows a pseudocode of the inverse iteration algorithm with reorthogonalization for computing the ℓ eigenvectors of B and is designed on the basis of the `dstein` routine. As well as the `dstein`, the modified Gram-Schmidt (MGS) algorithm [11] is applied to the reorthogonalization process in line 11. In line 10, Peters-Wilkinson's criterion with the above-mentioned modification is applied for dividing eigenvalues of B into clusters. For solving the linear equation (5.1), we once perform the LU factorization with the partial pivoting (PLU factorization) of $B - \tilde{\lambda}_k I$ by employing the `dgbtrf` routine (line 5), and then iteratively obtain $\boldsymbol{v}_k^{(i)}$ by employing the `dgbtrs` routine (line 9). Note that both `dgbtrf` and `dgbtrs` routines are provided in LAPACK. The `dgbtrf` routine is implemented on the basis of the block algorithm of the PLU factorization and is composed of the BLAS 2 and 3 routines. In addition, the `dgbtrs` routine is mainly composed of the BLAS 2 routines and requires P_k, L_k, and U_k, which are the resulting elements of the PLU factorization by the `dgbtrf` routine. For this purpose, we have to store P_k, L_k, and U_k in the working memory and their amount is about $(3w + 1)n$.

In this chapter, let us consider that the inverse iteration algorithm with reorthogonalization is parallelized by employing the parallel BLAS routines.

5.2.2 Block Inverse Iteration Algorithm with Reorthogonalization

A pseudocode of the block inverse iteration algorithm with reorthogonalization (BIR algorithm) for computing the eigenvectors corresponding to the clustered eigenvalues of B is shown in Algorithm 5.3, where $\hat{\ell}$ is the number of eigenvalues belonging to a certain cluster and r is a block parameter determined arbitrarily by users ($r \leq \hat{\ell}$). For convenience, we assume the r is a divisor of $\hat{\ell}$. Note that the BIR algorithm corresponds to the inverse iteration algorithm with reorthogonalization in Algorithm 5.2 if $r = 1$.

Algorithm 5.3 Block Inverse Iteration Algorithm with Reorthogonalization for A Real Symmetric Band Matrix B

1: **function** BANDBIR(B, r, $\hat{\ell}$, $\tilde{\lambda}_1$, $\ldots$, $\tilde{\lambda}_{\hat{\ell}}$)
2: Set an $n \times r$ matrix Q_0 be $Q_0 := \boldsymbol{O}$
3: **do** $j := 1$ to $\hat{\ell}/r$
4: $i := 0$
5: Generate $Q_{j,r}^{(0)} := \left[\boldsymbol{q}_{(j-1)r+1}^{(0)} \quad \cdots \quad \boldsymbol{q}_{jr}^{(0)}\right]$
6: `!$omp parallel do`
7: **do** $k =: (j-1)r+1$ to jr
8: PLU factorization: $B - \tilde{\lambda}_k = P_k L_k U_k$ ▷ Call `dgbtrf`
9: **end do**
10: **repeat**
11: $i := i + 1$
12: `!$omp parallel do`
13: **do** $k =: (j-1)r+1, \ldots, jr$
14: Solve $P_k L_k U_k \boldsymbol{v}_k^{(i)} = \boldsymbol{q}_k^{(i-1)}$ ▷ Call `dgbtrs`
15: **end do**
16: $V_{j,r}^{(i)} := V_{j,r}^{(i)} - Q_{(j-1)r} Q_{(j-1)r}^{\top} V_{j,r}$ ▷ Call `dgemm` ×2
17: QR factorization: $V_{j,r}^{(i)} = \overline{Q}_{j,r}^{(i)} R_{j,r}^{(i)}$
18: $\overline{Q}_{j,r}^{(i)} := \overline{Q}_{j,r}^{(i)} - Q_{(j-1)r} Q_{(j-1)r}^{\top} \overline{Q}_{j,r}^{(i)}$ ▷ Call `dgemm` ×2
19: QR factorization: $\overline{Q}_{j,r}^{(i)} = Q_{j,r}^{(i)} R_{j,r}^{(i)}$
20: **until** converge
21: $Q_{jr} := \left[Q_{(j-1)r} \quad Q_{j,r}^{(i)}\right]$ $\left(Q_r := \left[Q_{1,r}^{(i)}\right]\right)$
22: **end do**
23: **return** $Q_{\hat{\ell}} = \left[\boldsymbol{q}_1 \quad \cdots \quad \boldsymbol{q}_{\hat{\ell}}\right]$
24: **end function**

The BIR algorithm is composed of two processes as well as the inverse iteration algorithm with reorthogonalization. The one is to solve r linear equations simultaneously. For this process, the `dgbtrf` and `dgbtrs` routines are employed as well as the inverse iteration algorithm with reorthogonalization. Different from the inverse iteration algorithm with reorthogonalization, the computation of solving simultaneously r linear equations can be parallelized in terms of k since the linear equations are independent of each other. Thus, the computation of this process is parallelized by the OpenMP directives shown in lines 6-9 and lines 12-15. Note that we have to spend $r(3w+1)n$ working memory to store P_k, L_k, and U_k corresponded to the r linear equations for this parallelization.

The other is the block reorthogonalization process as shown in lines 16-19. In Algorithm 5.3, the block Gram-Schmidt algorithm with reorthogonalization (BCGS2 algorithm) [59] is used for this process. The BCGS2 algorithm is mainly composed of the matrix multiplication, which is one of the BLAS 3 routines. Thus, the `dgemm` routines are employed to the computation in lines 16 and 18 in Algorithm 5.3. As well as the

Table 5.1: Specifications of the experimental environment.
One node of Appro Green Blade 8000 at ACCMS

CPU	Intel Xeon E5-2670@2.6GHz, 16 cores (8 cores × 2)
	L3 cache: 20MB × 2
RAM	DDR3-1600 64GB, 136.4GB/sec
Compiler	Intel C++/Fortran Compiler 15.0.2
Options	`-O3 -xHOST -ipo -no-prec-div`
	`-openmp -mcmodel=medium -shared-intel`
Software	Intel Math Kernel Library 11.2.2

BIR algorithm proposed in Section 4.2, we consider that the recursive implementation of the classical Gram-Schmidt algorithm [57] is applied to the computation of the QR factorization in lines 17 and 19, which is also mainly composed of the matrix multiplications. In this chapter, the block reorthogonalization process is parallelized by employing the parallel BLAS routines.

The BIR algorithm corresponds to the simultaneous inverse iteration algorithm [46] if $r = \hat{\ell}$. Thus, the simultaneous inverse iteration algorithm always spends the larger amount of memory than the BIR algorithm does. Note that the memory use for the BIR algorithm is almost equal to that for the parallel bisection algorithms described in 5.1.3 if r is set to the number of the computing threads.

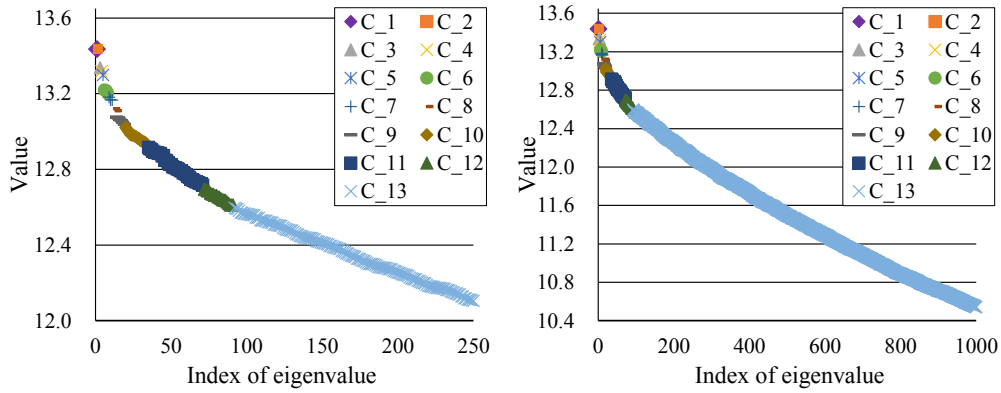
5.2.3 Remark on Computational Cost

The relationship between the inverse iteration algorithm with reorthogonalization and the BIR algorithm is analogous to that between the LU factorization and the block LU factorization. Thus, the computational cost of the BIR algorithm is almost equal to that of the inverse iteration algorithm with reorthogonalization.

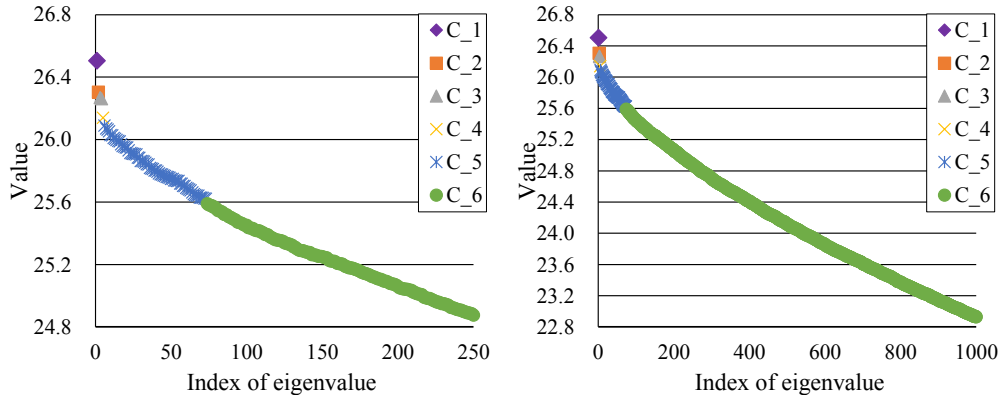
As mentioned before, both the inverse iteration algorithm with reorthogonalization and the BIR algorithm are composed two processes: solving linear equation and the (block) reorthogonalization process. Assuming ℓ is the number of the desired eigenvectors of B, the computational cost is $O(\ell w^2 n)$ for solving linear equations and is $O(\hat{\ell}_{\max}^2 n)$ for the reorthogonalization process, where $\hat{\ell}_{\max}$ is the number of eigenvalues belonging to the largest eigenvalue cluster ($\hat{\ell}_{\max} \leq \ell$). As a result, solving linear equations occupies most of the execution time for computing eigenvectors of B by the inverse iteration algorithm with reorthogonalization as long as it is not a case that ℓ is much larger than w. The same is true of the BIR algorithm.

5.3 Performance Evaluation

This section gives experimental results on shared-memory multi-core processors to evaluate the performance of the proposed eigensolver, which computes eigenvalues of real symmetric band matrices by employing parallel Murata's bisection algorithm



(a) Cases of B_1: (left) the 250 largest eigenvalues; (right) the 1000 largest eigenvalues.



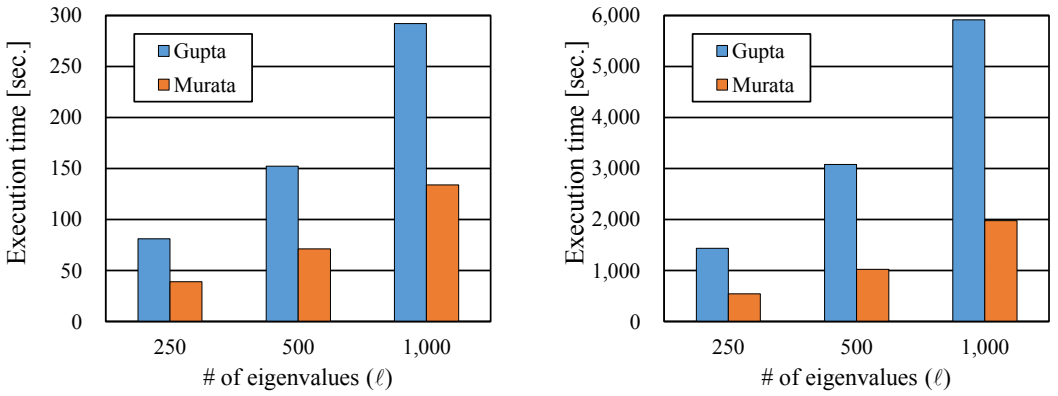
(b) Cases of B_2: (left) the 250 largest eigenvalues; (right) the 1000 largest eigenvalues.

Figure 5.1: Distributions of the 250 and 1000 largest eigenvalues of real symmetric band matrices, which belong to the cluster C_i ($i = 1, \ldots, 13$ for B_1 and $i = 1, \ldots, 6$ for B_2) in Peters-Wilkinson's criterion with the modification given in Section 5.2.1.

discussed in Section 5.1 and computes the corresponding eigenvectors by the BIR algorithm given in Section 4.2.

All the experiments were performed with 16 threads on one node of Appro Green Blade 8000 at ACCMS, Kyoto University, whose specification is listed in Table 5.1. The Intel Math Kernel Library (MKL) is used for the parallel execution of the BLAS and LAPACK routines and the OpenMP directives are also used for the thread parallelization as shown in Section 5.1 and 5.2. The block size r of the BIR algorithm in the proposed eigensolver was set to $r = 16$ in all the experiments, which is equal to the number of cores in the experimental environment shown in Table 5.1. Since the working memory for the BIR algorithm is almost equal to that for the parallel bisection algorithm, the total memory use can be easily estimated on this condition. Note that the maximum number of iterations in both the BIR algorithm and the inverse iteration algorithm with reorthogonalization is set to 5, as well as the `dstein` routine of LAPACK. In fact, the number of iterations in both of them is 3 in all the experiments.

We used two $n \times n$ symmetric band matrices with bandwidth w, namely B_1 and B_2, as



(a) Cases of B_1 (b) Cases of B_2

Figure 5.2: Execution times for computing the ℓ largest eigenvalues of real symmetric band matrices by using parallel Murata's bisection algorithm and parallel Gupta's bisection algorithm.

the target matrices of the performance evaluation, whose elements are uniform random numbers in the range $(0, 1)$: $n = 20,000$ and $w = 64$ for B_1; and $n = 40,000$ and $w = 256$ for B_2. In the experiments, the ℓ largest eigenvalues of them and the corresponding eigenvectors are computed, where $\ell \in \{250, 500, 1000\}$. Figures 5.1 show distributions of the 250 and 1000 largest eigenvalues of B_1 and B_2, where C_i ($i = 1, \ldots, 13$ for B_1 and $i = 1, \ldots, 6$ for B_2) denotes eigenvalue clusters of each matrix in Peters-Wilkinson's criterion with the modification given in Section 5.2.1. As can be seen in these figures, many eigenvalues belong to one cluster, i.e., C_{13} for B_1 and C_6 for B_2. In fact, C_{13} for B_1 contains $\tilde{\lambda}_{92}, \tilde{\lambda}_{93}, \ldots, \tilde{\lambda}_{1000}$ and C_6 for B_2 contains $\tilde{\lambda}_{74}, \tilde{\lambda}_{75}, \ldots, \tilde{\lambda}_{1000}$. It should be noted that the eigenvalue distribution of random symmetric matrices such as B_1 and B_2 is known to be similar to that of the matrices appearing in many applications.

5.3.1 Performance Evaluation of Murata's Bisection Algorithm

To evaluate the performance of parallel Murata's bisection algorithm in Section 5.1, we compared its execution time for computing the desired eigenvalues of real symmetric band matrices with that of Gupta's bisection algorithm. Their codes are parallelized by employing the OpenMP directives as shown in Section 5.1.3.

Figures 5.2a and 5.2b show the execution times for computing the ℓ largest eigenvalues of B_1 and B_2, respectively. According to the expectation in Section 5.1.2, we observe that Murata's bisection algorithm is much faster than Gupta's bisection algorithm in all the cases.

Tables 5.2 show the number of computations of $\nu_B(\mu)$ on the basis of the block LU factorization algorithm and Martin-Wilkinson's Gaussian elimination. These Tables indicate that most of the computations of $\nu_B(\mu)$ are performed by the block LU factorization-based algorithm in Murata's bisection. In addition, the total number of computations of $\nu_B(\mu)$ in Murata's bisection is almost the same as that in Gupta's bisection. We also observe that the total number of computations of $\nu_B(\mu)$ in both bisection

Table 5.2: The number of computations of $\nu_B(\mu)$ on the basis of the block *LU* factorization algorithm (Block LU) and Martin-Wilkinson's Gaussian elimination (M-W) in computing the ℓ largest eigenvalues by employing parallel Murata's bisection algorithm and parallel Gupta's bisection algorithm

(a) Cases of B_1

# of eigenvalues (ℓ)		250	500	1,000
Murata	Block LU	9,538	18,802	36,852
	M-W	108	209	692
Gupta	M-W	9,896	19,511	38,544

(b) Cases of B_2

# of eigenvalues (ℓ)		250	500	1,000
Murata	Block LU	9,030	17,728	34,719
	M-W	186	506	1,449
Gupta	M-W	9,216	18,234	36,168

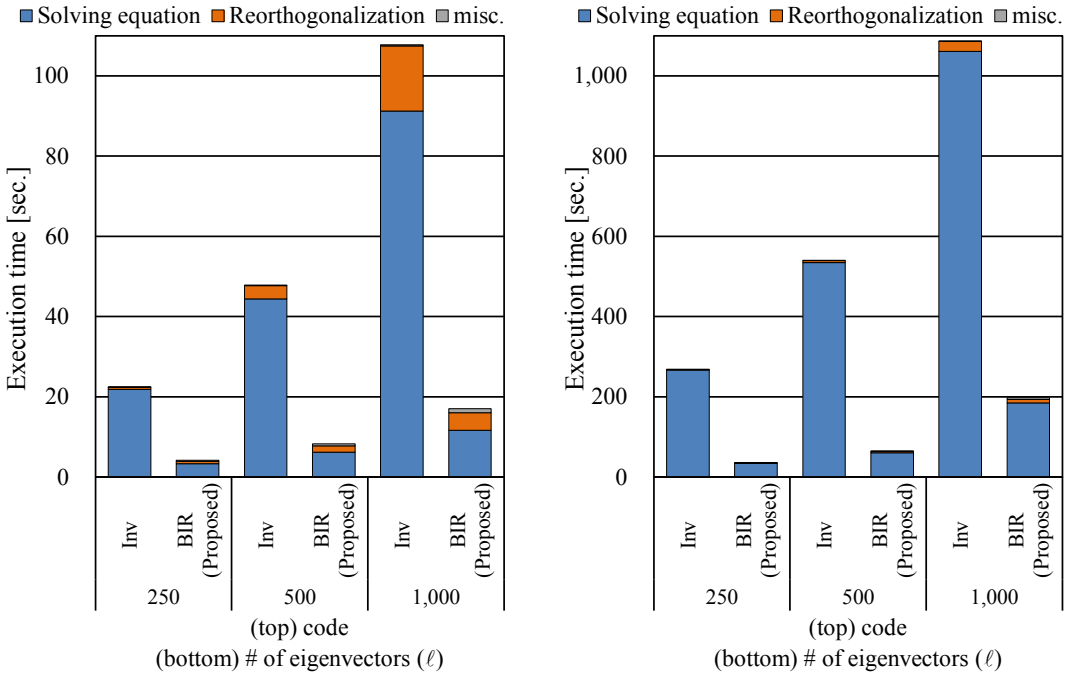
algorithms is determined by ℓ, the number of the desired eigenvalues, rather than the matrix size or the bandwidth.

5.3.2 Performance Evaluation of BIR Algorithm

In order to evaluate the performance of the proposed eigenvector computation algorithm given in Section 5.2, we compared the execution time of the proposed algorithm (**BIR**) for computing of the eigenvectors corresponding to the ℓ largest eigenvalues of real symmetric band matrices with that of the inverse iteration algorithm with reorthogonalization (**Inv**). Their codes are parallelized by employing the Intel MKL and the OpenMP directives as shown in Section 5.2. In addition, the ℓ largest eigenvalues of the target matrices are obtained by using parallel Murata's bisection algorithm.

Figure 5.3 shows the execution times for computing the corresponding eigenvectors to the ℓ largest eigenvalues of the target matrices and their details, where "Solving equation" denotes the execution time for solving linear equations (5.1) and "Reorthogonalization" denotes the execution time for the reorthogonalization process performed by the MGS algorithm in **Inv** or the BCGS2 algorithm in **BIR**. Figures 5.3a and 5.3b correspond to the cases of B_1 and B_2, respectively. These figures show that **BIR** is much faster than **Inv** in all the cases. According to the discussion about the computational cost of each process in Section 5.2.3, the "Solving equation" process occupies most of the execution time of the eigenvector computation in all the cases.

We also observe that the execution time for the "Solving equation" process in **BIR** is significantly shorter than that of **Inv** in all the cases. As discussed in Section 5.2, the parallelization of the "Solving equation" process in **BIR** differs from that in **Inv**. In the BIR algorithm, each linear equation is solved on each computing thread, and thus any barrier synchronization between the computing threads does not occur until all the computation assigned to each computing thread is finished. On the other hand, since



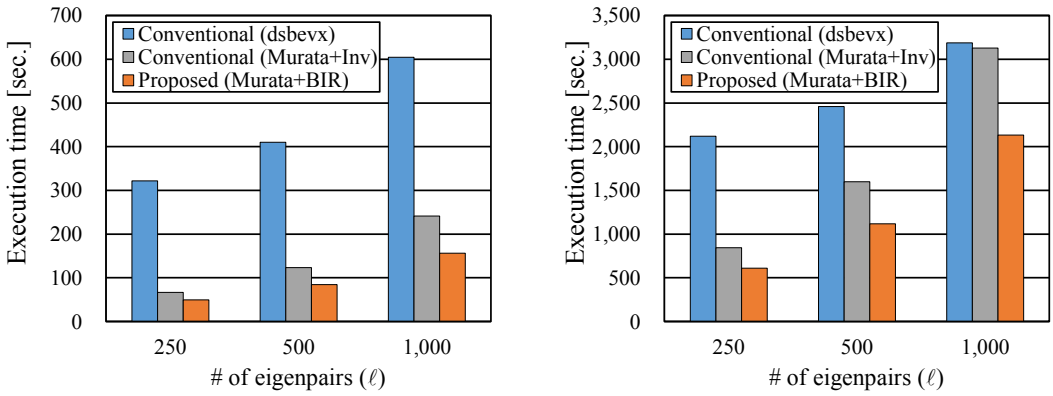
(a) Cases of B_1 (b) Cases of B_2

Figure 5.3: Execution times for computing the corresponding eigenvectors to the ℓ largest eigenvalues of symmetric band matrices by using the block inverse iteration algorithm with reorthogonalization (**BIR**) and the inverse iteration algorithm with reorthogonalization (**Inv**) and their details.

the inverse iteration algorithm with reorthogonalization is parallelized by employing the parallel BLAS routines, the barrier synchronization between the computation threads occurs each time the BLAS routine is called. Moreover, the BLAS-based computations in the `dgbtrf` and `dgbtrs` routines are difficult to achieve good performance in parallel processing since the size of vectors and matrices appearing in these computations is too small. This is one of the reasons why the "Solving equation" process in **BIR** achieves the higher performance in parallel processing than that of **Inv**.

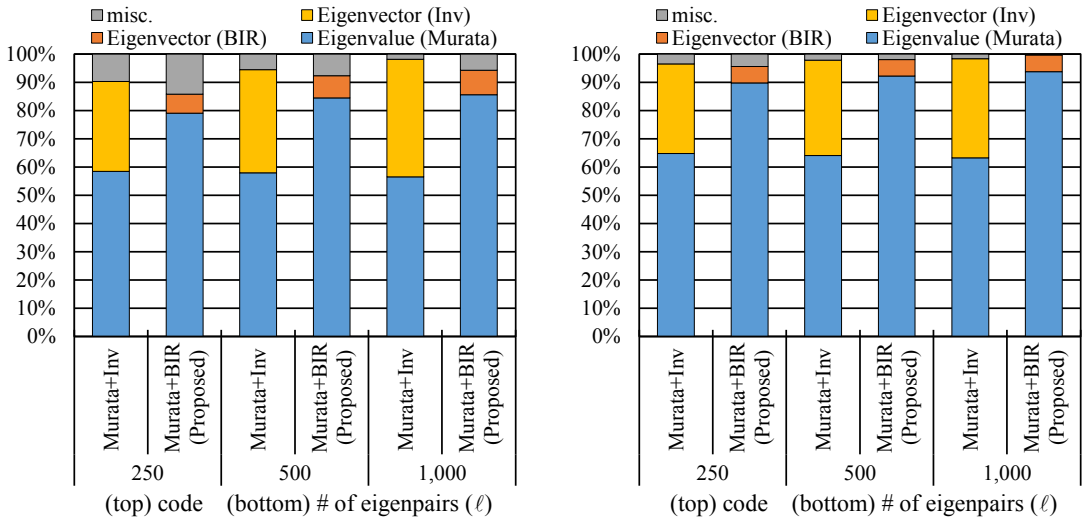
5.3.3 Performance Evaluation of Proposed Band Eigensolver

In order to evaluate the performance of the proposed symmetric band eigensolver, we compared the proposed solver (**Murata+BIR**) with the two conventional eigensolvers. One of the conventional solvers is also to compute eigenvalues by using Murata's bisection algorithm and to compute the corresponding eigenvectors by using the inverse iteration algorithm with reorthogonalization shown in Algorithm 5.2 and is referred to as **Murata+Inv**. The codes of **Murata+BIR** and **Murata+Inv** are also parallelized by employing the Intel MKL and the OpenMP directives as shown in Section 5.1.3 and Section 5.2. The other conventional solver is **dsbevx** provided in Intel MKL, which is a LAPACK routine for computing a subset of eigenpairs of real symmetric band matrices through the tridiagonalization. Note that **dsbevx** employs the `dsbtrd` routine to



(a) Cases of B_1 (b) Cases of B_2

Figure 5.4: Execution times for computing the eigenpairs corresponding to the ℓ largest eigenvalues of real symmetric band matrices by using the proposed solver (**Murata+BIR**) and the conventional solvers (**Murata+Inv** and **dsbevx**).

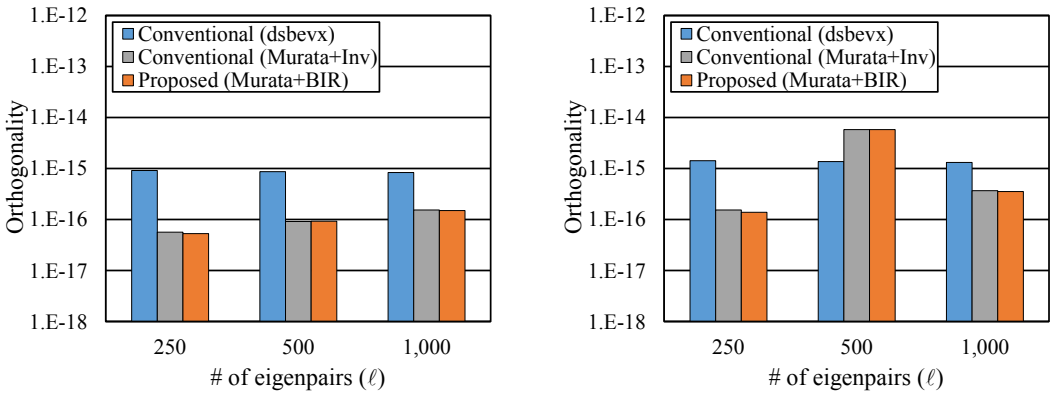


(a) Cases of B_1 (b) Cases of B_2

Figure 5.5: Details of the execution times for computing the eigenpairs corresponding to the ℓ largest eigenvalues of real symmetric band matrices by using **Murata+BIR** and **Murata+Inv**.

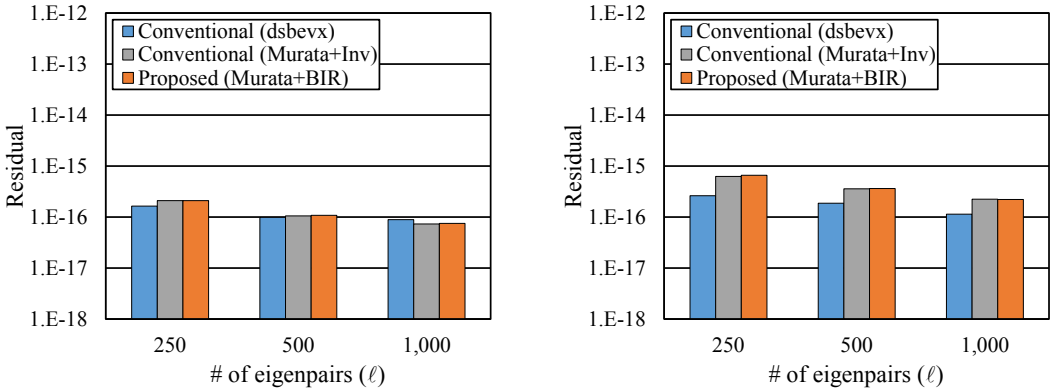
tridiagonalize the target band matrix in the way proposed in [38], the `dstebz` routine to compute the desired eigenvalues of the real symmetric tridiagonal matrix, and the `dstein` routine to compute the corresponding eigenvectors.

Figures 5.4a and 5.4b show the overall execution time for computing the eigenpairs corresponding to the ℓ largest eigenvalues of B_1 and B_2, respectively. We observe that the proposed eigensolver, **Murata+BIR**, is faster than the conventional solvers in all the cases. Figures 5.5a and 5.5b show the details of the overall execution time for **Murata+BIR** and **Murata+Inv**. We observe that most of the execution time in **Murata+BIR** remains to be occupied by that for the eigenvalue computation using parallel



(a) Cases of B_1 (b) Cases of B_2

Figure 5.6: Orthogonality $\|Q_\ell^\top Q_\ell - I\|_\infty/\ell$ of the corresponding eigenvectors to the ℓ largest eigenvalues of real symmetric band matrices by using the proposed solver (**Murata+BIR**) and the conventional solvers (**Murata+Inv** and **dsbevx**).



(a) Cases of B_1 (b) Cases of B_2

Figure 5.7: Residual $\|B_i Q_\ell - Q_\ell D_\ell\|_\infty/\ell$ of the ℓ largest eigenpairs of real symmetric band matrices by using the proposed solver (**Murata+BIR**) and the conventional solvers (**Murata+Inv** and **dsbevx**).

Murata's bisection algorithm in all the cases. One reason of this result is that the computational cost of **Murata**, Murata's bisection algorithm, is much higher than that of **BIR**. The other reason is that the execution time for eigenvector computation in **Murata+BIR** is significantly reduced from that in **Murata+Inv** as shown in Section 5.3.2.

The accuracy of the eigenpairs computed by using the proposed solver and the conventional solvers is shown as follows: Figures 5.6a and 5.6b show the orthogonality $\|Q_\ell^\top Q_\ell - I\|_\infty/\ell$ of B_1 and B_2, respectively. Similarly, Figures 5.7a and 5.7b show the residual $\|B_i Q_\ell - Q_\ell D_\ell\|_\infty/\ell$ of B_1 and B_2, respectively. Note that $D_\ell = \mathrm{diag}(\tilde{\lambda}_1, \cdots, \tilde{\lambda}_\ell)$ and $Q_\ell = \begin{bmatrix} \boldsymbol{q}_1 & \cdots & \boldsymbol{q}_\ell \end{bmatrix}$. These figures show that the proposed eigensolver computes the desired eigenpairs as accurately as the conventional solvers.

6 Subset Computation of Singular Triplets of Large Sparse Matrices

In this chapter, we discuss parallel solvers for the subset computation of singular triplets of a real $m \times n$ sparse matrices M on the basis of the *Golub-Kahan-Lanczos algorithm with reorthogonalization (GKLR algorithm)* [2, 43] ($m \geq n$). The GKLR algorithm iteratively generates bidiagonal matrices with much smaller size than n and requires the subset computation of singular triplets of the bidiagonal matrices. The bisection and inverse iteration algorithms for computing a subset of eigenpairs of real symmetric tridiagonal matrices can be applied for this purpose [2]. In the followings, we thus consider parallel implementations of the GKLR algorithm combined with the bisection and inverse iteration algorithms.

The reorthogonalization process of the GKLR algorithm is a bottleneck of the execution time in parallel processing if the size of M is large. To accelerate the computation of the reorthogonalization, an alternative parallel implementation of the *classical Gram-Schmidt algorithm with reorthogonalization (CGS2 algorithm)* [47] is proposed. Although the CGS2 algorithm is conventionally parallelized by employing the parallel BLAS routines, the proposed implementation of the CGS2 algorithm is parallelized on the basis of OpenMP [48]. Hereafter, the proposed implementation of the CGS2 algorithm is referred to as the *OMP-CGS2 algorithm*. The OMP-CGS2 algorithm enables us to effectively use the cache of CPUs especially on shared-memory multi-core processors. Thus, the OMP-CGS2 algorithm is expected to achieve higher performance in parallel processing than the conventional reorthogonalization algorithms, which are parallelized by employing the parallel BLAS routines.

The rest of this chapter is organized as follows: Section 6.1 gives an introduction of the GKLR algorithm. Section 6.2 shows the implementation of the GKLR algorithm, which includes the adoption of the bisection and inverse iteration algorithms. Section 6.3 provides an introduction of the CGS2 algorithm and then presents the OMP-CGS2 algorithm. Section 6.4 provides experimental results on shared-memory multi-core processors to evaluate the performance of a proposed GKLR algorithm, which is implemented with the OMP-CGS2 algorithm. In Section 6.5, 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.

6.1 GKLR Algorithm

The *Golub-Kahan-Lanczos (GKL algorithm)* [2, 11] algorithm, which is the origin of the GKLR 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 $\mathcal{K}(M^\top M, \boldsymbol{p}_1, k)$ and the $\boldsymbol{q}_k$ is an orthonormal basis of the alternative Krylov subspace $\mathcal{K}(MM^\top, M\boldsymbol{p}_1, k)$, where

$$\mathcal{K}(M^\top M, \boldsymbol{p}_1, k) = \mathrm{span}\left\{\boldsymbol{p}_1, (M^\top M)\boldsymbol{p}_1, (M^\top M)^2\boldsymbol{p}_1, \cdots, (M^\top M)^{k-1}\boldsymbol{p}_1\right\}, \tag{6.1}$$

$$\mathcal{K}(MM^\top, M\boldsymbol{p}_1, k) = \mathrm{span}\left\{M\boldsymbol{p}_1, (MM^\top)M\boldsymbol{p}_1, (MM^\top)^2M\boldsymbol{p}_1, \cdots, (MM^\top)^{k-1}M\boldsymbol{p}_1\right\}. \tag{6.2}$$

In the GKLR algorithm [43], 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 6.1 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 6.1, the $k \times k$ approximate matrices

$$B_k = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ & \alpha_2 & \beta_2 & & \\ & & \ddots & \ddots & \\ & & & \alpha_{k-1} & \beta_{k-1} \\ & & & & \alpha_k \end{bmatrix} \tag{6.3}$$

are obtained and the following equations hold

$$MP_k = Q_k B_k, \tag{6.4}$$

$$M^\top Q_k = P_k B_k^\top + \beta_k \boldsymbol{p}_{k+1}\boldsymbol{e}_k^\top, \tag{6.5}$$

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 M, we can stop the iterations of the GKLR algorithm. In line 8 of Algorithm 6.1, we check whether the ℓ largest singular values of B_k sufficiently approximate those of M or not. Criteria for this check are discussed in Section 6.2.1.

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 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:

$$\boldsymbol{u}_j^{(k)} = Q_k \boldsymbol{s}_j^{(k)}, \ \boldsymbol{v}_j^{(k)} = P_k \boldsymbol{t}_j^{(k)}. \tag{6.6}$$

In order to improve the accuracy of singular vectors, this computation is implemented by combining with the QR factorization [63].

As seen in Algorithm 6.1, 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.

Algorithm 6.1 GKLR algorithm

Set an unit vector $\boldsymbol{p}_1 \in \mathbb{R}^n$
$\boldsymbol{q} := M\boldsymbol{p}_1,\ \alpha_1 := \|\boldsymbol{q}\|_2,\ \boldsymbol{q}_1 := \boldsymbol{q}/\alpha_1$
$P_1 := [\boldsymbol{p}_1], Q_1 := [\boldsymbol{q}_1]$
do $k := 1,\ 2,\ \ldots$
    $\boldsymbol{p} := M^\top \boldsymbol{q}_k$
    $\tilde{\boldsymbol{p}} := \mathrm{Reorthogonalization}(P_k,\ \boldsymbol{p})$
    $\beta_k := \pm\|\tilde{\boldsymbol{p}}\|_2, \boldsymbol{p}_{k+1} := \tilde{\boldsymbol{p}}/\beta_k$
    Check the singular values of B_k
    $\boldsymbol{q} := M\boldsymbol{p}_{k+1}$
    $\tilde{\boldsymbol{q}} := \mathrm{Reorthogonalization}(Q_k,\ \boldsymbol{q})$
    $\alpha_{k+1} := \pm\|\tilde{\boldsymbol{q}}\|_2, \boldsymbol{q}_{k+1} := \tilde{\boldsymbol{q}}/\alpha_{k+1}$
    $P_{k+1} := \begin{bmatrix} P_k & \boldsymbol{p}_{k+1} \end{bmatrix}, Q_{k+1} := \begin{bmatrix} Q_k & \boldsymbol{q}_{k+1} \end{bmatrix}$
end do

6.2 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 M or not and a stopping strategy of the GKLR algorithm. Then we present how to combine the bisection and inverse iteration algorithms for computing real symmetric tridiagonal matrices with the GKLR algorithm.

6.2.1 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 [50] 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:

$$B_k \boldsymbol{t}_j^{(k)} = \sigma_j^{(k)} \boldsymbol{s}_j^{(k)},\ B_k^\top \boldsymbol{s}_j^{(k)} = \sigma_j^{(k)} \boldsymbol{t}_j^{(k)}. \tag{6.7}$$

Using Eqs. (6.4), (6.5), (6.6), and (6.7), we obtain

$$\begin{aligned} M^\top \boldsymbol{u}_j^{(k)} - \sigma_j^{(k)} \boldsymbol{v}_j^{(k)} &= M^\top Q_k \boldsymbol{s}_j^{(k)} - \sigma_j^{(k)} P_k \boldsymbol{t}_j^{(k)} \\ &= \left(M^\top Q_k - P_k B_k^\top \right) \boldsymbol{s}_j^{(k)} \\ &= \beta_k \boldsymbol{p}_{k+1} \boldsymbol{e}_k^\top \boldsymbol{s}_j^{(k)} \\ &= \beta_k s_j^{(k)}(k) \boldsymbol{p}_{k+1}, \end{aligned} \tag{6.8}$$

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

$$\left\| M^\top \boldsymbol{u}_j^{(k)} - \sigma_j^{(k)} \boldsymbol{v}_j^{(k)} \right\|_2 = \left| \beta_k s_j^{(k)}(k) \right|. \tag{6.9}$$

As the results, if the right-hand side of inequality (6.9) is sufficiently small, then the singular value $\sigma_j^{(k)}$ of B_k can be regarded to sufficiently approximate that of M. Hence,

Algorithm 6.2 Stopping strategy of GKLR algorithm

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

the following inequality is considered as one of the stopping criteria of the GKLR algorithm:

$$\left|\beta_k s_j^{(k)}(k)\right| \leq \delta, \quad j = 1, \ldots, \ell, \tag{6.10}$$

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 (6.10), 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 (6.10) is satisfied or not. The computational cost for computing the ℓ singular triplets of B_k is higher than that for this check. We should reduce the computational cost for this subset computation in order to reduce the total execution time for the GKLR algorithm. Hereafter, let k_t be the number of iterations where inequality (6.10) is satisfied for the first time.

Now let us consider the following inequality, which is one of the necessary conditions for inequality (6.10):

$$\left|\beta_k s_\ell^{(k)}(k)\right| \leq \delta. \tag{6.11}$$

We have to compute only the ℓ-th largest singular triplet of B_k in order to check whether inequality (6.11) is satisfied. Hence, from the viewpoint of the computational cost, inequality (6.11) is more suitable for the stopping criterion of the GKLR algorithm than inequality (6.10). In addition, for the iteration ordinal k_n at which inequality (6.11) 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 (6.11) can be also considered as one of the stopping criteria of the GKLR algorithm. However, since the theorems in [64] 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 (6.11) is satisfied, we must check whether inequality (6.10) is also satisfied for all j.

Summarizing the discussions above, the stopping strategy for the GKLR algorithm is shown by Algorithm 6.2. In the experiments presented in Section 6.4, we set $\delta = 1.0 \times 10^{-14}$ as the stopping criterion. Note that Algorithm 6.2 is used in Algorithm 6.1 for its line 8. After stopping the iteration of the GKLR algorithm, we compute the ℓ largest singular triplets of M, i.e., $(\sigma_j, \boldsymbol{u}_j, \boldsymbol{v}_j)$ for $j = 1, \ldots, \ell$, using Eqs. (6.6).

6.2.2 Subset Computation for Singular Triplets of Approximate Matrices

As mentioned in Section 6.2.1, 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 6.2.

The approximate matrix B_k, generated by the GKLR algorithm, is a lower bidiagonal matrix. As mentioned in [2], 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 [50]. The BI algorithm enables us to compute only the desired eigenpairs and is suitable for the subset computation of singular triplets in Algorithm 6.2. While computing ℓ singular triplets (line 3 in Algorithm 6.2), 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.

6.3 OMP-CGS2 Algorithm

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, the OMP-CGS2 algorithm is proposed for accelerating the computation of the reorthogonalization process. The OMP-CGS2 algorithm is algebraically equivalent to the CGS2 algorithm [47], but is parallelized by employing not the parallel BLAS routines but the OpenMP directives.

In the followings, we discuss the computation of $\tilde{\boldsymbol{x}}_i \in \mathbb{R}^m$, which is the orthogonalized vector of $\boldsymbol{a}_i \in \mathbb{R}^m$ $(2 \leq i \leq n)$ and satisfies $\langle \tilde{\boldsymbol{x}}_i,\ \boldsymbol{x}_j \rangle = 0$ for $j \neq i$, where $\boldsymbol{x}_j = \tilde{\boldsymbol{x}}_j / \|\tilde{\boldsymbol{x}}_j\|$. In addition, let X_{i-1} be $X_{i-1} = \begin{bmatrix} \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_{i-1} \end{bmatrix}$ $(2 \leq i \leq n)$. 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 6.1, and also correspond to Q_k, $\tilde{\boldsymbol{q}}$, and $\boldsymbol{q}$ of line 10 in Algorithm 6.1.

6.3.1 CGS2 Algorithm and Its Parallel Implementation

The classical Gram-Schmidt (CGS) algorithm [11] is a well-known reorthogonalization algorithm. The reorthogonalization of $\boldsymbol{a}_i$ using the CGS algorithm is formulated as follows:

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{a}_i - \sum_{j=1}^{i-1} \langle \boldsymbol{x}_j,\ \boldsymbol{a}_i \rangle \boldsymbol{x}_j. \tag{6.12}$$

Algorithm 6.3 CGS2 algorithm

```
1: function CGS2(X_{i-1}(:= [x_1, ..., x_{i-1}]), a_i)
2:     do I := 1, 2
3:         w := X_{i-1}^T a_i
4:         a_i := a_i − X_{i-1} w
5:     end do
6:     return x̃_i := a_i
7: end function
```

Eq. (6.12) is composed of the BLAS 1 routines, such as inner-dot product and scaled vector addition. 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. (6.12) is also replaced as

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{a}_i - X_{i-1}X_{i-1}^{\top}\boldsymbol{a}_i. \tag{6.13}$$

In general, to achieve better performance, we reduce the number of data synchronizations on shared-memory multi-core processors as much as possible. The BLAS 2 routines, such as the matrix-vector multiplications, have less data synchronization than the BLAS 1 routines. Thus, the BLAS 2 routines achieves better performance than the BLAS 1 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) [47], which repeats the CGS algorithm twice. A pseudocode of the CGS2 is shown in Algorithm 6.3. Although the orthogonality of computed vectors by the CGS2 algorithm is theoretically better than that by the CGS algorithm [47], the computational cost of the CGS2 is twice higher than that of the CGS.

6.3.2 OMP-CGS2 Algorithm

Recalling Eq. (6.12), 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 6.4. 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 [65], 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.

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

Algorithm 6.4 OMP-CGS2 algorithm

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

Table 6.1: Comparison of reorthogonalization algorithms [47, 54]

	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.

scaled vector addition (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 $\boldsymbol{a}_i$ in 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.

6.3.3 Comparison of Reorthogonalization Algorithms

As the summary of this section, the theoretical performance of the reorthogonalization algorithms is shown in Table 6.1, 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 com-

posed. In addition, ϵ is the machine epsilon and $\kappa(A)$ denotes the condition number of $A = \begin{bmatrix} \boldsymbol{a}_1 & \cdots & \boldsymbol{a}_k \end{bmatrix}$. Table 6.1 also includes the theoretical performance of the MGS algorithm [11] as "MGS" and the compact WY reorthogonalization algorithm in Algorithm 3.3 as "cWY", respectively.

As pointed out in [66], the CGS2 algorithm is not guaranteed to accomplish the reorthogonalization with the high accuracy if A does not have full numerical rank. Thus, the CGS2 algorithm may be less robust than the compact WY reorthogonalization, which has the robustness deriving from the Householder transformations. The same discussion is true for the OMP-CGS2 algorithm.

6.4 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.

6.4.1 Configurations of Numerical Experiments

In the numerical experiments, we compared the numerical results for computing the ℓ largest singular triplets of target matrices using four different codes of the GKLR algorithm, where $\ell \in \{100, 200, 400, 800\}$.

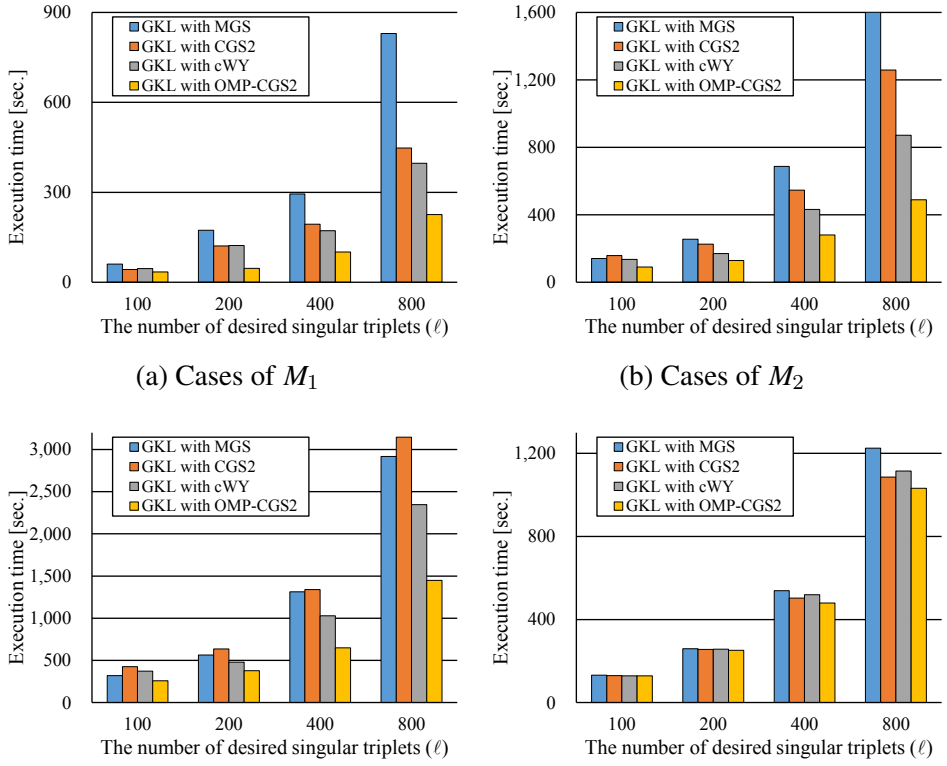
Each GKLR code is implemented with the following four reorthogonalization algorithms. **GKL with MGS** is implemented with the MGS algorithm. **GKL with CGS2** is implemented with the CGS2 algorithm shown in Algorithm 6.3. **GKL with cWY** is implemented with the compact WY reorthogonalization algorithm shown in Algorithm 3.3. 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 6.4.

In the experiments, we used four $m \times n$ real matrices M_1, M_2, M_3, and M_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. M_1, M_2, and M_3 are only different in both m and n from each other as follows: $m = 16,000$ and $n = 8,000$ for M_1. $m = 32,000$ and $n = 16,000$ for M_2. $m = 64,000$ and $n = 32,000$ for M_3. Note that the condition number is 4.803×10^1 for M_1, 4.754×10^1 for M_2, and 4.757×10^1 for M_3, respectively. The condition number of some matrices in actual applications is considered almost the same as that of these target matrices. In addition, we have to confirm that **GKL with OMP-CGS2** can compute a subset of singular triplets of target matrices at the same level of accuracy as the other GKLR codes even if the condition number of the target matrices is significantly large. For this purpose, the Frank matrix with $m = 32,000$, $n = 32,000$, and the condition number 1.600×10^9 was used as M_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 6.2. We used the Intel Math Kernel Library (MKL) [9] for parallelizing the BLAS routines. To control

Table 6.2: 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: 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`
Run Command	`numactl -localalloc`
Software	Intel Math Kernel Library 11.1.2



(a) Cases of M_1

(b) Cases of M_2

(c) Cases of M_3

(d) Cases of M_4

Figure 6.1: 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.

the memory allocation, all the experiments were run with `numactl -localalloc` command.

Table 6.3: The iteration number of the GKLR codes in each experiment.

ℓ: # of desired singular triplets	100	200	400	800
Matrix M_1	1,000	1,600	2,400	4,000
Matrix M_2	1,300	2,000	3,200	4,800
Matrix M_3	1,600	2,400	3,600	5,600
Matrix M_4	200	400	800	1,600

Table 6.4: $\|U_\ell^\top U_\ell - I\|_F/\sqrt{\ell}$: The orthogonality of the left singular vectors of the target matrix computed by four GKLR codes where the GKL algorithms with different reorthogonalization process are implemented.

(a) Cases of M_1

ℓ	100	200	400	800
GKL with MGS	1.40E-15	1.53E-15	1.81E-15	2.39E-15
GKL with CGS2	1.81E-15	2.08E-15	2.45E-15	3.05E-15
GKL with cWY	3.54E-15	3.92E-15	4.60E-15	5.41E-15
GKL with OMP-CGS2	1.75E-15	2.13E-15	2.38E-15	2.95E-15

(b) Cases of M_2

ℓ	100	200	400	800
GKL with MGS	1.41E-15	1.74E-15	2.10E-15	2.66E-15
GKL with CGS2	2.13E-15	2.31E-15	2.78E-15	3.43E-15
GKL with cWY	4.24E-15	4.74E-15	5.40E-15	6.33E-15
GKL with OMP-CGS2	1.98E-15	2.30E-15	2.84E-15	3.31E-15

(c) Cases of M_3

ℓ	100	200	400	800
GKL with MGS	1.89E-15	2.24E-15	2.50E-15	3.05E-15
GKL with CGS2	3.09E-15	2.97E-15	3.46E-15	3.95E-15
GKL with cWY	5.33E-15	5.90E-15	6.61E-15	7.57E-15
GKL with OMP-CGS2	2.89E-15	3.08E-15	3.60E-15	3.99E-15

(d) Cases of M_4

ℓ	100	200	400	800
GKL with MGS	8.09E-15	9.84E-15	2.19E-14	8.19E-14
GKL with CGS2	4.24E-15	4.68E-15	4.87E-15	4.94E-15
GKL with cWY	1.08E-14	1.12E-14	1.10E-14	1.11E-14
GKL with OMP-CGS2	4.42E-15	4.81E-15	4.88E-15	4.85E-15

Table 6.5: $\|V_\ell^\top V_\ell - I\|_F/\sqrt{\ell}$: The orthogonality of the right singular vectors of the target matrix computed by four GKLR codes where the GKL algorithms with different reorthogonalization process are implemented.

(a) Cases of M_1

ℓ	100	200	400	800
GKL with MGS	1.34E-15	1.56E-15	1.99E-15	2.35E-15
GKL with CGS2	1.47E-15	1.84E-15	2.15E-15	2.66E-15
GKL with cWY	2.59E-15	2.93E-15	3.67E-15	4.61E-15
GKL with OMP-CGS2	1.56E-15	1.85E-15	2.15E-15	2.59E-15

(b) Cases of M_2

ℓ	100	200	400	800
GKL with MGS	1.60E-15	1.64E-15	2.03E-15	2.61E-15
GKL with CGS2	1.74E-15	2.16E-15	2.54E-15	3.06E-15
GKL with cWY	3.38E-15	3.58E-15	4.25E-15	5.01E-15
GKL with OMP-CGS2	1.81E-15	2.20E-15	2.54E-15	2.92E-15

(c) Cases of M_3

ℓ	100	200	400	800
GKL with MGS	1.60E-15	1.82E-15	2.37E-15	2.97E-15
GKL with CGS2	1.98E-15	2.37E-15	2.84E-15	3.47E-15
GKL with cWY	3.95E-15	4.31E-15	4.94E-15	5.92E-15
GKL with OMP-CGS2	2.10E-15	2.59E-15	2.86E-15	3.47E-15

(d) Cases of M_4

ℓ	100	200	400	800
GKL with MGS	3.95E-14	3.97E-14	4.16E-14	4.40E-14
GKL with CGS2	4.39E-15	4.82E-15	5.00E-15	4.99E-15
GKL with cWY	1.08E-14	1.07E-14	1.07E-14	1.11E-14
GKL with OMP-CGS2	4.69E-15	4.36E-15	4.63E-15	4.66E-15

6.4.2 Experimental Results

Figure 6.1 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 6.1a, 6.1b, 6.1c, and 6.1d corresponds to the cases of target matrices M_1, M_2, M_3, and M_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 6.3. Note that the iteration number is not changed in Table 6.3 if we replace the reorthogonalization algorithm, for example, from the MGS

Table 6.6: 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 M_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 M_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 M_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 M_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

to the CGS2 algorithm.

Table 6.4 and 6.5 shows the orthogonality of the left and right singular vectors computed by each GKLR code, respectively. Table 6.4 shows $\|U_\ell^\top U_\ell - I\|_F/\sqrt{\ell}$, where $U_\ell = \begin{bmatrix} \boldsymbol{u}_1 & \cdots & \boldsymbol{u}_\ell \end{bmatrix}$ and each of $\boldsymbol{u}_j$ is the left singular vector computed by each code. Similarly, Tables 6.5 show $\|V_\ell^\top V_\ell - I\|_F/\sqrt{\ell}$, where $V_\ell = \begin{bmatrix} \boldsymbol{v}_1 & \cdots & \boldsymbol{v}_\ell \end{bmatrix}$ and each of $\boldsymbol{v}_j$ is the left singular vector computed by each code. From both of Tables 6.4 and 6.5, we can observe that **GKL with OMP-CGS2** achieves as high orthogonality as other GKLR codes.

In addition, Tables 6.6 shows the number of desired singular triplets and the execution time spending for the reorthogonalization process in computing the singular triplets of

the target matrices M_1, M_2, M_3, and M_4 using each GKLR code. The tables show that, in the case of M_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.

As shown above, we have confirmed that **GKL with OMP-CGS2** shows the better performance than the other GKLR codes. However, we should evaluate the performance of **GKL with OMP-CGS2** through further numerical experiments for large sparse matrices in actual applications since all the target matrices in the numerical experiments are artificial. This study is considered as one of future work.

6.5 Cache Utilization in OMP-CGS2

As mentioned in Section 6.3.2, 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.

6.5.1 Discussion

Recalling Algorithm 6.4, 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:

$$m \times (T \times 2 + 1) \times 8 \leq C \times 1024 \times 1024, \tag{6.14}$$

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 (6.14), the OMP-CGS2 algorithm is theoretically guaranteed to achieve higher performance than the CGS2 algorithm. Hereafter, let $m_{\mathrm{cache}}^{(\mathrm{theory})}$ be the maximum value

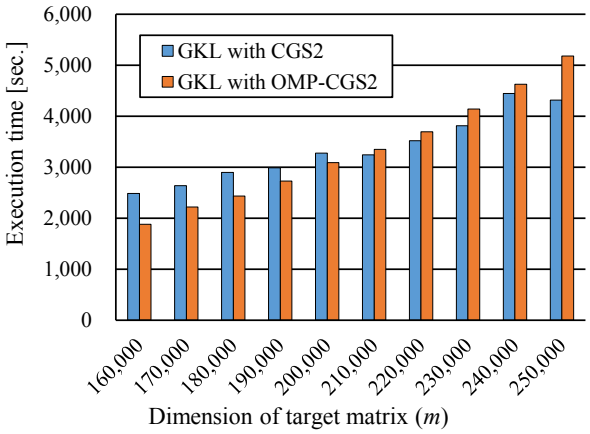


Figure 6.2: Comparison of the execution times for computing the desired singular triplets of the $m \times m$ target matrix using **GKL with CGS2** and **GKL with OMP-CGS2**.

of m at which inequality (6.14) 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 6.4 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}.

6.5.2 Verification

According to inequality (6.14), $m_{\text{cache}}^{(\text{theory})}$ for a machine in Table 6.2 is computed as follows: Since $T = 8$ and $C = 20$ from Table 6.2,

$$m_{\text{cache}}^{(\text{theory})} = 154202. \tag{6.15}$$

In fact, **GKL with OMP-CGS2** is faster than **GKL with CGS2** in all the experiments shown in Section 6.4 because both $m \leq m_{\text{cache}}^{(\text{theory})}$ and $n \leq m_{\text{cache}}^{(\text{theory})}$ are satisfied.

To examine a true value of m_{cache} for a machine shown in Table 6.2, the numerical results is shown as follows: Figure 6.2 shows the execution times for computing the 100 largest 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 Figure 6.2, m_{cache} for the experimental environment in this thesis is ranged from $200,000$ to $210,000$.

7 Conclusions

In this thesis, various parallel algorithms based on the bisection and inverse iteration algorithms for computing a subset of eigenpairs of symmetric matrices are discussed. Each chapter of this thesis is summarized as follows:

Chapter 2 gives an introduction of the bisection and inverse iteration algorithms for real symmetric tridiagonal matrices and discusses their parallel implementation. It is pointed out in Section 2.2.2 that a bottleneck of the conventional inverse iteration algorithm in parallel processing is the reorthogonalization process since this process is mainly composed of the BLAS 1 routines. In order to solve this problem in the computation of eigenvectors, the inverse iteration algorithm with the compact WY reorthogonalization and the BIR algorithm is proposed in Chapters 3 and 4, respectively.

Chapter 3 presents a new implementation of the inverse iteration algorithm with the compact WY reorthogonalization. Since the compact WY reorthogonalization is mainly composed of the BLAS 2 routines, it achieves higher performance in parallel processing than the MGS algorithm, which is used as the reorthogonalization process of the conventional inverse iteration algorithm. In addition, the BLAS-based implementations of the compact WY reorthogonalization are discussed in this chapter. Both the computational cost and the amounts of working memory for the compact WY reorthogonalization can be reduced by employing suitable BLAS routines. Numerical experiments on two shared-memory multi-core processors show that the inverse iteration algorithm with the compact WY reorthogonalization achieves a higher performance than the conventional inverse iteration algorithm.

Chapter 4 presents an implementation of the BIR algorithm, i.e., the block inverse iteration algorithm with reorthogonalization, in Algorithm 4.2, where a block parameter is added to the simultaneous inverse iteration algorithm. The BIR algorithm is mainly composed of the matrix multiplications owing to the introduction of a block parameter, and thus is expected to accelerate the computation of eigenvectors even on massively parallel computers. Numerical experiments on shared-memory multi-core processors shows that the BIR algorithm is faster than the simultaneous inverse iteration algorithm. In addition, a parallel symmetric tridiagonal eigensolver, which is a combination of the parallel bisection algorithm shown in Algorithm 2.2 and the BIR algorithm, is shown to achieve a highly scalable performance through the numerical experiments.

The BIR algorithm is more suitable for the parallel computation of eigenvectors than the inverse iteration algorithm with reorthogonalization since the BIR algorithm is mainly composed of the BLAS 3 routines. On the other hand, the BIR algorithm is sometimes less robust than the inverse iteration algorithm with reorthogonalization as discussed in Section 4.2.1. Thus, the inverse iteration algorithm with reorthogonalization should be used rather than the BIR algorithm if a robust computation of eigenvectors is required.

Chapter 5 focuses on the subset computation of eigenpairs of real symmetric band

matrices on the basis of bisection and inverse iteration algorithms. To achieve a high-performance subset computation, a parallel symmetric band eigensolver is proposed in Chapter 5. The proposed eigensolver includes the parallel implementation of Murata's bisection algorithm (Section 5.1) for computing the desired eigenvalues and the BIR algorithm in Algorithm 5.3 for computing the corresponding eigenvectors. Murata's bisection algorithm employs not only Martin-Wilkinson's Gaussian elimination but also the block LU factorization, and hence, its parallel implementation is faster than that of Gupta's bisection algorithm. Numerical experiments on shared-memory multi-core processors show that the BIR algorithm is much faster than the inverse iteration algorithm with reorthogonalization since the BIR algorithm is parallelized with lower communication cost than the other. In addition, the numerical experiments also shows that the proposed eigensolver is faster than the conventional solvers.

If the proposed symmetric band eigensolver is applied to the subset computation of real symmetric full matrices appearing in actual applications such as the kernel principal component analysis, the number of the desired eigenpairs may be fewer than the number of the processing elements. In such a situation, the multi-section method [67, 68] or the multi-section with multiple eigenvalues method [69] is expected to achieve a much higher performance than the parallel bisection algorithm in Section 5.1.3. Thus, the development of the multi-section algorithm based on Murata's bisection algorithm is considered as one of future work.

Chapter 6 focuses on the subset computation of singular triplets of real sparse matrices on the basis of the GKLR algorithm, which employs the bisection and inverse iteration algorithms for real symmetric tridiagonal matrices. To achieve a high-performance subset computation on shared-memory multi-core processors with large cache, the OMP-CGS2 algorithm is proposed in Algorithm 6.4 for the reorthogonalization process of the GKLR algorithm. Owing to high reusability of data, the OMP-CGS2 algorithm is expected to achieve the higher performance in parallel processing than the conventional reorthogonalization algorithms, which are parallelized by employing the parallel BLAS routines. The numerical experiments on shared-memory multi-core processors with large cache show that the OMP-CGS2 algorithm accelerated the GKLR algorithm more effectively for computing a subset of singular triplets of a sparse matrix than other reorthogonalization algorithms. In addition, the cache utilization in the OMP-CGS2 algorithm is discussed and the condition, under which the OMP-CGS2 algorithm achieves a higher performance than the CGS2 algorithm, is examined through both theoretical approach and numerical experiments.

The numerical experiments also shows that the OMP-CGS2 algorithm accelerates the parallel computation of the reorthogonalization process more effectively than the compact WY reorthogonalization does. However, the compact WY reorthogonalization is based on the Householder transformations and is guaranteed to perform a reorthogonalization with high orthogonality, as discussed in Section 3.1. On the other hand, the CGS2 algorithm, which is the origin of the OMP-CGS2 algorithm, is not always guaranteed to be as robust as the compact WY reorthogonalization as discussed in 6.3.3. Thus, the compact WY reorthogonalization should be used rather than the OMP-CGS2 algorithm if robust computation is required.

As mentioned earlier in Chapter 1, subset computations of eigenpairs of symmetric

matrices or singular triplets of sparse matrices are one of the most important computations used in many applications. The parallelization of the subset computations is crucial for reducing the overall execution time. Thus, this thesis discusses parallel solvers for the subset computations on the basis of bisection and inverse iteration algorithms and presents their high-performance implementations. The key approaches to accelerate the parallel solvers are to improve the cache hit ratio of each computation and reduce the communication cost in parallel processing. For this purpose, the proposed parallel solvers employ the higher-level BLAS routines, the BLAS 2 and BLAS 3, and are implemented with lower communication cost. It should be noted that these approaches are derived from discussions on mathematical structures of classical algorithms such as the bisection, inverse iteration, CGS, and GKLR algorithms. Therefore all the proposed parallel solvers are shown to have higher performances than the conventional solvers through the numerical experiments.

In this thesis, all the proposed implementations of the bisection and inverse iteration algorithms are evaluated on shared-memory multi-core processors. However, as mentioned in Chapter 1, the computation by the proposed algorithms should be performed on massively parallel systems with distributed memory if the scale of problems is large. Thus, the development of implementations of the proposed algorithms for such parallel systems and its evaluation are considered as future work. In particular, a real symmetric eigensolver, including the proposed symmetric band eigensolver in Chapter 5, is expected to achieve a better performance on massively parallel systems than the conventional parallel eigensolvers. This is because the proposed symmetric band eigensolver includes one of the breakthroughs: eliminating bottlenecks of the conventional parallel eigensolvers by avoiding the band reduction into the symmetric tridiagonal form, i.e., the second step of the two-step framework in [33, 34], and its corresponding back-transformation. In addition, to achieve a better performance in massively parallel systems, it may be considered that the BIR algorithm employs the communication-optimal algorithms for the QR factorization [70, 71]. Another future work is to develop the implementation of the proposed algorithms for accelerators such as Intel Xeon Phi Co-processor and GPUs. Relevant to this future work, the author has proposed in [A5] the GPU implementation of the inverse iteration algorithm with reorthogonalization for real symmetric tridiagonal matrices. The proposed GPU implementation has been shown to achieve a high-performance computation on a heterogeneous computer with a multi-core CPU and a single GPU. Moreover, as well as the case of the GKLR algorithm, the bisection and inverse iteration algorithms for real symmetric band matrices mentioned in Chapter 5 can be applied to the *block GKL algorithm with reorthogonalization (BGKLR)* [43, 72], which is a block version of the GKLR algorithm. Thus, the development of the BGKLR algorithm combined with the bisection and inverse iteration algorithms for real symmetric band matrices and its evaluation should be further studied. Needless to say, the adoption of the proposed algorithms to the computations appearing in actual applications is also a future work.

Acknowledgement

The author would like to express his sincere gratitude toward his supervisor Professor Yoshimasa Nakamura for his great guidance and encouragements. He also would like Professor Hiroshi Nakashima and Professor Ken Umeno for their helpful comments on earlier drafts of this thesis. He would like to acknowledge Program-Specific Associate Professor Kinji Kimura for his helpful advice and many extensive discussions. He also would like to acknowledge Professor Hidehiko Hasegawa from University of Tsukuba and Lecturer Masami Takata from Nara Women's University for their helpful comments and cooperation. He would like to thank the members and the alumni of Nakamura-Tsujimoto Laboratory at Graduate School of Informatics, Kyoto University, for many helpful discussions and comments. In particular, he would like to thank Doctor Hiroki Toyokawa, Mister Yuki Fujii, and Mister Hiroki Tanaka to their collaborative researches. Moreover, he would like to acknowledge Doctor Takeshi Osoekawa from Fujitsu Limited for an instructive and fruitful internship about data analysis. Furthermore, he acknowledges the financial support as Research Fellowship for Young Scientists from Japan Society for the Promotion of Science. Finally, he would like to express his gratitude to his parents for their encouragement and support.

Bibliography

[1] B. Schölkopf, A. Smola, and K. Müller, "Nonlinear component analysis as a kernel eigenvalue problem," *Neural Computation*, vol. 10, no. 5, pp. 1299–1319, 1998.

[2] G. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix," *SIAM J. Numer. Anal.*, vol. 2, no. 2, pp. 205–224, 1965.

[3] H. D. Simon and H. Zha, "Low-rank matrix approximation using the Lanczos bidiagonalization process with applications," *SIAM J. Sci. Comput.*, vol. 21, no. 6, pp. 2257–2274, 2000.

[4] K. Pearson, "On lines and planes of closest fit to systems of points in space," *Philosophical Magazine*, vol. 2, no. 11, pp. 559–572, 1901.

[5] N. Halko, P.-G. Martinsson, Y. Shkolnisky, and M. Tygert, "An algorithm for the principal component analysis of large data sets," *SIAM J Sci. Comput.*, vol. 33, no. 5, pp. 2580–2594, 2011.

[6] TOP500. [Online]. Available: http://www.top500.org/

[7] Netlib, "BLAS." [Online]. Available: http://www.netlib.org/blas/

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

[9] Intel Math Kernel Library. [Online]. Available: https://software.intel.com/en-us/intel-mkl/

[10] GotoBLAS2. [Online]. Available: https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2/

[11] G. H. Golub and C. F. van Loan, *Matrix Computations*, 3rd ed. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[12] H. Bowdler, R. Martin, C. Reinsch, and J. Wilkinson, "The QR and QL algorithms for symmetric matrices," *Numer. Math.*, vol. 11, no. 4, pp. 293–306, 1968.

[13] J. Cuppen, "A divide and conquer method for the symmetric tridiagonal eigenproblem," *Numer. Math.*, vol. 36, no. 2, pp. 177–195, 1980.

[14] M. Gu and S. C. Eisenstat, "A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem," *SIAM J. Matrix Anal. Appl.*, vol. 16, no. 1, pp. 172–191, 1995.

[15] W. Barth, R. Martin, and J. Wilkinson, "Calculation of the eigenvalues of a symmetric tridiagonal matrix by the method of bisection," *Numer. Math.*, vol. 9, no. 5, pp. 386–393, 1967.

[16] I. C. F. Ipsen, "Computing an eigenvector with inverse iteration," *SIAM Review*, vol. 39, no. 2, pp. 254–291, 1997.

[17] W. Kahan, "Accurate eigenvalues of a symmetric tridiagonal matrix," *Technical Report, Computer Science Dept. Stanford University*, no. CS41, 1966.

[18] I. S. Dhillon, "A new $O(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1997.

[19] I. S. Dhillon, B. N. Parlett, and C. Vömel, "The design and implementation of the MRRR algorithm," *ACM Trans. Math. Softw.*, vol. 32, no. 4, pp. 533–560, 2006.

[20] E. Anderson, Z. Bai, C. Bischof, L. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA, USA: SIAM, 1999.

[21] Netlib, "LAPACK." [Online]. Available: http://www.netlib.org/lapack/

[22] F. Tisseur and J. Dongarra, "A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures," *SIAM J. Sci. Comput.*, vol. 20, no. 6, pp. 2223–2236, 1999.

[23] M. Petschow and P. Bientinesi, "MR^3-SMP: A symmetric tridiagonal eigensolver for multi-core architectures," *Parallel Computing*, vol. 37, no. 12, 2011.

[24] M. Petschow, E. Peise, and P. Bientinesi, "High-performance solvers for dense hermitian eigenproblems," *SIAM J. Sci. Comput.*, vol. 35, no. 1, pp. C1–C22, 2013.

[25] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vömel, "Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers," *SIAM J. Sci. Comput.*, vol. 30, no. 3, pp. 1508–1526, 2008.

[26] I. S. Dhillon, B. N. Parlett, and C. Vömel, "Glued matrices and the MRRR algorithm," *SIAM J. Sci. Comput.*, vol. 27, no. 2, pp. 496–510, 2005.

[27] O. Marques, B. Parlett, and C. Vömel, "Computations of eigenpair subsets with the mrrr algorithm," *Numer. Linear Algebra Appl.*, vol. 13, no. 8, pp. 643–653, 2006.

[28] P. Willems and B. Lang, "A framework for the mr^3 algorithm: Theory and implementation," *SIAM J. Sci. Comput.*, vol. 35, no. 2, pp. A740–A766, 2013.

[29] ——, "Twisted factorizations and qd-type transformations for the MR^3 algorithm—new representations and analysis," *SIAM J. Matrix Anal. Appl.*, vol. 33, no. 2, pp. 523–553, 2012.

[30] M. Petschow, E. Quintana-OrtÃŋ, and P. Bientinesi, "Improved accuracy and parallelism for MRRR-based eigensolvers—a mixed precision approach," *SIAM J. Sci. Comput.*, vol. 36, no. 2, pp. C240–C263, 2014.

[31] J. Wilkinson, "Householder's method for symmetric matrices," *Numer. Math.*, vol. 4, no. 1, pp. 354–361, 1962.

[32] J. Dongarra, D. Sorensen, and S. Hammarling, "Block reduction of matrices to condensed forms for eigenvalue computations," *J. Comput. Appl. Math.*, vol. 27, no. 1-2, pp. 215–227, 1989.

[33] C. Bischof, X. Sun, and B. Lang, "Parallel tridiagonalization through two-step band reduction," in *Proceedings of the Scalable High-Performance Computing Conference*, pp. 23–27, IEEE, 1994.

[34] C. H. Bischof, B. Lang, and X. Sun, "A framework for symmetric band reduction," *ACM Trans. Math. Softw.*, vol. 26, no. 4, pp. 581–601, 2000.

[35] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. Willems, "Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations," *Parallel Computing*, vol. 37, no. 12, pp. 783–794, 2011.

[36] T. Auckenthaler, H.-J. Bungartz, T. Huckle, L. Krämer, B. Lang, and P. Willems, "Developing algorithms and software for the parallel solution of the symmetric eigenvalue problem," *Journal of Computational Science*, vol. 2, no. 3, pp. 272–278, 2011.

[37] G. Ballard, J. Demmel, and N. Knight, "Avoiding communication in successive band reduction," *ACM Trans. Parallel Comput.*, vol. 1, no. 2, pp. 11:1–11:37, 2015.

[38] L. Kaufman, "Band reduction algorithms revisited," *ACM Trans. Math. Softw.*, vol. 26, no. 4, pp. 551–567, 2000.

[39] T. Katagiri and S. Itoh, "A massively parallel dense symmetric eigensolver with communication splitting multicasting algorithm," in *High Performance Computing for Computational Science - VECPAR 2010*, ser. Lecture Notes in Computer Science, vol. 6449, pp. 139–150, Springer Berlin Heidelberg, 2011.

[40] K. K. Gupta, "Eigenproblem solution by a combined Sturm sequence and inverse iteration technique," *Int. J. num. Meth. Engng*, vol. 7, no. 1, pp. 17–42, 1973.

[41] T. Imamura, S. Yamada, and M. Machida, "Development of a high performance eigensolver on the peta-scale next generation supercomputer system," *Prog. Nuclear Science and Technology*, vol. 2, pp. 643–650, 2011.

[42] K. Murata, "Reexamination of the standard eigenvalue problem of the symmetric matrix. II The direct sturm inverse-iteration for the banded matrix (in Japanese)," *Research report of University of Library and Information Science*, vol. 5, no. 1, pp. 25–45, 1986.

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

[44] Y. Yamamoto and Y. Hirota, "A parallel algorithm for incremental orthogonalization based on the compact WY representation," *JSIAM Letters*, vol. 3, pp. 89–92, 2011.

[45] R. Schreiber and C. van Loan, "A storage-efficient *WY* representation for products of Householder transformations," *SIAM J. Sci. Stat. Comput.*, vol. 10, no. 1, pp. 53–57, 1989.

[46] F. Chatelin, *Eigenvalues of Matrices*. Philadelphia, PA, USA: SIAM, 2012.

[47] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart, "Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization," *Math. Comput.*, vol. 30, no. 136, pp. 772–795, 1976.

[48] OpenMP. [Online]. Available: http://openmp.org/wp/

[49] J. Wilkinson, "Calculation of the eigenvalues of a symmetric tridiagonal matrix by the method of bisection," *Numer. Math.*, vol. 4, no. 1, pp. 362–367, 1962.

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

[51] J. W. Demmel, I. S. Dhillon, and H. Ren, "On the correctness of parallel bisection in floating point," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-94-805, 1994.

[52] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA, USA: SIAM, 1997.

[53] G. Peters and J. Wilkinson, "The calculation of specified eigenvectors by inverse iteration," in *Linear Algebra*, ser. Handbook for Automatic Computation, F. Bauer, Ed., vol. 2, pp. 418–439, Springer Berlin Heidelberg, 1971.

[54] Å. Björck, "Solving linear least squares problems by Gram-Schmidt orthogonalization," *BIT*, vol. 7, no. 1, pp. 1–21.

[55] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*. Oxford, UK: Oxford University Press, 1965.

[56] I. S. Dhillon and B. N. Parlett, "Orthogonal eigenvectors and relative gaps," *SIAM J. Matrix Anal. Appl.*, vol. 25, no. 3, pp. 858–899, 2003.

[57] T. Yokozawa, D. Takahashi, T. Boku, and M. Sato, "Parallel implementation of a recursive blocked algorithm for classical Gram-Schmidt orthogonalization," *Proc. 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA 2008)*, 2008.

[58] G. Stewart, "Block Gram-Schmidt orthogonalization," *SIAM J. Sci. Comput.*, vol. 31, no. 1, pp. 761–775, 2008.

[59] J. L. Barlow and A. Smoktunowicz, "Reorthogonalized block classical Gram-Schmidt," *Numer. Math.*, vol. 123, no. 3, pp. 1–29, 2012.

[60] J. W. Demmel, *Applied Numerical Linear Algebra*. Philadelphia, PA, USA: SIAM, 1997.

[61] R. Martin and J. Wilkinson, "Solution of symmetric and unsymmetric band equations and the calculation of eigenvectors of band matrices," *Numer. Math.*, vol. 9, no. 4, pp. 279–301, 1967.

[62] H. Hasegawa, "Symmetric band eigenvalue solvers for vector computers and conventional computers (in Japanese)," *J. Inf. Process.*, vol. 30, no. 3, pp. 261–268, 1989.

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

[64] Y. Saad, "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.

[65] T. Katagiri, "Performance evaluation of parallel Gram-Schmidt reorthogonalization methods," in *High Performance Computing for Computational Science - VECPAR 2002*, ser. Lecture Notes in Computer Science, vol. 2565, pp. 302–314, Springer Berlin Heidelberg, 2003.

[66] L. Giraud, J. Langou, M. Rozloẑnìk, and J. van den Eshof, "Rounding error analysis of the classical Gram-Schmidt orthogonalization process," *Numer. Math.*, vol. 101, no. 1, pp. 87–100, 2005.

[67] S. Lo, B. Philippe, and A. Sameh, "A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem," *SIAM J. Sci. Stat. Comput.*, vol. 8, no. 2, pp. s155–s165, 1987.

[68] H. Simon, "Bisection is not optimal on vector processors," *SIAM J. Sci. Stat. Comput.*, vol. 10, no. 1, pp. 205–209, 1989.

[69] T. Katagiri, C. Vömel, and J. W. Demmel, "Automatic performance tuning for the multi-section with multiple eigenvalues method for symmetric tridiagonal eigenproblems," in *Applied Parallel Computing. State of the Art in Scientific Computing*, ser. Lecture Notes in Computer Science, vol. 4699, pp. 938–948, Springer Berlin Heidelberg, 2007.

[70] J. W. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM J. Sci. Comput.*, vol. 34, no. 1, pp. A206–A239, 2012.

[71] T. Fukaya, Y. Nakatsukasa, Y. Yanagisawa, and Y. Yamamoto, "CholeskyQR2: A simple and communication-avoiding algorithm for computing a tall-skinny QR factorization on a large-scale parallel system," in *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '14, pp. 31–38, Piscataway, NJ, USA: IEEE, 2014.

[72] G. H. Golub, F. T. Luk, and M. L. Overton, "A block Lanczos method for computing the singular values and corresponding singular vectors of a matrix," *ACM Trans. Math. Softw.*, vol. 7, no. 2, pp. 149–169, 1981.

List of Author's Papers and Works Related to the Thesis

Original Papers

[A1] H. Ishigami, K. Kimura, and Y. Nakamura, "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, 2011. (A part of Chapter 3)

[A2] H. Ishigami, K. Kimura, and Y. Nakamura, "Reorthogonalized block inverse iteration algorithm for parallel computation of eigenvectors (in Japanese)," *IPSJ Transactions on Advanced Computing Systems*, vol. 7, no. 3, pp. 1–12, 2014.

[A3] M, Takata, H. Ishigami, K. Kimura, Y. Fujii, H. Tanaka, and Y. Nakamura, "Performance evaluation of Golub-Kahan-Lanczos algorithm with reorthogonalization by classical Gram-Schmidt algorithm and OpenMP," *IPSJ Transactions on Mathematical Modeling and Its Applications*, accepted. (A part of Chapter 6)

[A4] H. Ishigami, H. Hasegawa, K. Kimura, and Y. Nakamura, "A parallel bisection and inverse iteration solver for a subset of eigenpairs of symmetric band matrices," submitted. (A part of Chapter 5)

Peer-Reviewed Proceedings of International Conferences

[A5] H. Ishigami, K. Kimura, and Y. Nakamura, "GPU implementation of inverse iteration algorithm for computing eigenvectors," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pp. 664–671, 2014.

[A6] H. Ishigami, K. Kimura, and Y. Nakamura, "A new parallel symmetric tridiagonal eigensolver based on bisection and inverse iteration algorithms for shared-memory multi-core processors," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2015 Tenth International Conference on*, pp. 216–223, IEEE, 2015. (A part of Chapter 4)
 doi:10.1109/3PGCIC.2015.26