CCF Transactions on High Performance Computing (2023) 5:171–190
https://doi.org/10.1007/s42514-023-00135-1

REGULAR PAPER

GCGE: a package for solving large scale eigenvalue problems by parallel block damping inverse power method

Yu Li[1] · Zijing Wang[2,3] · Hehu Xie[2,3]

Received: 8 August 2022 / Accepted: 5 January 2023 / Published online: 7 February 2023
© China Computer Federation (CCF) 2023

Abstract

In this paper, we introduce some strategies to improve the efficiency and scalability of the generalized conjugate gradient algorithm and build a package GCGE for solving large scale eigenvalue problems. This method is the combination of damping idea, subspace projection method and inverse power algorithm with dynamic shifts. To reduce the dimensions of projection subspaces, a moving mechanism is developed when the number of desired eigenpairs is large. The numerical methods, implementing techniques and the structure of the package are presented. Plenty of numerical results are provided to demonstrate the efficiency, stability and scalability of the concerned eigensolver and the package GCGE for computing many eigenpairs of large symmetric matrices arising from applications.

Keywords Large scale eigenvalue problem · Block damping inverse power method · GCGE · Efficiency · Stability · Scalability

Mathematics Subject Classification 65N30 · 65N25 · 65L15 · 65B99

1 Introduction

A fundamental and challenging task in modern science and engineering is to solve large scale eigenvalue problems. Although high-dimensional eigenvalue problems are ubiquitous in physical sciences, data and imaging sciences, and machine learning, there is no so many classes of eigensolvers as that of linear solvers. Compared with linear equations, there are less efficient numerical methods for solving large scale eigenvalue problems, which poses significant challenges for scientific computing (Bai et al. 2000). Along with the development of science and engineer, the eigenvalue problems from complicated systems bring strong demand for eigensolvers with good efficiency, stability and scalability.

The Arnoldi and Lanczos methods which are based on the Krylov subspace are always used to design the eigensolvers (Saad 1992). In order to use explicitly and implicitly restarted techniques for generalized eigenvalue problems, it is necessary to solve the included linear equations exactly to produce upper Hessenberg matrices. But this requirement is always very difficult for large scale sparse matrices with poor conditions. Based on this consideration, locally optimal block preconditioned conjugate gradient (LOBPCG) is designed based on some types of iteration processes which

This research is supported partly by National Key R &D Program of China 2019YFA0709600, 2019YFA0709601, Science Challenge Project (No. TZ2016002), the National Center for Mathematics and Interdisciplinary Science, CAS, and Tianjin Education Commission Scientific Research Plan (2017KJ236).

✉ Hehu Xie
hhxie@lsec.cc.ac.cn

Yu Li
liyu@tjufe.edu.cn

Zijing Wang
zjwang@lsec.cc.ac.cn

[1] Coordinated Innovation Center for Computable Modeling in Management Science, Tianjin University of Finance and Economics, Tianjin 300222, China

[2] LSEC, ICMSEC, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing 100190, China

[3] School of Mathematical Sciences, University of Chinese Academy of Sciences, Beijing 100049, China

do not need to solve the included linear equations exactly (Knyazev and Neymeyr 2003; Knyazev 2006; Hetmaniuk and Lehoucq 2006; Knyazev et al. 2007; Duersch et al. 2018). This property makes LOBPCG be a reasonable candidate for solving large scale eigenvalue problems on parallel computers. But the subspace generating method and orthogonalization way lead to the unstability of LOBPCG algorithm (Li et al. 2020; Ning Zhang et al. 2020).

The appearance of high performance computers brings more possibilities for computing plenty of eigenpairs of large scale matrices. However, it is harder to design the eigensolver for large scale eigenvalue problems which has the same efficiency and scalability as the linearsolver for large scale linear equations. Solving eigenvalue problems on high performance computers needs new considerations such as the stability and scalability of orthogonalization for plenty of vectors, efficiency and memory costing for computing Rayleigh-Ritz problems. The first aim of this paper is to introduce new strategies to improve the efficiency and scalability of generalized conjugate gradient (GCG) method which is proposed in Li et al. (2020), Ning Zhang et al. (2020). These strategies include new efficient implementing techniques for orthogonalization and computing Rayleigh-Ritz problems. A recursive orthogonalization method with singular value decomposition (SVD) is proposed to improve scalability. In addition, we also provide a moving mechanism to reduce the dimensions of projection subspaces when solving Rayleigh-Ritz problems. The second aim is to build the package GCGE for solving large scale eigenvalue problems. The package Generalized Conjugate Gradient Eigensolver (GCGE) is written by C language and constructed with the way of matrix-free and vector-free. The source code can be downloaded from GitHub with the address https://github.com/Materials-Of-Numerical-Algebra/GCGE.

The rest of the paper is organized as follows. In Sect. 2, we present the concerned algorithm for eigenvalue problems. The implementing techniques are designed in Sect. 3. The package GCGE will be introduced in Sect. 4. In Sect. 5, plenty of numerical tests are provided to demonstrate the efficiency, stability and scalability of the proposed algorithm and the associated package. Concluding remarks are given in the last section.

2 GCG algorithm

For simplicity, in this paper, we are concerned with the following generalized algebraic eigenvalue problem: Find eigenvalue $\lambda \in \mathbb{R}$ and eigenvector $x \in \mathbb{R}^N$ such that

$$Ax = \lambda Bx, \tag{1}$$

where A is a $N \times N$ real symmetric matrix and B is a $N \times N$ real symmetric positive definite (SPD) matrix.

The GCG algorithm is a type of subspace projection method, which uses the damping block inverse power idea to generate triple blocks $[X, P, W]$, where X saves the current eigenvector approximation, P saves the information from previous iteration step, and W saves vectors from the inverse power iteration with some conjugate gradient (CG) steps to X. We name this method as generalized conjugate gradient algorithm since the structure of triple blocks $[X, P, W]$ is similar to that of conjugate gradient method for linear equations. The GCG algorithm is defined by Algorithm 1, where `numEigen` stands for the number of desired eigenpairs.

Algorithm 1 GCG algorithm

1. Choose `numEigen` vectors to build the block X and two null blocks $P = [\,]$, $W = [\,]$.
2. Define $V = [X, P, W]$ and do orthogonalization to V in the sense of inner product deduced by the matrix B.
3. Solve the Rayleigh-Ritz problem $(V^\top AV)\widehat{x} = \widehat{x}\Lambda_x$ to obtain Λ_x and $\widehat{x}$, then get new approximate eigenvectors $X^{\mathtt{new}} = V\widehat{x}$.
4. Check the convergence of eigenpair approximations $(\Lambda_x, X^{\mathtt{new}})$. If the smallest `numEigen` eigenpairs converge, the iteration will stop.
5. Otherwise, compute $P = X^{\mathtt{new}} - X(X^\top BX^{\mathtt{new}})$ and update $X = X^{\mathtt{new}}$.
6. Generate W by solving linear equations $(A - \theta B)W = BX(\Lambda_x - \theta I)$ by some CG steps with the initial guess X, where the shift θ is selected dynamically.
7. Then go to **STEP 2**.

The main difference of Algorithm 1 from LOBPCG is the way to generate W and orthogonalization to V. The GCG algorithm uses the inverse power method with dynamic shifts to generate W. Meanwhile, the full orthogonalization to V is implemented in order to guarantee the numerical stability. In addition, a new type of recursive orthogonalization method with SVD will be designed in the next section.

In **Step 3** of Algorithm 1, solving Rayleigh-Ritz problem is a sequential process which can not be accelerated by using normal parallel computing. Furthermore, it is well known that the computing time is superlinearly dependent on the number of desired eigenpairs (Saad 1992). Then in order to accelerate this part, reducing the dimensions of Rayleigh-Ritz problems is a reasonable way. We will compute the desired eigenpairs in batches when the number of desired eigenpairs is large. In each iteration step, the dimensions of P and W are set to be `numEigen/5` or `numEigen/10`. Moreover, a moving mechanism is presented for computing a large number of desired eigenpairs. These two strategies can further reduce not only the time proportion of the sequential process for solving Rayleigh-Ritz problems but also the amount of memory required by **STEP 3**. In addition, the Rayleigh-Ritz problem is distributed to multi computing processes and each process only computes a small part of desired eigenpairs. In other words, the Rayleigh-Ritz problem is solved in parallel. More details of implementing techniques will be introduced in Sects. 3.2–3.3 or refer to Li et al. (2020), Ning Zhang et al. (2020).

In **STEP 6** of Algorithm 1, though the matrix $A - \theta B$ may not be SPD, the CG iteration method is adopted for solving the included linear equations due to the warm start X and the shift θ. Please refer to Sect. 3.2 for more details. Furthermore, it is suggested to use the algebraic multigrid method as the preconditioner for **STEP 6** of Algorithm 1 with the shift $\theta = 0.0$, when the concerned matrices are sparse and come from the discretization of partial differential operators by finite element, finite difference or finite volume, etc.

3 Implementing techniques

In this section, we introduce implementing techniques to improve efficiency, scalability and stability for the concerned eigensolver in this paper. Based on the discussion in Sect. 2, we focus on the methods for doing the orthogonalization and computing Rayleigh-Ritz problems. A recursive orthogonalization method with SVD and a moving mechanism are presented. In addition, the package GCGE is introduced, which is written by C language and constructed with the way of matrix-free and vector-free.

3.1 Improvements for orthogonalization

This subsection is devoted to introducing the orthogonalization methods which have been supported by GCGE. So far, we have provided modified block orthogonalization method and recursive orthogonalization method with SVD. The criterion for choosing the orthogonalization methods should be based on the number of desired eigenpairs and the scales of the concerned matrices. The aim is to keep the balance among efficiency, stability and scalability.

The modified Gram-Schmidt method (Stewart 2008) is designed to improve the stability of classical orthogonalization method. The modified block orthogonalization method is the block version of modified Gram-Schmidt method, which can be defined by Algorithm 3. They have the same accuracy and stability, but the modified block orthogonalization method has better efficiency and scalability.

Let us consider the orthogonalization for $X \in \mathbb{R}^{N\times m}$ and assume $m = b\ell$ in Algorithm 2. We divide X into ℓ blocks, i.e., $X = [X_1, X_2, \ldots, X_\ell]$, where $X_i \in \mathbb{R}^{N\times b}, i = 1, \ldots, \ell$. The orthogonalization process is to make X be orthogonal to X_0 and do orthogonalization for X itself, where $X_0 \in \mathbb{R}^{N\times m_0}$ has already been orthogonalized, i.e., $X_0^\top B X_0 = I$.

Firstly, in order to maintain the numerical stability, the process of deflating components in X_0 from X is repeated until the maximum absolute value of elements in $X_0^\top BX$ is small enough. Secondly, the columns of X in blocks of ℓ columns are orthogonalized through the modified Gram-Schmidt method. For each $k = 1, \ldots, \ell$ in Algorithm 2, when X_k is linear dependent, the rearmost vectors of X are copied to the corresponding location. In addition, there are $b + 1$ global communications in each `for` circle on the 4-th line of Algorithm 2. In other words, the total number of global communications is

$$(b + 1)(\ell - 1) + b = m + m/b - 1.$$

In fact, in modified block orthogonalization method, we deflate the components in previous orthogonalized vectors successively for all unorthogonalized vectors in each iteration step. This means Algorithm 2 uses block treatment for the unorthogonalized vectors to improve efficiency and scalability without loss of stability. As default, b is set to be $\min(m/4, 200)$.

Algorithm 2 Modified block orthogonalization

1: **repeat**
2: Compute $X = X - X_0(X_0^\top(BX))$;
3: **until** the norm of $X_0^\top(BX)$ is small enough;
4: **for** $k = 1 : \ell$ **do**
5: Orthogonalize X_k by modified Gram-Schmidt method;
6: **if** $k = \ell$ **then**
7: break;
8: **end if**
9: **repeat**
10: Compute $\begin{bmatrix} R_{k+1} \\ \vdots \\ R_\ell \end{bmatrix} = [X_{k+1}, \cdots, X_\ell]^\top(BX_k)$;
11: Compute $[X_{k+1}, \cdots, X_\ell] = [X_{k+1}, \cdots, X_\ell] - X_k \begin{bmatrix} R_{k+1} \\ \vdots \\ R_\ell \end{bmatrix}^\top$;
12: **until** the maximum absolute value of elements in $R_{k+1}, \cdots, R_\ell$ is small enough;
13: **end for**

In order to improve efficiency and scalability further, we design a type of recursive orthogonalization method with SVD and the corresponding scheme is defined by Algorithm 3. The aim here is to make full use of level-3 BLAS operations. We also find the paper (Yokozawa et al. 2006) has discussed the similar orthogonalization method without SVD. The contribution here is to combine the recursive orthogonalization method and SVD to improve the scalability.

Algorithm 3 $\texttt{RecursiveOrthSVD}(\mathrm{X}, \texttt{s}, \texttt{e})$

1: Compute $\texttt{length} = \texttt{e} - \texttt{s} + 1$;
2: **if** $\texttt{length} \leq c$ **then**
3: **repeat**
4: Compute $M = X(:, \texttt{s} : \texttt{e})^\top BX(:, \texttt{s} : \texttt{e})$;
5: Compute SVD of $M = Q\Lambda Q^\top$;
6: Compute $X(:, \texttt{s} : \texttt{e}) = X(:, \texttt{s} : \texttt{e})Q\Lambda^{-1/2}$;
7: **until** the norm of $\Lambda - I$ is small enough;
8: **else**
9: $\texttt{s1} = \texttt{s}$; $\texttt{e1} = \texttt{s} + \texttt{length}/2 - 1$;
10: $\texttt{s2} = \texttt{e1} + 1$; $\texttt{e2} = \texttt{e}$;
11: Call $\texttt{RecursiveOrthSVD}(X, \texttt{s1}, \texttt{e1})$;
12: **repeat**
13: Compute $R = (BX(:, \texttt{s1} : \texttt{e1}))^\top X(:, \texttt{s2} : \texttt{e2})$;
14: Compute $X(:, \texttt{s2} : \texttt{e2}) = X(:, \texttt{s2} : \texttt{e2}) - X(:, \texttt{s1} : \texttt{e1})R$;
15: **until** the maximum absolute value of elements in R is small enough;
16: Call $\texttt{RecursiveOrthSVD}(X, \texttt{s2}, \texttt{e2})$;
17: **end if**

Let us consider $X \in \mathbb{R}^{N\times m}$ and $m = 2^\eta$ in Algorithm 3. The orthogonalization of X is completed by calling `RecursiveOrthSVD` recursively. We use `X(: ,s : e)` to stand for the `s`-th column to `e`-th column of X. When `length` $\le c$, SVD is applied to do the orthogonalization to X, where c is set to be $\min(m, 16)$ as default. In order to maintain the numerical stability, the orthogonalization to X with SVD is repeated until the matrix Λ is close enough to the identity matrix. Always, the above condition is satisfied after two or three iterations. If M has eigenvalues close to zero, i.e., the columns of X_k are linearly dependent, the subsequent vectors will be copied to the corresponding location.

If $c = 16$ and we do the orthogonalization to X with SVD three times when `length` $\le c$, the total number of global communications is

$$2^0 + 2^1 + 2^2 + \cdots + 2^{\eta-5} + 3 \times 2^{\eta-4} = \frac{1}{4}m - 1,$$

which is much smaller than that of Algorithm 2.

The recursive orthogonalization method with SVD is recommended and it is the default choice in our package for the orthogonalization. In fact, Algorithms 2 and 3 can both reach the required accuracy for all numerical examples in this paper. In the case of solving generalized eigenvalue problems, B-orthogonalization should be considered. Algorithm 3 is more efficient than Algorithm 2 in most cases, which will be shown in Sect. 5.5.

3.2 Computation reduction for Algorithm 1

In this subsection, let us continue considering the computation reduction for Algorithm 1, i.e., design efficient ways to compute the Rayleigh-Ritz problem in **STEP 3** which include

- Orthogonalizing $V = [X, P, W]$;
- Computing the small scale matrix $\bar{A} = V^\top AV$;
- Solving the standard eigenvalue problem $\bar{A}\hat{x} = \hat{x}\Lambda_x$.

Except for the moving mechanism shown in Sect. 3.3 and the inverse power method with dynamic shifts for solving W, the techniques here are almost the same as those in Li et al. (2020), Ning Zhang et al. (2020). But for easier understanding and completeness, we also introduce them here using more concise expressions. In conclusion, the following main optimization techniques are implemented:

(1) The converged eigenpairs do not participate the subsequent iteration;
(2) The sizes of P and W are set to be `blockSize`, which is equal to `numEigen/5` as default;
(3) The shift is selected dynamically when solving W;
(4) The large scale orthogonalization to V is transformed into the small scale orthogonalization to P and a large scale orthogonalization to W;
(5) The submatrix of $\bar{A}$ corresponding to X can be obtained by Λ_x;
(6) The submatrix of $\bar{A}$ corresponding to P can be computed by multiplication of small scale dense matrices;
(7) The Rayleigh-Ritz problem $\bar{A}\hat{x} = \hat{x}\Lambda_x$ is solved in parallel;
(8) The moving mechanism is presented to reduce the dimension of $\bar{A}$ further.

According to **STEP 2** of Algorithm 1, we decompose X into three parts

$$X = \left[X_c,\ X_n,\ X_{\tilde{n}}\right],$$

where X_c denotes the converged eigenvectors and $[X_n, X_{\tilde{n}}]$ denotes the unconverged ones. The number of vectors in X_n is `blockSize`. Based on the structure of X, the block version has the following structure

$$V = \left[X_c,\ X_n,\ X_{\tilde{n}},\ P,\ W\right]$$

with $V^\top BV = I$. And the eigenpairs Λ_x and $\hat{x}$ can be decomposed into the following form

$$\Lambda_x = \begin{bmatrix} \Lambda_c & O & O \\ O & \Lambda_n & O \\ O & O & \Lambda_{\tilde{n}} \end{bmatrix},\ \hat{x} = \left[\hat{x}_c,\ \hat{x}_n,\ \hat{x}_{\tilde{n}}\right], \tag{2}$$

where Λ_x is the diagonal matrix.

Then in **STEP 3** of Algorithm 1, the small scale eigenvalue problem

$$\bar{A}\hat{x} = \hat{x}\Lambda_x$$

has the following form

$$\bar{A}\begin{bmatrix} I & O & O \\ O & \hat{x}_{nn} & \hat{x}_{n\tilde{n}} \\ O & \hat{x}_{\tilde{n}n} & \hat{x}_{\tilde{n}\tilde{n}} \\ O & \hat{x}_{pn} & \hat{x}_{p\tilde{n}} \\ O & \hat{x}_{wn} & \hat{x}_{w\tilde{n}} \end{bmatrix} = \begin{bmatrix} I & O & O \\ O & \hat{x}_{nn} & \hat{x}_{n\tilde{n}} \\ O & \hat{x}_{\tilde{n}n} & \hat{x}_{\tilde{n}\tilde{n}} \\ O & \hat{x}_{pn} & \hat{x}_{p\tilde{n}} \\ O & \hat{x}_{wn} & \hat{x}_{w\tilde{n}} \end{bmatrix}\begin{bmatrix} \Lambda_c & O & O \\ O & \Lambda_n & O \\ O & O & \Lambda_{\tilde{n}} \end{bmatrix}, \tag{3}$$

where $\bar{A} = V^\top AV$, $\hat{x}^\top\hat{x} = I$ and $\hat{x}_c$, $\hat{x}_n$, $\hat{x}_{\tilde{n}}$ have following structures

$$\hat{x}_c = \begin{bmatrix} I \\ O \\ O \\ O \\ O \end{bmatrix},\ \hat{x}_n = \begin{bmatrix} O \\ \hat{x}_{nn} \\ \hat{x}_{\tilde{n}n} \\ \hat{x}_{pn} \\ \hat{x}_{wn} \end{bmatrix},\ \hat{x}_{\tilde{n}} = \begin{bmatrix} O \\ \hat{x}_{n\tilde{n}} \\ \hat{x}_{\tilde{n}\tilde{n}} \\ \hat{x}_{p\tilde{n}} \\ \hat{x}_{w\tilde{n}} \end{bmatrix}. \tag{4}$$

In addition, Ritz vectors are updated as

$$X^{\mathrm{new}} = V\widehat{x}.$$

In **STEP 4** of Algorithm 1, the convergence of the eigenpairs $(\Lambda_x, X^{\mathrm{new}})$ is checked. Due to (2), we set

$$\widehat{x}_n = \left[\widehat{x}_{n_1}, \widehat{x}_{n_2}\right], \ \widehat{x}_{\widetilde{n}} = \left[\widehat{x}_{\widetilde{n}_1}, \widehat{x}_{\widetilde{n}_2}\right],$$
$$\Lambda_n = \begin{bmatrix} \Lambda_{n_1} & O \\ O & \Lambda_{n_2} \end{bmatrix}, \ \Lambda_{\widetilde{n}} = \begin{bmatrix} \Lambda_{\widetilde{n}_1} & O \\ O & \Lambda_{\widetilde{n}_2} \end{bmatrix},$$

and the diagonal of Λ_{n_1} inclues the new converged eigenvalues. Then all ℓ convergened eigenvectors are in

$$X_c^{\mathrm{new}} = V\left[\widehat{x}_c, \widehat{x}_{n_1}\right],$$

and the unconverged ones are in

$$X_n^{\mathrm{new}} = V\left[\widehat{x}_{n_2}, \widehat{x}_{\widetilde{n}_1}\right] \text{ and } X_{\widetilde{n}}^{\mathrm{new}} = V\widehat{x}_{\widetilde{n}_2}.$$

If ℓ is equal to `numEigen`, the iteration will stop. Otherwise, $0 \le \ell <$ `numEigen` and the iteration will continue. Here, the length of X_n^{new} is

$$\texttt{blockSize} = \min(\texttt{numEigen}/5, \texttt{numEigen} - \ell).$$

In **STEP 5** of Algorithm 1, in order to produce P for the next GCG iteration, from the definition of $\widehat{x}_n$ in (4) and the orthonormality of V, i.e., $V^\top BV = I$, we first set

$$\widetilde{P} = V\widehat{x}_n - X_n(X_n^\top BV\widehat{x}_n) = V\widetilde{p},$$

where

$$\widetilde{p} = \begin{bmatrix} O \\ O \\ \widehat{x}_{\widetilde{n}n} \\ \widehat{x}_{pn} \\ \widehat{x}_{wn} \end{bmatrix}. \tag{5}$$

In order to compute P^{new} to satisfy $(X^{\mathrm{new}})^\top BP^{\mathrm{new}} = O$, we come to do the orthogonalization for small scale vectors in $[\widehat{x}, \widetilde{p}]$ according to the L^2 inner product. Since vectors in $\widehat{x}$ are already orthonormal, the orthogonalization only needs to be done for $\widetilde{p}$ against $\widehat{x}$ to get a new vectors $\widehat{p}$. Thus let $[\widehat{x}, \widehat{p}]$ denote the orthogonalized block, i.e.,

$$\left[\widehat{x}, \widehat{p}\right]^\top \left[\widehat{x}, \widehat{p}\right] = I. \tag{6}$$

Then,

$$P^{\mathrm{new}} = V\widehat{p}.$$

Moreover, it is easy to check that

$$(X^{\mathrm{new}})^\top BP^{\mathrm{new}} = \widehat{x}^\top V^\top BV\widehat{p} = O$$

and

$$(P^{\mathrm{new}})^\top BP^{\mathrm{new}} = \widehat{p}^\top V^\top BV\widehat{p} = I.$$

In **STEP 6** of Algorithm 1, $\widetilde{W}$ is obtained by some CG iterations for the linear equations

$$(A - \theta B)\widetilde{W} = BX_n^{\mathrm{new}} \begin{bmatrix} \Lambda_{n_2} - \theta I & O \\ O & \Lambda_{\widetilde{n}_1} - \theta I \end{bmatrix} \tag{7}$$

with the initial guess X_n^{new}, where the shift θ is related to the largest converged eigenvalue in the convergence process. It is noted that the shift is not fixed and the matrix $A - \theta B$ may not be SPD, but the initial guess X_n^{new} is perpendicular to the eigenvectors of $A - \theta B$ corresponding to all negative eigenvalues, i.e.,

$$(X_n^{\mathrm{new}})^\top (A - \theta B)X_c^{\mathrm{new}} = O,$$

since X_c^{new} reaches the convergence criterion. In other words, $A - \theta B$ is SPD in the orthogonal complement space of span (X_c^{new}). Then the CG iteration method can be adopted for solving the included linear equations. Due to the shift θ, the multiplication of matrix and vector of each CG iteration takes more time, but the convergence of GCG algorithm is accelerated. In addition, there is no need to solve linear equations (7) with high accuracy, and only 10-30 CG iterations are enough during each GCG iteration. In Remark 3.1, an example is presented to explain why the convergence of GCG algorithm with dynamic shifts is accelerated after one CG iteration. In Sect. 5.1, we give some numerical results to show the performance of GCGE with dynamic shifts and the convergence procedure under different number of CG iterations. In order to produce W^{new} by Algorithm 3 for the next GCG iteration, we need to do the orthogonalization to $\widetilde{W}$ according to $[X^{\mathrm{new}}, P^{\mathrm{new}}]$, i.e.,

$$\left[X^{\mathrm{new}}, P^{\mathrm{new}}\right]^\top BW^{\mathrm{new}} = O, \ (W^{\mathrm{new}})^\top BW^{\mathrm{new}} = I.$$

Remark 3.1 We give an example to present the accelerating convergence of GCG algorithm with dynamic shifts after one CG iteration. Assuming that the first eigenpair (λ_1, v_1) has been found for the standard eigenvalue problem

$$Ax = \lambda x,$$

we have the approximate eigenvector $x_0 = a_2v_2 + a_3v_3$ to the second eigenvector, where

$$Av_2 = \lambda_2 v_2, \ Av_3 = \lambda_3 v_3, \text{ and } 0 < \lambda_1 < \lambda_2 \le \lambda_3.$$

Since the linear equations

$$(A - \theta I)w = (\tilde{\lambda} - \theta)x_0 \text{ and } 0 \le \theta < \lambda_2,$$

we can obtain the new approximate eigenvector

$$x_1 = \frac{a_2^2 + a_3^2}{a_3^2(\lambda_2 - \theta) + a_2^2(\lambda_3 - \theta)}\Big((\lambda_3 - \theta)a_2 v_2 + (\lambda_2 - \theta)a_3 v_3\Big),$$

after the first CG iteration with the initial guess x_0, where $\tilde{\lambda} = x_0^\top A x_0 / x_0^\top x_0$. It is noted that the convergence rate is

$$\frac{\lambda_2 - \theta}{\lambda_3 - \theta},$$

which is less than that of the case $\theta = 0$.

Backing to **STEP 2** of Algorithm 1, we denote

$$V^{\text{new}} = \left[X^{\text{new}},\ P^{\text{new}},\ W^{\text{new}}\right] = \left[V\hat{x},\ V\hat{p},\ W^{\text{new}}\right].$$

During solving the Rayleigh-Ritz problem, we need to assemble the small scale matrices $(V^{\text{new}})^\top A V^{\text{new}}$ and $(V^{\text{new}})^\top B V^{\text{new}}$. Since the orthogonalization to the vectors in V^{new} has been done under the inner product deduced by the matrix B, $(V^{\text{new}})^\top B V^{\text{new}}$ is an identity matrix. Then we only need to compute the matrix $\bar{A}_{\text{new}}$, which is equal to

$$\begin{bmatrix} (X^{\text{new}})^\top A X^{\text{new}} & (X^{\text{new}})^\top A P^{\text{new}} & (X^{\text{new}})^\top A W^{\text{new}} \\ (P^{\text{new}})^\top A X^{\text{new}} & (P^{\text{new}})^\top A P^{\text{new}} & (P^{\text{new}})^\top A W^{\text{new}} \\ (W^{\text{new}})^\top A X^{\text{new}} & (W^{\text{new}})^\top A P^{\text{new}} & (W^{\text{new}})^\top A W^{\text{new}} \end{bmatrix}. \tag{8}$$

From (3), the submatrix $(X^{\text{new}})^\top A X^{\text{new}}$ does not need to be computed explicitly since it satisfies the following formula

$$(X^{\text{new}})^\top A X^{\text{new}} = \hat{x}^\top V^\top A V \hat{x} = \hat{x}^\top \bar{A} \hat{x} = \Lambda_x. \tag{9}$$

Based on the basis in V and (6), we have

$$(P^{\text{new}})^\top A P^{\text{new}} = \hat{p}^\top V^\top A V \hat{p} = \hat{p}^\top \bar{A} \hat{p} \tag{10}$$

and

$$(P^{\text{new}})^\top A X^{\text{new}} = \hat{p}^\top V^\top A V \hat{x} = \hat{p}^\top \bar{A} \hat{x} = \hat{p}^\top \hat{x} \Lambda_x = O. \tag{11}$$

Thus from (8), (9), (10) and (11), we know the matrix $\bar{A}^{\text{new}}$ has the following structure

$$\begin{bmatrix} \Lambda_0 & O & O & O \\ O & \Lambda_1 & O & \alpha_1 \\ O & O & \alpha_0 & \alpha_2 \\ O & \alpha_1^\top & \alpha_2^\top & \alpha_3 \end{bmatrix}, \tag{12}$$

where

$$\Lambda_0 = \begin{bmatrix} \Lambda_c & O \\ O & \Lambda_{n_1} \end{bmatrix},\ \Lambda_1 = \begin{bmatrix} \Lambda_{n_2} & O & O \\ O & \Lambda_{\tilde{n}_1} & O \\ O & O & \Lambda_{\tilde{n}_2} \end{bmatrix},$$

$$\alpha_0 = \hat{p}^\top \bar{A} \hat{p},\ \alpha_1 = \left[X_n^{\text{new}},\ X_{\tilde{n}}^{\text{new}}\right]^\top A W^{\text{new}},$$
$$\alpha_2 = (P^{\text{new}})^\top A W^{\text{new}},\ \alpha_3 = (W^{\text{new}})^\top A W^{\text{new}}.$$

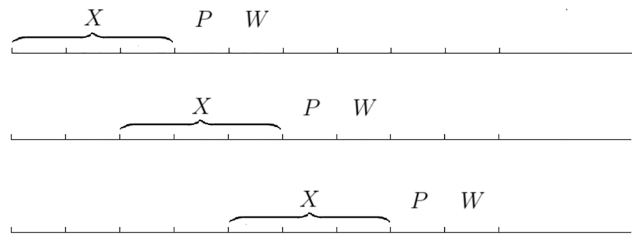


Fig. 1 Moving $[X, P, W]$, when $2 \times$ `blockSize` eigenpairs converged

It is noted that since X_c^{new} reaches the convergence criterion, we assume the equation

$$A X_c^{\text{new}} = B X_c^{\text{new}} \Lambda_c^{\text{new}}$$

is satisfied. Then

$$(W^{\text{new}})^\top A X_c^{\text{new}} = (W^{\text{new}})^\top B X_c^{\text{new}} \Lambda_c^{\text{new}} = O$$

is satisfied approximately since $(W^{\text{new}})^\top B X_c^{\text{new}} = O$.

After assembling matrix $\bar{A}^{\text{new}}$, the next task is to solve the new small scale eigenvalue problem:

$$\bar{A}^{\text{new}} \hat{x}^{\text{new}} = \hat{x}^{\text{new}} \Lambda_x^{\text{new}}, \tag{13}$$

in **STEP 3**. Due to the converged eigenvectors X_c^{new} in V^{new}, there are already ℓ converged eigenvectors of $\bar{A}^{\text{new}}$ and they all have the form

$$(0, ..., 0, 1, 0, ..., 0)^\top,$$

where 1 stays in the position of associated converged eigenvalue. We only need to compute the unconverged eigenpairs corresponding to $[X_n^{\text{new}}, X_{\tilde{n}}^{\text{new}}]$ for the eigenvalue problem (13). The subroutine `dsyevx` from LAPACK (Anderson et al. 1999) is called to compute the $(\ell + 1)$-th to `numEigen` -th eigenvalues and their associated eigenvectors.

In order to reduce time consuming for solving (13), this task is distributed to multi computing processes and each process only computes a small part of desired eigenpairs. After all processes finish their tasks, the subroutine `MPI_Allgatherv` is adopted to gather all eigenpairs from all processes and deliver them to all. This way leads to an obvious time reduction for computing the desired eigenpairs of (13). Since more processes lead to more communicating time, we choose the number of used processes for solving (13) such that each process computes at least 10 eigenpairs.

Remark 3.2 In order to accelerate the convergence, the size of X, `sizeX`, is always chosen to be greater than `numEigen`. In GCGE, `sizeX` is set to be the minimum of `numEigen` $+ 3 \times$ `blockSize` and the dimension of A, as default.

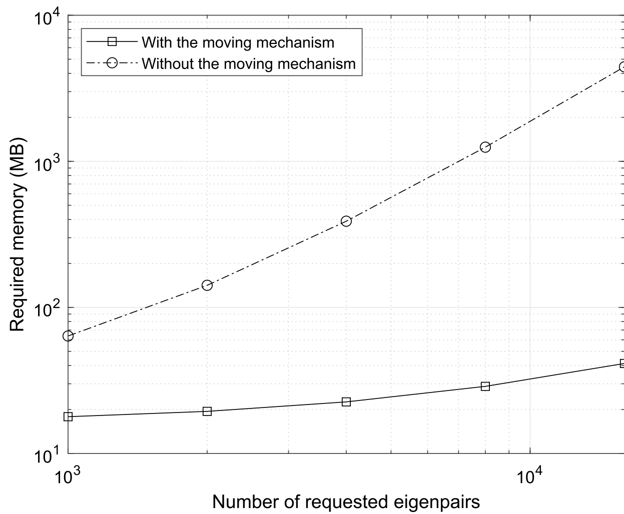


Fig. 2 Requested memory in each process

Table 1 Supported matrix–vector structures

	matrix structure name	vector structure name
MATLAB	sparse distributed matrix	full stored matrix
Hypre	hypre_ParCSRMatrix	hypre_ParVector
PETSc	Mat	Vec
PHG	MAT	VEC
SLEPc	Mat	BV

Remark 3.3 Since the converged eigenpairs (Λ_c, X_c) do not participate the subsequent iterations, $\bar{A}$ is computed as follows

$$\left[X_n,\ X_{\widetilde{n}},\ P,\ W\right]^{\top} A \left[X_n,\ X_{\widetilde{n}},\ P,\ W\right],$$

and the corresponding eigenpairs have the forms

$$\begin{bmatrix} \Lambda_n & O \\ O & \Lambda_{\widetilde{n}} \end{bmatrix}, \begin{bmatrix} \widehat{x}_{nn} & \widehat{x}_{n\widetilde{n}} \\ \widehat{x}_{\widetilde{n}n} & \widehat{x}_{\widetilde{n}\widetilde{n}} \\ \widehat{x}_{pn} & \widehat{x}_{p\widetilde{n}} \\ \widehat{x}_{wn} & \widehat{x}_{w\widetilde{n}} \end{bmatrix}.$$

In other words, the internal locking (deflation) is implemented to avoid the computation to the converged eigenpairs.

3.3 The moving mechanism

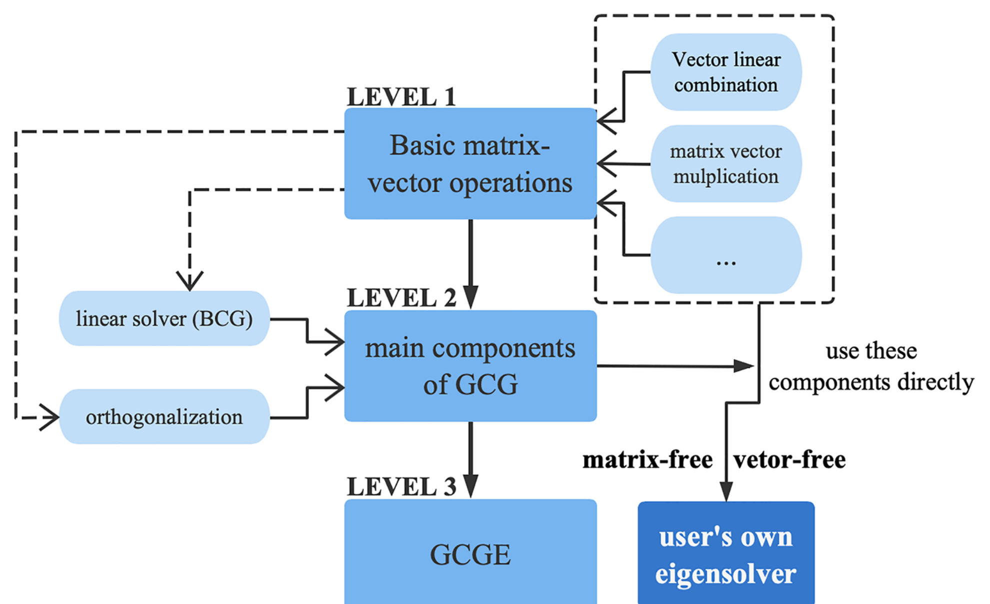
In Algorithm 1, the small scale eigenvalue problem (13) needs to be solved, in which the dimension of the dense matrix $\bar{A}$ is

$$\texttt{sizeX} + 2 \times \texttt{blockSize},$$

where the size of X, `sizeX`, is equal to `numEigen` $+ 3 \times$ `blockSize`. When `numEigen` is large, e.g., 5000, with `blockSize` $= 200$, `dsyevx` should be called to solve 5000 eigenpairs for a dense matrix of 6000-dimension. In this case, the time of **STEP 3** of Algorithm 1 is always dominated.

In order to improve efficiency further for computing plenty of eigenpairs, we present a moving mechanism. Firstly, the maximum project dimension is set to be `maxProjDim` $= 5 \times$ `blockSize` in moving procedure, i.e., the size of X is set to be $3 \times$ `blockSize` and the sizes of P and W are both `blockSize`. Secondly, when $2 \times$ `blockSize` eigenpairs converged, all the eigenpairs of $\bar{A}$ will be solved, i.e, $\bar{A}$ is decomposed into

Fig. 3 The software structure of GCGE



$$\bar{A} = \left[\hat{x},\ \hat{p},\ \hat{w}\right] \Lambda_{xpw} \left[\hat{x},\ \hat{p},\ \hat{w}\right]^{-1},$$

where

$$\bar{A} = V^{\top}AV,\ V = \left[X,\ P,\ W\right].$$

In addition, the new X is equal to $V[\hat{x}, \hat{p}, \hat{w}]$, and Λ_{xpw} can be used to construct the new $\bar{A}$ in the next **STEP 3**. In other words, P and W have been integrated into X. Then, the new P and W will be computed and stored behind the new X. When there are new converged $2 \times \texttt{blockSize}$ eigenpairs again, P and W will be integrated into X again, and so on. The above process is shown in Fig. 1. It is noted that the dimension of the dense matrix $\bar{A}$ is $\texttt{maxProjDim} = 5 \times \texttt{blockSize}$ at most in the eigenvalue problem (13).

Moreover, the moving mechanism can greatly reduce memory requirements, which allows more eigenpairs to be computed under the same memory requirement. Specifically speaking, the double array, of which the size is

$$(\texttt{sizeX} + 2 \times \texttt{blockSize}) + 2 \times (\texttt{maxProjDim})^2 + 10 \times (\texttt{maxProjDim}) + \texttt{sizeX} \times \texttt{blockSize}, \tag{14}$$

is required to be stored in each process. The first two terms denote the sizes of the two arrays which are used to store the eigenpairs and the dense matrix in the small scale eigenvalue problem (13). The third term is the size of workspace for `dsyevx`. The last term is the size of the array which is used in **STEP 5**. In Fig. 2, the required memory computed by (14) is shown with and without the moving mechanism.

4 GCGE package

Based on Algorithm 1 and its implementing techniques presented in above sections, we develop the package GCGE, which is written by C language and constructed with the way of matrix-free and vector-free. So far, the package has included the eigensolvers for the matrices which are stored in dense format, compressed row/column sparse format or are supported in MATLAB, Hypre (Falgout et al. 2006), PETSc (Balay et al. 1997), PHG (Zhang 2009) and SLEPc (Hernandez et al. 2005). Table 1 presents the currently supported matrix–vector structure. It is noted that there is no need to copy the built-in matrices and the vectors from these softwares/libraries to the GCGE package.

For understanding, Fig. 3 shows the three levels of structure of the GCGE package. Level 1 includes all the basic matrix and vector operations. Level 2 implements the GCG algorithm with the matrix-free and vector-free way, which is the main part of GCGE. Level 3 combines the matrix and vector operations from Level 1 and the GCG algorithm from Level 2 to build the eigensolver.

A user can also build his own eigensolver by providing the matrix, vector structures and their operations. The following six matrix–vector operations should be provided by the user:

(1) `VecCreateByMat`
(2) `VecDestroy`
(3) `VecLocalInnerProd`
(4) `VecSetRandomValue`
(5) `VecAxpby`
(6) `MatDotVec`

They realize creating and destroying vector according to matrix, computing local inner product of vectors x and y, setting random values for vector x, computing vector $y = \alpha x + \beta y$, computing vector $y = Ax$, respectively. `VecInnerProd`, i.e., computing inner product of vectors x and y, has been provided through calling `VecLocalInnerProd` and `MPI_Allreduce`.

The default matrix-multi-vector operations are invoked based on the above matrix–vector operations and the additional two operations: `GetVecFromMultiVec` and `RestoreVecForMultiVec`, which are getting/restoring one vector from/to multi vectors. For higher efficiency, it is strongly recommended that users should provide the following six matrix-multi-vector operations:

(1) `MultiVecCreateByMat`
(2) `MultiVecDestroy`
(3) `MultiVecLocalInnerProd`
(4) `MultiVecSetRandomValue`
(5) `MultiVecAxpby`
(6) `MatDotMultiVec`

In addition, if user-defined multi-vector is stored in dense format, BLAS library can be used to implement (1)-(5) operators easily, which has been provided in the GCGE package. In other words, only one operator, i.e., computing the multiplication of matrix and multi-vector needs to be provided by users.

In order to improve the parallel efficiency of computing inner products of multi-vectors X and Y, i.e., the operation `MultiVecInnerProd`, a new MPI data type with the corresponding reduced operation has been created by

```
MPI_Type_vector, MPI_Op_create.
```

The variable `MPI_IN_PLACE` is used as the value of `sendbuf` in `MPI_Allreduce` at all processes.

Although SLEPc (Hernandez et al. 2005) provides an inner product operation for BV structure, we still recommend using our own multi-vector inner product operation.

Table 2 Testing matrices

ID	Matrix	Dimension	Non-zero entries	Density
1	Andrews	60,000	760,154	2.11e-4
2	CO	221,119	7,666,057	1.57e-4
3	Ga10As10H30	113,081	6,115,633	4.78e-4
4	Ga19As19H42	133,123	8,884,839	5.01e-4
5	Ga3As3H12	61,349	5,970,947	1.59e-3
6	Ga41As41H72	268,096	18,488,476	2.57e-4
7	Ge87H76	112,985	7,892,195	6.18e-4
8	Ge99H100	112,985	8,451,395	6.62e-4
9	Si34H36	97,569	5,156,379	5.42e-4
10	Si41Ge41H72	185,639	15,011,265	4.36e-4
11	Si5H12	19,896	738,598	1.87e-3
12	Si87H76	240,369	10,661,631	1.85e-4
13	SiO2	155,331	11,283,503	4.68e-4
14	FEM matrices A and B	14,045,759	671,028,055	3.40e-6

Let us give an example to illustrate the reason. For instance, we need to compute the inner products

$$[x_i, \cdots, x_j]^{\mathrm{T}}[y_p, \cdots, y_q]$$

and the results are stored in the following submatrix

$$\begin{bmatrix} c_{ip} & \cdots & c_{iq} \\ \vdots & & \vdots \\ c_{jp} & \cdots & c_{jq} \end{bmatrix}. \tag{15}$$

Always, the vectors $[x_i, \cdots, x_j]$ and $[y_p, \cdots, y_q]$ come from the multi-vector

$$X = [x_1, \cdots, x_i, \cdots, x_j, \cdots, x_n],$$
$$Y = [y_1, \cdots, y_p, \cdots, y_q, \cdots, y_m].$$

and the dense matrix (15) is one submatrix of the following matrix

$$\begin{bmatrix} * & * & \cdots & * & * \\ * & c_{ip} & \cdots & c_{iq} & * \\ * & \vdots & & \vdots & * \\ * & c_{jp} & \cdots & c_{jq} & * \\ * & * & \cdots & * & * \end{bmatrix}_{s\times t},$$

which is stored by column. Thus, it can be noted that the above mentioned submatrix (15) is not stored continuously.

The result of the SLEPc's inner product operation, `BVDot`, must be stored in a sequential dense matrix with dimensions $n \times m$ at least. In other words, regardless of the values of i, j, p and q, in each process, the additional memory space is required, of which the size is $n \times m$. In general, n and m are set to be $\texttt{sizeX} + 2 \times \texttt{blockSize}$ in the GCG algorithm, while s and t are much less than n and m, respectively.

In the GCGE package, the operation `MultiVecInnerProd` is implemented as follows:

(1) Through `MultiVecLocalInnerProd`, local inner products are calculated and stored in the above mentioned submatrix for each process;

(2) A new `MPI_Datatype` named `SUBMAT` is created by

```
int MPI_Type_vector(int count, int length, int stride,
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

with

```
count = q-p+1,  length = j-i+1,  stride = s;
```

(3) Through `MPI_Op_create`, the operation of sum of `SUBMAT` is created, which is named as `SUM_SUBMAT`;

(4) Then

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op,  MPI_Comm comm)
```

is called with

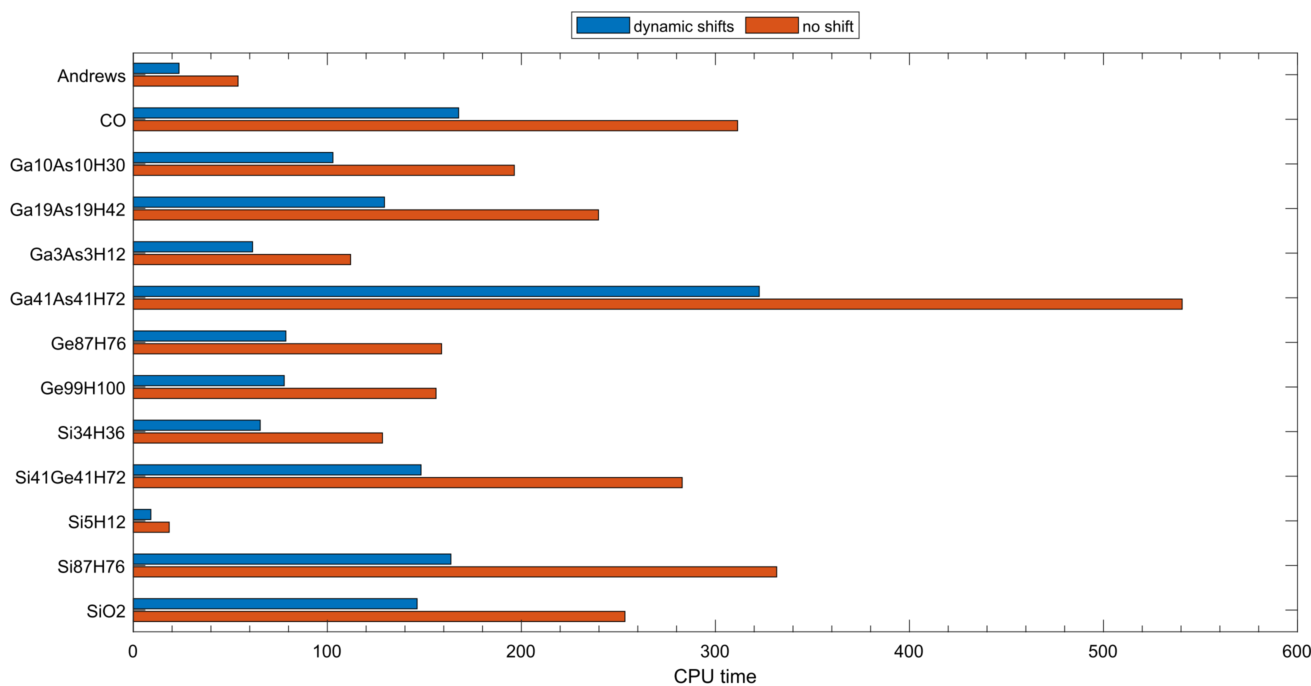


Fig. 4 `tol` $= 10^{-8}$, `numEigen` = 800, and `numProc` = 36

Table 3 The total number of GCG iterations

ID	Matrix	Dynamic shifts	No shift	Ratio
1	Andrews	102	281	36.29%
2	CO	97	195	49.74%
3	Ga10As10H30	105	213	49.29%
4	Ga19As19H42	110	216	50.92%
5	Ga3As3H12	81	165	49.09%
6	Ga41As41H72	133	236	56.35%
7	Ge87H76	78	212	36.79%
8	Ge99H100	77	206	37.37%
9	Si34H36	79	207	38.16%
10	Si41Ge41H72	87	208	41.82%
11	Si5H12	86	201	42.78%
12	Si87H76	89	232	38.36%
13	SiO2	90	164	54.87%

Table 4 FEM matrices with `numEigen` = 800, `tol` $= 10^{-12}$ and `numProc` = 576

	The total number of GCG iterations	CPU time (in seconds)
Dynamic shifts	83	1669.19
No shift	88	1777.87
Ratio	94.31%	93.88%

```
sendbuf = MPI_IN_PLACE,  count = 1,
datatype = SUBMAT,  op = SUM_SUBMAT
```

to gather values from all processes and distribute the results back to all processes.

Obviously, no extra workspace is needed here. The memory requirements are reduced for each process.

5 Numerical results

The numerical experiments in this section are carried out on LSSC-IV in the State Key Laboratory of Scientific and Engineering Computing, Chinese Academy of Sciences. Each computing node has two 18-core Intel Xeon Gold 6140 processors at 2.3 GHz and 192 GB memory. For more information, please check http://lsec.cc.ac.cn/chinese/lsec/LSSC-IVintroduction.pdf. We use `numProc` to denote the number of processes in numerical experiments.

In this section, the GCG algorithm defined by Algorithm 1 and the implementing techniques in Sect. 3 are investigated for thirteen standard eigenvalue problems and one generalized eigenvalue problem. The first thirteen matrices

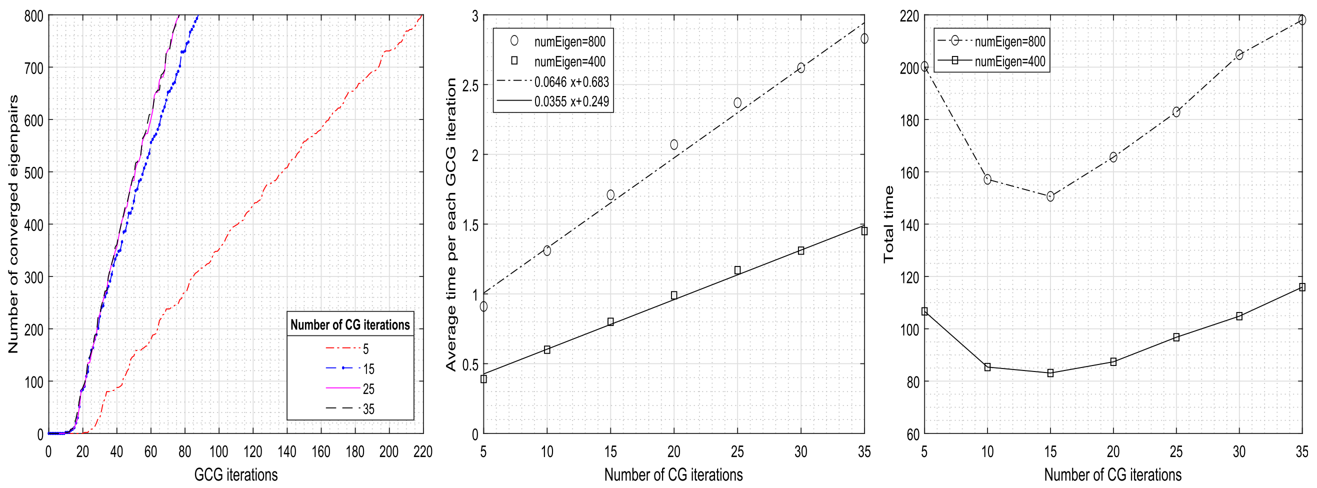


Fig. 5 Convergence procedure (left), average time of each GCG step (middle) and Figure4-1total time (right) of GCGE for SiO2 with `tol` $= 10^{-8}$

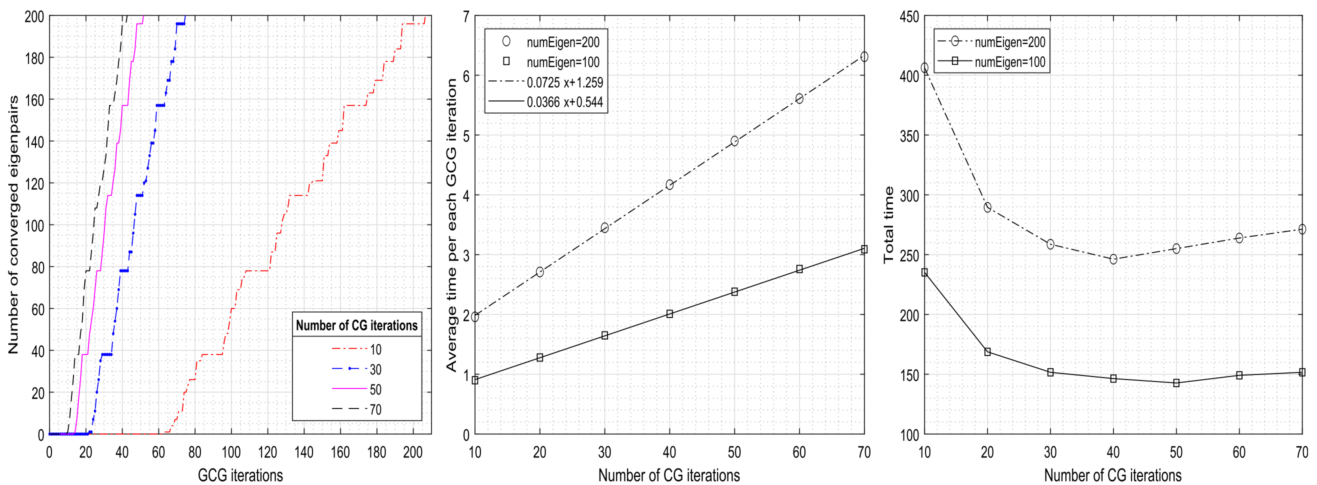


Fig. 6 Convergence procedure (left), average time of each GCG step (middle) and total time (right) of GCGE for FEM matrices with `tol` $= 10^{-8}$

are available in Suite Sparse Matrix Collection[1], which have clustered eigenvalues and many negative eigenvalues. The first matrix named Andrews is provided by Stuart Andrews at Brown University, which has seemingly random sparsity pattern. The second to the thirteenth matrices are generated by the pseudo-potential algorithm for real-space electronic structure calculations (Kronik et al. 2006; Natan et al. 2008; Saad et al. 2010). The FEM matrices A and B come from the finite element discretization for the following Laplace eigenvalue problem: Find $(\lambda, u) \in \mathbb{R} \times H_0^1(\Omega)$ such that

$$\begin{cases} -\Delta u &= \lambda u, \quad \text{in } \Omega, \\ \quad\; u &= 0, \quad\;\; \text{on } \partial\Omega, \end{cases} \tag{16}$$

where $\Omega = (0,1) \times (0,1) \times (0,1)$. The discretization of the eigenvalue problem (16) by the conforming cubic finite element (P3 element) with 3,145,728 elements leads to the stiffness matrix A and the mass matrix B. The concerned matrices are listed in Table 2, where the density is defined by

[1] https://sparse.tamu.edu.

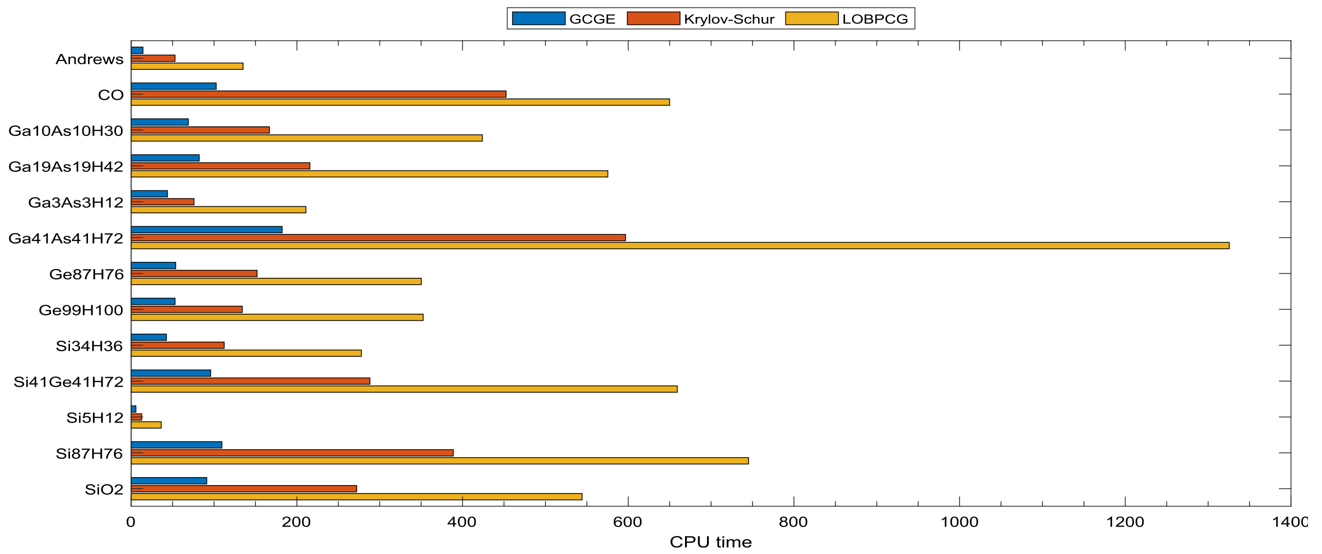


Fig. 7 $\texttt{tol} = 10^{-4}$, $\texttt{numEigen} = 800$, and $\texttt{numProc} = 36$

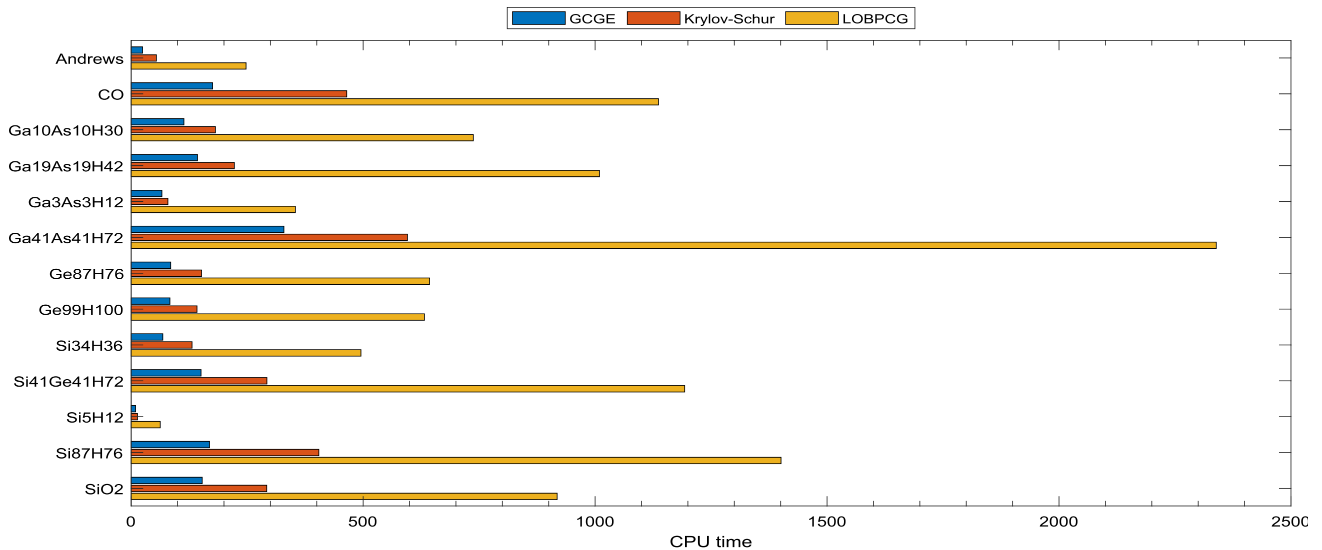


Fig. 8 $\texttt{tol} = 10^{-8}$, $\texttt{numEigen} = 800$, and $\texttt{numProc} = 36$

$$\frac{\text{the number of non-zero entries}}{\text{dimension} \times \text{dimension}}.$$

The proposed GCG algorithm given by Algorithm 1 based on BV structure from SLEPc is adopted to solve eigenpairs of the concerned matrices in Table 2.

The convergence criterion is set to be

$$\|Ax - \lambda x\|_2/\|x\|_2 < \texttt{tol}$$

for the first thirteen matrices and

$$\|Ax - \lambda Bx\|_2/(\lambda\|B^{1/2}x\|_2) < \texttt{tol}$$

for FEM matrices, where the tolerance, $\texttt{tol}$, is set to be 10^{-8} as default. Moreover, we set $\texttt{blockSize} = \texttt{numEigen}/10$ for the first thirteen matrices and $\texttt{blockSize} = \texttt{numEigen}/5$ for FEM matrices. In addition, we choose the parameters:

```
-eps_lobpcg_blocksize numEigen/5
-eps_lobpcg_restart   0.1
```

such that the LOBPCG has the best efficiency for comparison.

In order to confirm the efficiency, stability and scalability of GCGE, we investigate the numerical comparison between

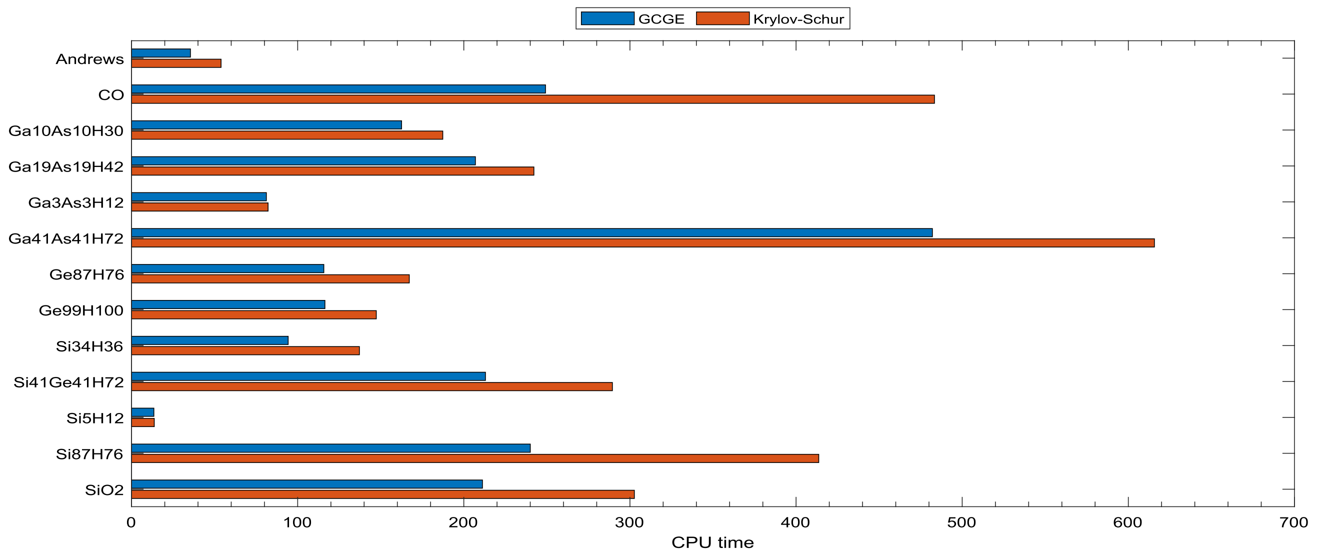


Fig. 9 `tol` $= 10^{-12}$, `numEigen` $= 800$, and `numProc` $= 36$

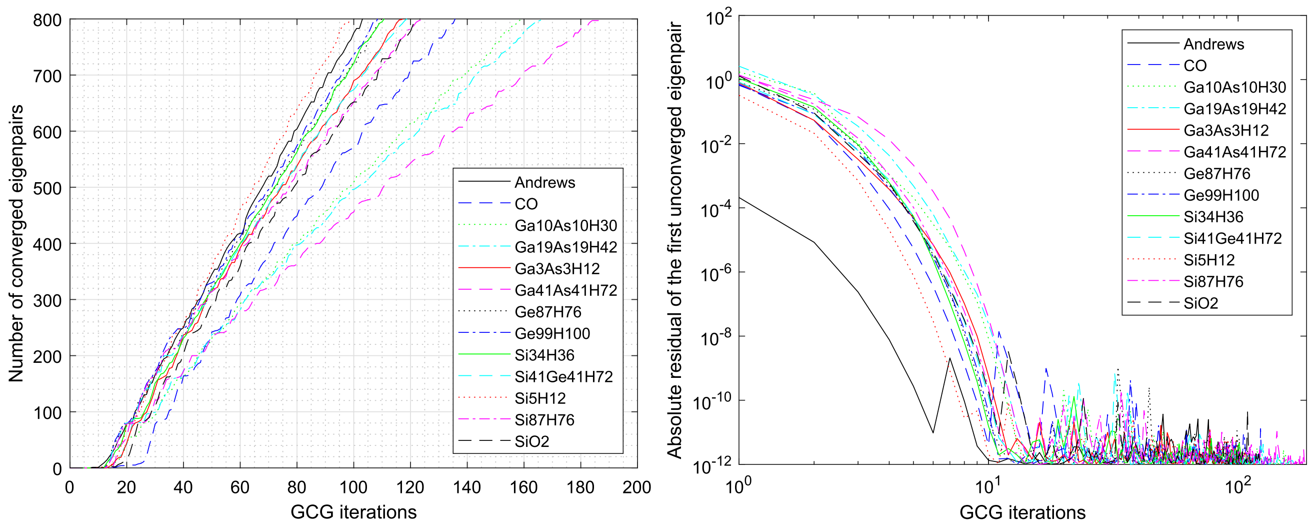


Fig. 10 Convergence procedure (left) and absolute residual of the first non-converged eigenpair(right) for the first thirteen matrices with `tol` $= 10^{-12}$, `numEigen` $= 800$, and `numProc` $= 36$

GCGE and LOBPCG. We will find that GCGE has better efficiency, stability than LOBPCG and they have almost the same good scalability. In addition, Krylov-Schur method is also compared in Sects. 5.2 and 5.5.

5.1 About dynamic shifts and the number of CG iterations

In this subsection, we give some numerical results to show the performance of GCGE with dynamic shifts and the convergence procedure under different number of CG iterations.

In **STEP 6** of Algorithm 1, the linear equations (7) are solved by some CG iterations. Due to the shift θ, the multiplication of matrix and vector of each CG iteration takes more time, but the convergence of GCG algorithm is accelerated. For the standard eigenvalue problems, i.e., $B = I$, because the additional computation is only the linear operations on vectors, each GCG iteration with dynamic shifts takes a little more time than the case of no shift. As shown in Fig. 4, the performance of GCGE with dynamic shifts is greatly improved. In addition, the total number of GCG iterations is presented in Table 3.

For the generalized eigenvalue problems, there is no significant improvement for the overall performance of GCGE with dynamic shifts by the additional computation of the multiplication of matrix B and vectors. When the

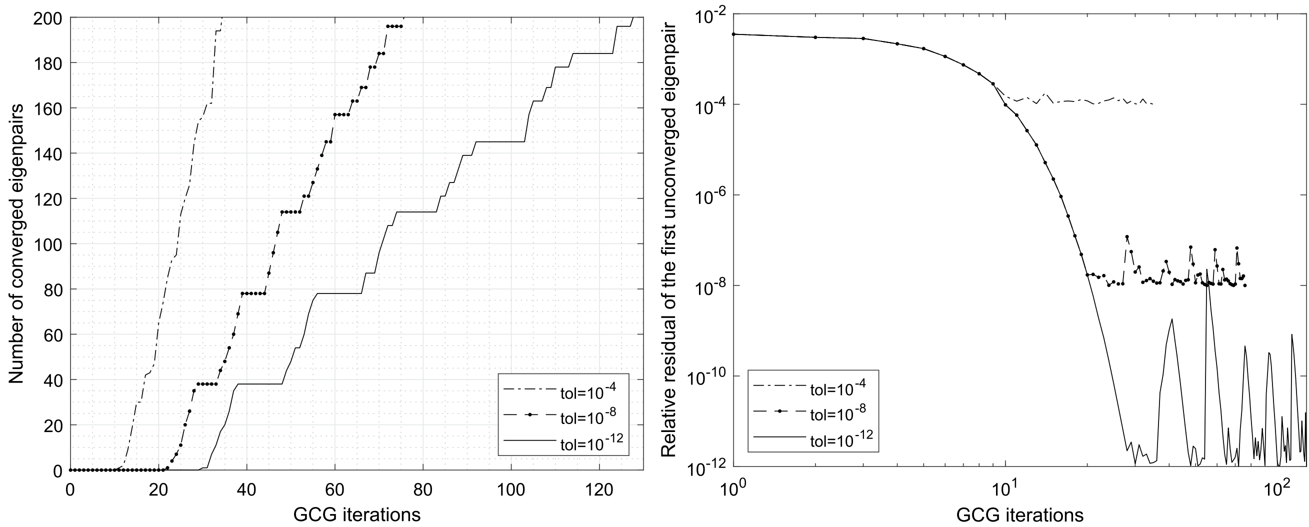


Fig. 11 Convergence procedure (left) and absolute residual of the first non-converged eigenpair(right) for the FEM matrices with $\texttt{numEigen} = 200$ and $\texttt{numProc} = 576$

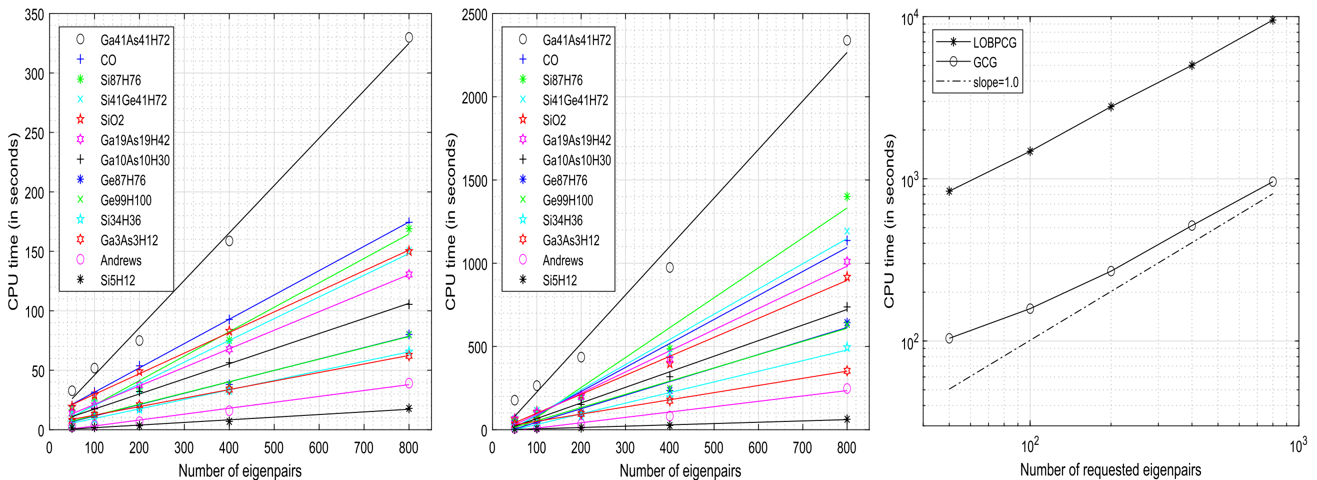


Fig. 12 CPU time of GCGE (left) and LOBPCG (middle) for the first thirteen matrices with $\texttt{tol} = 10^{-8}$ and $\texttt{numProc} = 36$, CPU time for FEM matrices (right) with $\texttt{tol} = 10^{-8}$ and $\texttt{numProc} = 576$

Table 5 CPU time for FEM matrices (P1 element) with $\texttt{tol} = 10^{-8}$ and $\texttt{numProc} = 36$

Method	50	100	200
GCGE	20.15	38.98	71.49
Krylov-Schur	1032.33	1360.56	2180.28
LOBPCG	63.99	114.65	286.67

matrix A can be modified, we recommend users to generate $A - \theta B$ explicitly and do CG steps for $A - \theta B$ directly. In this event, GCGE with dynamic shifts will perform better for the generalized eigenvalue problem and the results for $\texttt{numEigen} = 800$ and $\texttt{numEigen} = 5000$ are shown in Tables 4 and 7, respectively.

In addition, the GCG algorithm does not need to solve linear equations exactly in **STEP 6**. In the rest of this subsection, the total time of GCGE and the average time per each GCG iteration are presented under different number of CG iterations. Because the first thirteen matrices have similar density, we choose SiO2 with $\texttt{numProc} = 36$ and FEM matrices with $\texttt{numProc} = 576$ for the test.

For SiO2 with $\texttt{numEigen} = 400$ and 800, as shown in Fig. 5, when the number of CG iterations is increased from 5 to 35 in each GCG iteration, the number of GCG iterations decreases and the average time per each GCG iteration increases. And the total time reaches a minimum near

Table 6 Small scale matrices with $\mathtt{tol} = 10^{-8}$ and $\mathtt{numEigen} = 800$

numProc	Method	Andrews	Ga3As3H12	Si5H12
36	GCGE	37.28	60.32	9.38
	Krylov-Schur	54.66	69.90	11.95
	LOBPCG	447.09	650.75	113.48
72	GCGE	26.09	39.85	7.65
	Krylov-Schur	26.34	34.13	5.30
	LOBPCG	247.59	353.91	62.59
144	GCGE	22.69	26.54	7.11
	Krylov-Schur	14.82	15.73	2.90
	LOBPCG	155.08	193.54	44.10
288	GCGE	34.55	20.36	8.64
	Krylov-Schur	16.22	8.49	2.69
	LOBPCG	162.46	115.88	34.98

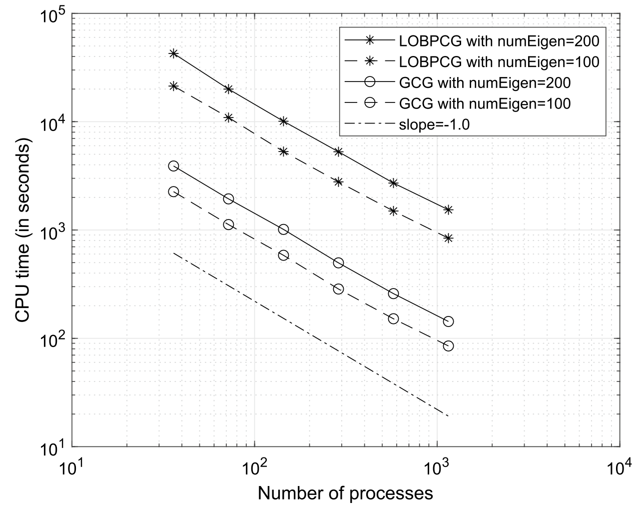


Fig. 14 CPU time for FEM matrices with $\mathtt{tol} = 10^{-8}$

15 CG iterations according to the right subfigure in Fig. 5. In fact, the examples from Andrews to SiO2 have similar conclusions.

Figure 6 shows the corresponding results for FEM matrices with $\mathtt{numEigen} = 100$ and 200. When the number of CG iterations is increased from 10 to 70, the number of GCG iterations decreases and the average time per each GCG iteration increases. The best performance is achieved at 30–40 CG iterations as shown in the right subfigure in Fig. 6.

It is noted that the number of CG iterations in each GCG iteration affects the efficiency of the algorithm deeply as presented in Figs. 5 and 6. The average time per each GCG iteration is linearly associated with the number of CG iterations. So, the number of CG iterations is a key parameter for trading off between the number of GCG iterations and the average time for each GCG iteration. In fact, the total time of GCG algorithm is nearly equal to the multiplication of the number of GCG iterations and the average time of GCG iterations. In other words, though increasing the number of CG iterations can accelerate convergence, it takes more time in each GCG iteration.

In fact, the sparsity, the condition number and the dimension of the matrix all affect the convergence rate of the CG iteration. In the GCGE package, we set two stop conditions of the CG iteration. When the residual of the solution is less than one percent of the initial residual, or the number of CG iterations is greater than 30, the CG iteration will be stopped.

5.2 About different tolerances

In this subsection, we will compare the performance of GCGE, LOBPCG and Krylov-Schur methods under different tolerances.

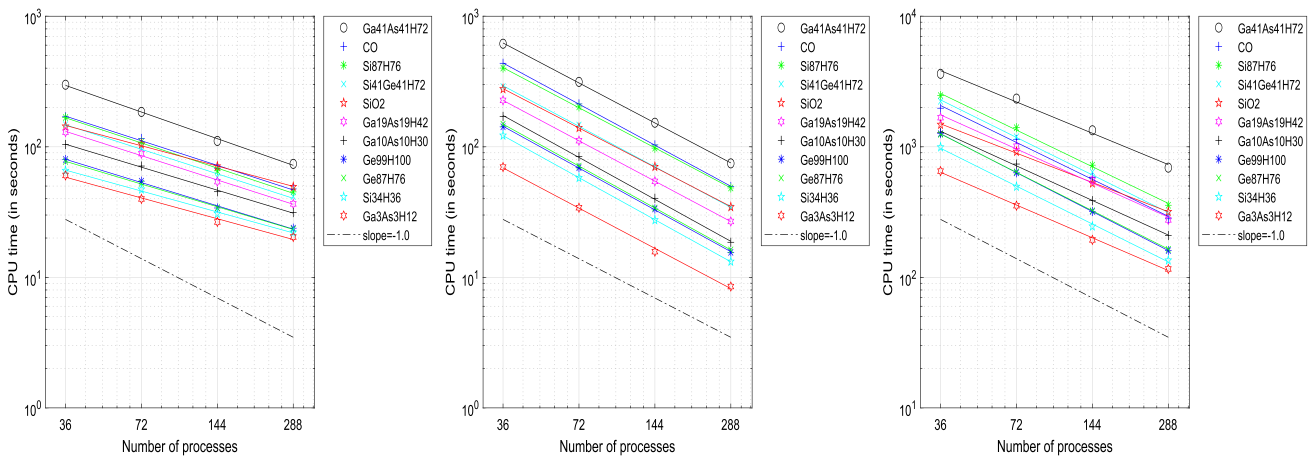


Fig. 13 CPU time of GCGE (left), Krylov-Schur (middle) and LOBPCG (right) for the first thirteem matroces with $\mathtt{tol} = 10^{-8}$ and $\mathtt{numEigen} = 800$

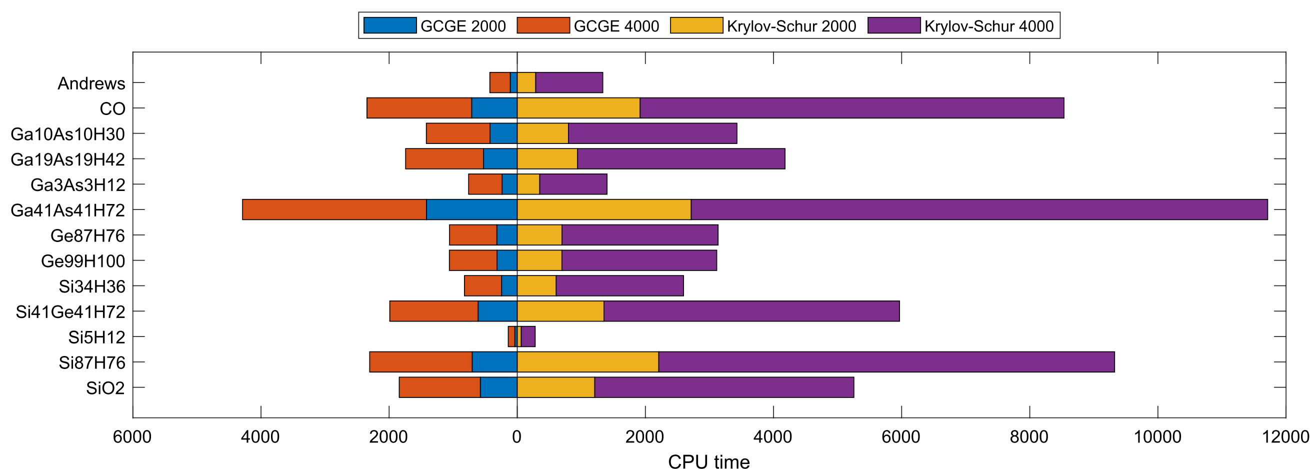


Fig. 15 `tol` $= 10^{-12}$, `blockSize` $= 100$, and `numProc` $= 36$

Table 7 The performance for FEM matrices with `numEigen` $= 5000$, `tol` $= 10^{-8}$, `blockSize` $= 200$, and `numProc` $= 1152$

	Without moving mechanism		With moving mechanism		With moving mechanism	
	Without dynamic shifts		Without dynamic shifts		With dynamic shifts	
	Time	Percentage	Time	Percentage	Time	Percentage
STEP 2	445.05	5.33%	20.02	0.48%	20.03	0.62%
STEP 3	4727.57	56.65%	729.12	17.79%	605.75	18.67%
STEP 4	78.94	0.95%	41.54	1.01%	30.39	0.94%
STEP 5	281.12	3.37%	122.68	2.99%	99.29	3.06%
STEP 6	2811.89	33.70%	3185.45	77.72%	2489.61	76.72%
Total time	8344.57	100.00%	4098.83	100.00%	3245.07	100.00%
Ratio	100.00%		49.12%		38.89%	

In Figs. 7 and 8, GCGE, LOBPCG and Krylov-Schur methods with `numProc` $= 36$ are compared under `tol` $= 10^{-4}$ and 10^{-8}, respectively. Under the tolerance 10^{-12}, LOBPCG can not converge after 3000 iterations, which means that the LOBPCG has no good stability. So only the performance of GCGE and Krylov-Schur methods are compared under `tol` $= 10^{-12}$ and the results are presented in Fig. 9. Here, MUMPS (Amestoy et al. 2001, 2019) is used as linear solver for Krylov-Schur method.

Obviously, GCGE is always more efficient than LOBPCG under different tolerances. In addition, when `tol` $= 10^{-4}$ and 10^{-8}, GCGE is much faster than Krylov-Schur method. Under tolerances 10^{-12}, the CPU time of GCGE and Krylov-Schur method is similar and GCGE is slightly faster.

In addition, the convergence procedure of GCG algorithm with `tol` $= 10^{-12}$ for the first thirteen matrices is shown in the left subfigure of Fig. 10. As the number of GCG iterations increases, the number of converged eigenpairs increases. In the right subfigure of Fig. 10, the absolute residual of the first unconverged eigenpair is presented.

For FEM matrices, the performances of GCGE are shown in Fig. 11. Due to `blockSize` $=$ `numEigen`$/5 = 40$, there are four noticeable pauses for the case of `tol` $= 10^{-12}$ when the number of converged eigenpairs is close to $1 \times 40, 2 \times 40, 3 \times 40$, and 4×40 at around the 40th, 60th, 80th, and 100th GCG iteration. Roughly speaking, the 40 eigenpairs can be converged once every twenty GCG iterations.

5.3 Scaling for the number of eigenpairs

Here, we investigate the dependence of computing time on the number of desired eigenpairs. For this aim, we compute the first 50-800 eigenpairs of matrices listed in Table 2.

The test for the first thirteen matrices is performed on a single node with 36 processes. The results in Fig. 12

show that just like LOBPCG, GCGE has almost linear scaling property, which means the computing time is linearly dependent on the number of desired eigenpairs. Moreover, GCGE has better efficiency than LOBPCG. From Andrews to SiO2, the total time (the summation of CPU time over all cases) ratios of GCGE to LOBPCG are

17.59%, 19.17%, 16.70%, 15.02%, 19.35%, 15.46%,
14.67%, 14.43%, 15.85%, 14.44%, 28.82%, 14.15%, 19.55%.

Since the scales of FEM matrices are large, the test is performed with 576 processes on 16 nodes. The dependence of CPU time (in seconds) for FEM matrices on the number of eigenpairs is shown in the right subfigure of Fig. 12, which implies that GCGE has better efficiency than LOBPCG for large scale matrices. Moreover, GCGE and LOBPCG both have almost linear scaling property for large scale FEM matrices.

Remark 5.1 In fact, Krylov-Schur method is low efficient for FEM matrices on multi-nodes. In Table 5, for $\texttt{numEigen} = 50, 100, 200$, the generalized eigenvalue problem is tested, which is the discretization of the eigenvalue problem (16) for the conforming linear finite element (P1 element) with 3,145,728 elements. The dimensions of the matrices A and B are both 512,191.

5.4 Scalability test

In order to do the scalability test, we use 36-288 processes to compute the first 800 eigenpairs of the first thirteen matrices listed in Table 2. The comparisons of the scalability of GCGE and LOBPCG are shown in Fig. 13 and Table 6. It is noted that GCGE, LOBPCG, and Krylov-Schur methods have similar scalability for the first thirteen matrices, but the total time (the summation of CPU time over all cases) ratios of GCGE to LOBPCG are

11.92%, 10.10%, 9.61%, 8.79%, 11.19%, 8.37%,
7.88%, 8.10%, 8.86%, 7.93%, 12.85%, 7.82%, 11.36%,

from Andrews to SiO2. In other words, GCGE has better efficiency than LOBPCG. In addition, the total time (the summation of CPU time over all cases) ratios of GCGE to Krylov-Schur method are

107.63%, 50.08%, 80.26%, 73.64%, 114.66%, 57.77%,
69.20%, 73.86%, 75.06%, 64.07%, 143.52%, 51.67%, 70.54%,

from Andrews to SiO2. Only for small scale matrices Andrews (60,000), Ga3As3H12 (61,349), and Si34H36 (97,567), the Krylov-Schur method is more efficient than GCGE, which are shown in Table 6.

About the large scale FEM matrices, we use 36-1152 processes for computing the lowest 100 and 200 eigenpairs. In Fig. 14, we can find that GCGE and LOBPCG have similar scalability for large scale matrices, but GCGE has better efficiency. And the total time ratio of GCGE to LOBPCG is about 10%.

5.5 The performance of GCGE with large `numEigen`

In this subsection, the performance of the moving mechanism presented in Sect. 3.3 is tested. The maximum project dimensions, `maxProjDim`, are set to 1000 and 2000 for the first thirteen matrices and FEM matrices, respectively.

In Fig. 15, the performance of GCGE with the moving mechanism is shown for the first thirteen matrices. For Krylov-Schur method, we set `numEigen` to be 2000 and 4000 and the parameters are

```
-eps_nev 2000
-eps_ncv 2400
-eps_mpd 800
```

and

```
-eps_nev 4000
-eps_ncv 4400
-eps_mpd 1000
```

respectively, such that Krylov-Schur method has best efficiency for comparison. Figure 15 shows that GCGE has better efficiency than Krylov-Schur. From Andrews to SiO2, the total time (the summation of CPU time over $\texttt{numEigen} = 2000$ and 4000) ratios of GCGE to Krylov-Schur are

32.04%, 27.49%, 41.38%, 41.70%, 54.18%, 36.60%, 33.72%,
34.02%, 31.76%, 33.35%, 50.08%, 24.71%, 35.05%,

For FEM matrices with $\texttt{numEigen} = 5000$, without the moving mechanism, the time of **STEP 3** is dominated as shown in Table 7. And with the moving mechanism, the total time is reduced by about 50%. In addition, the total time with dynamic shifts is reduced by about 20% again due to the reduction of the total number of GCG iterations.

6 Concluding remarks

This paper highlights some new issues for computing plenty of eigenpairs of large scale matrices on high performance computers. The GCGE package is presented which is built with the damping block inverse power method with dynamic shifts for symmetric eigenvalue problems. Furthermore, in order to improve the efficiency, stability and scalability of the concerned package, the new efficient implementing techniques are designed for updating subspaces, orthogonalization and computing Rayleigh-Ritz problems. Plenty of numerical tests are provided to validate the proposed package GCGE, which can be downloaded from https://github.com/Materials-Of-Numerical-Algebra/GCGE.

Declarations

Conflict of interest All authors declare that they have no conflict of interest.

References

Amestoy, P.. R., Buttari, A., L'Excellent, J.. Y., Mary, T.: Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. ACM Trans. Math. Software **45**(1), 21–226 (2019)

Amestoy, Patrick. R., Duff, Iain S., Koster, Jacko, L'Excellent, Jean Yves: fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis & Applications **23**(1), 15–41 (2001)

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., GreenBaum, A., Hammarling, S., Mckenney, A., Sorensen, D.: LAPACK Users' Guide. SIAM (1999)

Bai, Zhaojun, Demmel, James, Dongarra, Jack, Ruhe, Axel, van der Vorst, Henk: Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide. Vol. 11. SIAM (2000)

Balay, Satish, Gropp, William D., McInnes, Lois Curfman, Smith, Barry F.: Efficient management of parallelism in object oriented numerical software libraries. In Modern Software Tools in Scientific Computing. . Birkhäuser Press (1997)

Duersch, Meiyue, amd Shao, Jed A., Yang, Chao: Robust and Efficient Implementation of LOBPCG. SIAM Journal on Scientific Computing **40**(5), 655–676 (2018)

Falgout, Robert D , Jones, Jim E, Yang, Ulrike Meier: The design and implementation of hypre, a library of parallel high performance preconditioners. In Numerical Solution of Partial Differential Equations on Parallel Computers. Springer, 267–294 (2006)

Hernandez, Vicente, Roman, Jose E., Vidal, Vicente: SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. ACM Trans. Math. Software **31**(3), 351–362 (2005)

Hetmaniuk, U., Lehoucq, R.: Basis selection in LOBPCG. J. Comput. Phys. **218**(1), 324–332 (2006)

Knyazev, Andrew V.: Toward the optimal preconditioned eigensolver: locally optimal block preconditioned conjugate gradient method. SIAM Journal on Scientific Computing **23**(2), 517–541 (2006)

Knyazev, Andrew V., Argentati, Merico E., Lashuk, Ilya, Ovtchinnikov, Evgueni E.: Block locally optimal preconditioned eigenvalue xolvers (BLOPEX) in HYPRE and PETSc. SIAM Journal on Scientific Computing **29**(5), 2224–2239 (2007)

Knyazev, Andrew V., Neymeyr, Klaus: Efficient solution of symmetric eigenvalue problems using multigrid preconditioners in the locally optimal block conjugate gradient method. Electronic Transactions on Numerical Analysis **15**(2003), 38–55 (2003)

Kronik, Leeor, Makmal, Adi, Tiago, Murilo L., Alemany, M..M..G., Jain, Manish, Huang, Xiangyang, Saad, Yousef, Chelikowsky, James R.: PARSEC – the pseudopotential algorithm for real-space electronic structure calculations: recent advances and novel applications to nano-structures. Physica Status Solidi **243**(5), 1063–1079 (2006)

Li, Yu., Xie, Hehu, Ran, Xu., You, Chun'Guang., Zhang, Ning: A parallel generalized conjugate gradient method for large scale eigenvalue problems. CCF Transactions on High Performance Computing **2**(2020), 111–122 (2020)

Natan, Amir, Benjamini, Ayelet, Naveh, Doron, Kronik, Leeor, Tiago, Murilo L., Beckman, Scott P., Chelikowsky, James R.: Real-space pseudopotential method for first principles calculations of general periodic and partially periodic systems. Physical Review B **78**(7), 75–109 (2008)

Saad, Youcef : Numerical Methods for Large Eigenvalue Problems. Vol. 158. SIAM (1992)

Saad, Yousef, Chelikowsky, James R., Shontz, Suzanne M.: Numerical methods for electronic structure calculations of materials. SIAM Rev. **52**(1), 3–54 (2010)

Stewart, G.W.: Block Gram-Schmidt Orthogonalization. SIAM Journal on Entific Computing **31**(1), 761–775 (2008)

Yokozawa, Takuya, Takahashi, Daisuke, Boku, Taisuke, Sato, Mitsuhisa: Efficient parallel implementation of classical Gram-Schmidt orthogonalization using matrix multiplication. In Proceedings of Fourth International Workshop on Parallel matrix Algorithms and Applications (PMAA'06). 37–38 (2006)

Zhang, Linbo: A parallel algorithm for adaptive local refinement of tetrahedral meshes using bisection. Numerical Mathematics: Theory, Methods and Applications **2**(2009), 65–89 (2009)

Zhang, N.Y., Li, H.X., Ran, X., You, C.: A generalized conjugate gradient method for eigenvalue problems. Sci. Sin. Math. **50**(12), 1–24 (2020)

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Yu Li received the B.S. degree in computational mathematics from Xiamen University, Fujian, China, in 2009, and the Ph.D. degree in computational mathematics from the Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing, China, in 2014. Since October 2014, he has worked at Coordinated Innovation Center for Computable Modeling in Management Science, Tianjin University of Finance and Economics, Tianjin, China. His current research interests include numerical methods for PDEs, eigenvalue problems, and stochastic optimal control.

Zijing Wang is currently a Ph.D. student in Academy of Mathematics and Systems Science, Chinese Academy of Sciences. She received the B.S. degree in school of mathematics from Hefei University of Technology, Anhui, China, in 2019, and the M.S. degree in computational mathematics from the Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing, China, in 2022. Her current research interests include numerical methods for eigenvalue problems, and parallel computing.

Hehu Xie received the B.S. degree in computational mathematics from Peking University, Beijing, China, in 2003, and the Ph.D. degree in computational mathematics from the Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing, China, in 2008. Since June 2008, he has worked at Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing, China. His current research interests include numerical methods for PDEs, eigenvalue problems, and machine learning method for high dimensional PDEs.