A parallel generator of non-hermitian matrices computed from given spectra

Xinzhe Wu, Serge Petiton, Yutong Lu

A Parallel Generator of Non-Hermitian Matrices computed from Given Spectra*

Xinzhe Wu[1,2], Serge G. Petiton[1,2], and Yutong Lu[3]

[1] Maison de la Simulation, CNRS, F-91191 Gif-sur-Yvette Cedex, France
[2] CRIStAL, UMR CNRS 9189, University of Lille, France
[3] National Supercomputing Center in Guangzhou, Sun Yat-sen University, China
{xinzhe.wu@ed.univ-lille1.fr, serge.petiton@univ-lille1.fr, yutong.lu@nscc-gz.cn}

Abstract. Iterative linear algebra methods are the important parts of the overall computing time of applications in various fields since decades. Recent research related to social networking, big data, machine learning and artificial intelligence has increased the necessity for non-hermitian solvers associated with much larger sparse matrices and graphs. The analysis of the iterative method behaviors for such problems is complex, and it is necessary to evaluate their convergence to solve extremely large non-Hermitian eigenvalue and linear problems on parallel and/or distributed machines. This convergence depends on the properties of spectra. Then, it is necessary to generate large matrices with known spectra to benchmark the methods. These matrices should be non-Hermitian and non-trivial, with very high dimension. This paper highlights a scalable matrix generator that uses the user-defined spectrum to construct large-scale sparse matrices and to ensure their eigenvalues as the given ones with high accuracy. This generator is implemented on CPUs and multi-GPU platforms. Good strong and weak scaling performance is obtained on several supercomputers. We also propose a method to verify its ability to guarantee the given spectra.

Keywords: Parallel · Non-Hermitian Matrix · Matrix Generation · Spectrum.

1 Introduction

The eigenvalue problem can be defined as finding some pairs (λ, u) with $\lambda \in \mathbb{C}$ and $u \in \mathbb{C}^m$ of matrix $A \in \mathbb{C}^{m \times m}$, that satisfy the relation $Au = \lambda u$. Many applications from various fields can be expressed as eigenvalue problems or linear system problems. In numerical simulations, Schrödinger equations, molecular simulations, geology, etc. are usually analyzed by solving eigenvalue problems

* This work was partially supported by the HPC Center of Champagne-Ardenne *Romeo*. It is funded by the project *MYX* of *French National Research Agency (ANR)* (Grant No. ANR-15-SPPE-003) under the SPPEXA framework.

and linear systems. In machine learning and pattern recognition, both supervised and unsupervised learning algorithms, such as principal component analysis (PCA), Fisher discriminant analysis (FDA), and clustering, often require solving eigenvalue problems. An insufficient accuracy and a failure of the solvers usually result in, respectively, a poor approximation to original problems and a failure of entire algorithms. A good selection of eigenvalue and linear system solvers becomes especially essential. Researchers urgently require test matrices to benchmark the numerical performance and parallel efficiency of these methods.

Nowadays, the size of eigenvalue/linear system problems and the supercomputer systems continues to scale up. The whole ecosystem of High Performance Computing (HPC), especially the linear algebra applications, should be adjusted to larger computing platforms. Under this background, there are four special requirements for the test matrices to evaluate the numerical algorithms: 1) the spectra must be known and can be customized; 2) sparse, non-Hermitian and non-trivial; 3) a very high dimension, including the non-zero element numbers and/or the matrix dimension to evaluate the algorithms on large-scale systems; 4) the controllable sparsity patterns. The matrix generator should be implemented in parallel to profit from the distributed memory clusters.

Since the eigenvalue/linear system solvers and some of their preconditioners are sensitive to a specific part of the spectra, the test matrices with customized spectra can help to analyze and provide numerically robust solvers. In practice, the spectrum is one of the important factors which influence the convergence of different solvers. Although the impact of spectral distribution of linear system on the Krylov solvers is complicated and cannot be ignored even for the normal matrix [8], the existence of test matrix generator with customized eigenvalues can still guide the study of numerical method. Moreover, the purpose of most preconditioners such as ILU, Jacobi, and SOR is to convert the distribution of related spectrum to another by right or left-multiplying a preconditioning matrix. The spectral distribution of the preconditioned matrix might speed up the convergence. As an example, X. Wu [14] et al. implemented a Unite and Conquer hybrid method for solving linear systems with the combination of a Krylov linear system solver, an eigenvalue solver, and a Least Square polynomial method proposed by Y. Saad [11] in 1987. In the preconditioning part of this method, the dominant eigenvalues are used to accelerate the convergence. It is extremely necessary to evaluate the influence of the distribution of dominant eigenvalues on the acceleration. In addition, some scientific communities may be interested in matrices with clustered, conjugated eigenvalues or other special spectral distributions. It is important to develop a very large set of non-Hermitian test matrices whose eigenvalues can be customized.

The properties of being sparse, non-Hermitian and non-trivial together can add many mathematical features for the test matrices. Additionally, they should have very high dimensions for experiments on large scale platforms, which means that the proposed generation method should be easy to parallelize. Furthermore, since the enormous matrices are generated in parallel, their different slices are already distributed over separate computing units. These data can be used di-

rectly to evaluate the solvers, without having to load the large matrix from the file system. It can save time and increase the efficiency of the applications.

In this paper, we present a Scalable Matrix Generator from Given Spectra (SMG2S) to benchmark the linear/eigenvalue solvers on large-scale platforms. Firstly, it has been implemented in parallel based on PETSc (Portable, Extensible Toolkit for Scientific Computation) for homogenous platforms and PETSc+CUDA (Compute Unified Device Architecture) for multi-GPU heterogeneous machines. Then an open source package[4] is available, with specific communication optimization based on MPI and C++ [13]. Its scalability and ability to maintain the given spectra have been evaluated on different supercomputers.

This paper is organized as follows: Section 2 talks about the related work on the test matrix collections. Section 3 gives an overview of the proposed algorithm for matrix generation. The parallel implementation of SMG2S is discussed in Section 4. In Section 5, we evaluate its strong and weak scalability on two supercomputers. The eigenvalue accuracy of SMG2S is tested with different spectral distributions in Section 6. Finally, we conclude in Section 7.

2 Related Work

It's rare but there are already several efforts to supply test matrix collections. SPARSEKIT [10] implemented by Y. Saad contains various simple matrix generation subroutines. The Galeri package of Trilinos provides to generate simple well-known finite element matrices in parallel. Z. Bai [1] presented a collection of test matrices for developing numerical algorithms for solving nonsymmetric eigenvalue problems. There are also two widely spread matrix providers, the Tim Davis [4] and Matrix Market collections [2]. They both contain many matrices from scientific fields with various mathematical characteristics. But the spectra of matrices in these collections are fixed, and cannot be customized. M.T. Chu [3] provides an overview of the inverse eigenvalues problems concerning the reconstruction of a structured matrix from prescribed spectrum without parallel implementation. A test matrix generation suite with given spectra was introduced by J. Demmel [5] in 1989 to benchmark the routines of dense matrices in LAPACK[5]. Their method uses an orthogonal matrix to transfer a diagonal matrix with given spectrum into dense with the same spectrum, and then transform the dense matrix into an unsymmetric one by Householder transform. This method is not suitable and efficient to test the solvers for the sparse matrix, because it requires $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ storage even for generating a small bandwidth matrix. Furthermore, it was implemented for the shared memory systems rather than larger distributed memory systems, thus it is difficult to generate large-scale test matrices targetting for extreme-scale clusters. That is the motivation for us to propose SMG2S which can generate large-scale non-Hermitian matrices with given spectra in parallel. This method requires much less time and storage, and can be easily parallelized on modern distributed memory systems.

[4] Released package download: https://smg2s.github.io/download.html
[5] Linear Algebra PACKage

3 Matrix Generation Algorithm

In this section, we introduce the proposed matrix generation algorithm. First of all, in Section 3.1, we present a summary of its mathematical framework based on the preliminary theorem proposed by H. Gachlier et al. [6].

3.1 Matrix Generation Method

Theorem 1. *Let's consider a collection of matrices* $M(t) \in \mathbb{C}^{n\times n}$, $n \in \mathbb{N}^*$. *If* $M(t)$ *verifies:*

$$\begin{cases} \dfrac{dM(t)}{dt} = AM(t) - M(t)A, \\ M(t=0) = M_0. \end{cases}$$

Then $M(t)$ *and* M_0 *are similar.* $M(t)$ *has the same eigenvalues as* M_0.

Based on this theorem proposed by H. Gachlier [6], a matrix M_0 with given spectra can be transferred to another one $M(t)$ that satisfies *Theorem* 1 and keeps the spectra of M_0. Due to page limitation, we do not give the definitive proof of this theorem. We propose a matrix generation method by selecting many parameters such as the matrices A and M_0.

Denote a linear operator of matrix M determined by matrix A as $\widetilde{A_A} = AM - MA$, $\forall A \in \mathbb{C}^{n\times n}$, $M \in \mathbb{C}^{n\times n}$, $n \in \mathbb{N}^*$. Here AM and MA are the matrix-matrix multiplication operation of matrices A and M. By solving the differential equation in *Theorem* 1, we can firstly get the formule of $M(t)$ with the exponential operator and then extend it by the *Taylor series formula*:

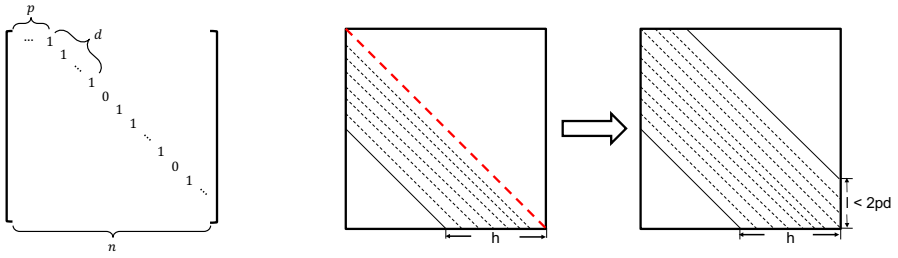
$$\begin{cases} M(t) = e^{\widetilde{A_A}(M_0)t}, \\ M(t) = \displaystyle\sum_{k=0}^{\infty} \frac{t^k}{k!} (\widetilde{A_A})^k (M_0). \end{cases} \tag{1}$$

Through the loop $M_{i+1} = M_i + \frac{1}{i!}(\widetilde{A_A})^i(M_0), i \in (0, +\infty)$, a initial matrix $M_0 \in \mathbb{C}^{n\times n}$ can be transfered into a new non-trivial and non-Hermitian matrix $M_{+\infty} \in \mathbb{C}^{n\times n}$, which has the same spectra but different eigenvectors with M_0.

It is not reasonable to generate a matrix by infinity times of iterations, thus a good selection of matrix A which can make $\widetilde{(A_A)}^i$ tends to $\mathbf{0}$ in limited steps is very necessary. In this paper, we define A as a nilpotent matrix, which means that there exists an integer k such that: $A^i = 0$ for all $i \geq k$. Such k is called the nilpotency of A. In fact, the selection of nilpotent matrix will influence the sparsity pattern of the upper band of the generated matrix.

The exact shape of A is given in Fig. 1a. Inside a $n \times n$ matrix A, its entries are default 0, except on the upper diagonal of the distance p from the diagonal. In this diagonal, its entries start with d consecutive 1 and then a 0, this term repeats until the end. Matrix A can be nilpotent by well choosing the parameters p, d and n. The determination of A to be nilpotent or not is complicated. Firstly, d should be divisible by p, secondly, it should exist the integers e and $f \in \mathbb{N}^*$

that makes $(d+1)e$ be divisible by $i+pf$, for all $i \in 1, 2, \cdots, \frac{d}{p}$. But the cases that $p = 1$ or $p = 2$ are very simple, which can completely fulfil our demands.



(a) Nilpotent matrix (b) Matrix Generation

Fig. 1: (a) gives the nilpotent Matrix, with p off-diagonal offset, d number of continuous 1, and n matrix dimension; (b) shows the matrix generation example.

If $p = 1$, with $d \in \mathbb{N}^*$, or $p = 2$ with $d \in \mathbb{N}^*$ to be even, the nilpotency of A and the upper band's bandwidth of generated matrix are respectively $d+1$ and $2pd$. Obviously, there is another constraint that the matrix size n should be greater or equal to the upper band's width $2pd$. For $p = 2$, if d is odd, the matrix A will not be nilpotent, thus we do not take it into account.

3.2 Algorithm

As shown in Algorithm 1, the procedure of SMG2S is simple. Firstly, it reads an array $Spec_{in} \in \mathbb{C}^n$, as the given eigenvalues. Then it inserts entries in h lower diagonals of the initial matrix M_0 randomly or with selected strategies, and sets its diagonal to be $Spec_{in}$, and scales it with $(2d)!$. Meanwhile, it generates a nilpotent matrix A with the parameters d and p. The final matrix M_t can be generated as $M_t = \frac{1}{(2d)!} M_{2d}$, where M_{2d} is the result after $2d$ times of loop $M_{i+1} = M_i + (\prod_{k=i+1}^{2d} k)(\widetilde{A_A})^i(M_0)$. The slight modification of the loop formula is to reduce the potential rounding errors coming from numerous division operations on modern computer systems.

Algorithm 1 Matrix Generation Method

1: **function** MATGEN($input$:$Spec_{in} \in \mathbb{C}^n$, p, d, h, $output$: $M_t \in \mathbb{C}^{n \times n}$)
2: Insert the entries in h lower diagonals of $M_o \in \mathbb{C}^{n \times n}$
3: Insert $Spec_{in}$ on the diagonal of M_0 and $M_0 = (2d)!M_0$
4: Generate nilpotent matrix $A \in \mathbb{C}^{n \times n}$ with selected parameters d and p
5: **for** $i = 0, \cdots, 2d-1$ **do**
6: $M_{i+1} = M_i + (\prod_{k=i+1}^{2d} k)(\widetilde{A_A})^i(M_0)$
7: $M_t = \frac{1}{(2d)!} M_{2d}$

For M_t, if M_0 is a lower triangular matrix having h non zero diagonals, it will be a band diagonal matrix, whose number of new diagonals in the upper triangular zone will be at most $2pd-1$. Thus the maximal number of the bandwidth of matrix M_t is: $width = h + 2pd - 1$, as in Fig. 1b. In general, researchers

use these matrices to test the iterative methods for sparse linear systems. The h lower diagonals of the initial matrix can be set to be sparse, which ensures the sparsity of the final generated matrix, as shown in Fig. 2. Moreover, the permutation matrix can also be applied to further change the sparsity of the generated matrix.

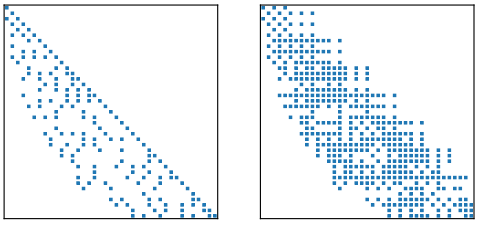


Fig. 2: Matrix Generation Sparsity Pattern Example.

The operation complexity of SMG2S is $max(\mathcal{O}(hdn), \mathcal{O}(d^2n))$. The worst case would require $\mathcal{O}(n^3)$ operations and $\mathcal{O}(n^2)$ memory storage with large d and h. However, if we want to generate a band matrix with a small bandwidth, or if the lower band of M_0 is sparse, it becomes a $\mathcal{O}(n)$ problem with good potential scalability and consuming $\mathcal{O}(n)$ memory storage.

4 Parallel Implementation

In this section, we will introduce parallel implementations of SMG2S on homogeneous and heterogeneous clusters. Firstly, we implemented SMG2S based on PETSc on the CPUs, and based on MPI, CUDA, and PETSc on multi-GPU. We chose PETSc because it provides several ways to verify the generated matrices and the basic operations optimized for different computer architectures. After the PETSc-based SMG2S validation, an open source parallel software with specific optimized communication is also implemented based on MPI and C++.

4.1 Basic Implementation on CPUs

For the initial implementation, we chose PETSc instead of ScaLAPACK because we want to evaluate the solvers for sparse linear systems. The kernels of SMG2S are the sparse matrix-matrix multiplication (SpGEMM) AM and MA, and the matrix-matrix addition (AYPX operation) as $AM - MA$. All the sparse matrices during the generation procedure are stored by the block diagonal Compressed Sparse Row (CSR) format which is supported in default by PETSc. This matrix format keeps separately the block diagonal and off-diagonal parts of a matrix on each process into two sequence CSR format matrices. We use the matrix operations SpGEMM and AYPX provided by PETSc to facilitate the implementation.

4.2 Implementation on mutil-CPU

PETSc does not support SpGEMM and AXPY operations on multiple GPUs, so we implement them based on PETSc data structures, MPI, CUDA and, cuS-

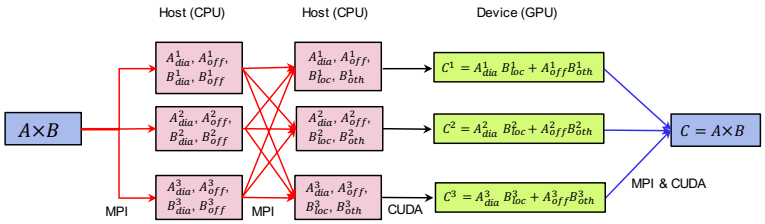


Fig. 3: The structure of a CPU-GPU implementation of SpGEMM, where each GPU is attached to a CPU. The GPU is in charge of the computation, while the CPU handles the MPI communication among processes.

PARSE. The implementation is given in Fig. 3, by an example of $A \times B$. Firstly, same as in PETSc, A and B are stored by the block CSR format, noted as A^i_{dia}, A^i_{off}, B^i_{dia} and B^i_{off} the sequence matrices on process i. Then B^i_{dia} and B^i_{off} are combined together as a novel sequence matrix as B^i_{loc} on each process i. With MPI functionalities, each CPU gather all the remote data of matrix B from the other processes, and construct them to a new sequence matrix B^i_{oth}. These matrices from each process are copied to one attached GPU, and calculate $C^i = A^i_{dia}B^i_{loc} + A^i_{off}B^i_{oth}$. The matrix operations on each GPU device is supported by the cuSPARSE. The final result C can be obtained by gathering all slices C_i from all the devices.

4.3 Specific Optimized Communication Implementation on CPUs

In fact, the parallel SpGEMM kernel's communication can be specifically optimized based on the particular property of nilpotent matrix A. Since A is determined by three parameters p, d and n as we mentioned in Section 3.1, it is not necessary to explicitly implement this parallel matrix. We note this nilpotent matrix as $A(p, d, n)$. Denote $J(i, j)$ the entry in row i and column j of matrix J; $J(i, :)$ all the entries of row i; and $J(:, j)$ all the entries of column j. As shown in Fig. 4, the right-multiplcation $A(p, d, n)$ will cause all the entries of the first $n-p$ columns of M to shift right by an offset p. Denote MA the result gotten by the

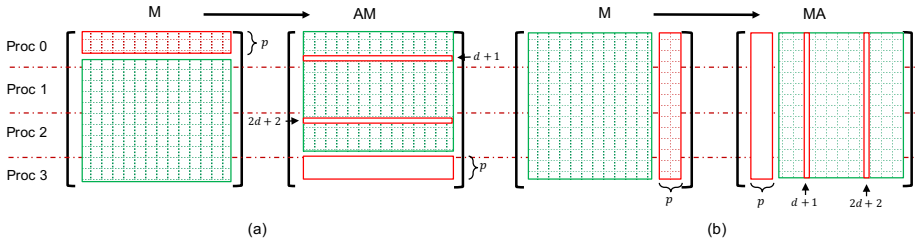


Fig. 4: (a) AM operation; (b) MA operation.

right-multiplying A on M. We have $MA(:,j) = M(:,j-p), \forall j \in p, \cdots, n-1$, and $MA(:,j) = 0, \forall j \in 0, \cdots, p-1$. Similarly, the left-multiplying $A(p,d,n)$ on M will shift up the whole entries of last $n-p$ rows by an offset p. Denote AM the matrix gotten by the left-multiplying A on M. We have $AM(i,:) = M(i+p,:), \forall i \in 0, \cdots, n-p-1$, and $AM(i:,) = 0, \forall i \in p, \cdots, n-1$. Moveover, the parameter d decides that $MA(:, r(d+1)) = 0$ and $AM(r(d+1),:) = 0$ with $r \in 1, \cdots, \lfloor \frac{n}{d+1} \rfloor$.

Algorithm 2 Parallel MPI AM Implementation

```
1: function AM(input: matrix M, matrix row number n, p, d, proc number m;
   output: matrix AM)
2:     Distribute t row blocks M_k of M to MPI process k
3:     for p + 1 ≤ i < t do
4:         for 0 ≤ j < n do
5:             if M(i, j) ≠ 0 then
6:                 AM_k(i − p, j) = M_k(i, j)
7:     for 0 ≤ i < p do
8:         if k ≠ 0 then
9:             isend ith row M_k(i) to k − 1
10:        if k ≠ m − 1 then
11:            irecv ith row M_k(i) from k + 1
12:            AM_k(t − p + i) = M_k(i)
```

Algorithm 3 Parallel MPI MA Implementation

```
1: function MA(input: matrix M, matrix row number n, p, d, proc number m;
   output: matrix MA)
2:     Distribute t row blocks M_k of M to process k
3:     for 0 ≤ i < t do
4:         for p + 1 ≤ j < n do
5:             if M_k(i, j) ≠ 0 then
6:                 MA_k(i, j + p) = M_k(i, j)
```

For the parallel implementation on distributed memory systems, the three parameters p, d and n can be shared by all MPI processes, then operations AM and MA are different from a general parallel SpGEMM. Firstly, the matrix M is one-dimensional distributed by row across m MPI process. As shown in Fig. 4b, for MA, there is no communication inter different MPI processes since the data are moved inside each row. Ensure that $\lfloor \frac{n}{m} \rfloor \geq p$, for AM, the intercommunication of MPI takes place when the MPI process k ($k \in 1, \cdots, m-1$) should send the first p rows of their sub-matrix to the closest previous MPI process numbering $k-1$. The communication complexity for each process is $\mathcal{O}(np)$. When generating the band matrix with low bandwidth b, it tends to be a $\mathcal{O}(bp)$ with $p = 1$ or 2. The MPI-based optimization implementations of AM and MA are respectively given by Algorithm 2 and 3. The communication inter MPI process is implemented by the asynchronous sending and receiving functions. In this algorithm, M_k, MA_k and AM_k imply the sub-matrices on process k with t rows.

The rows and columns of these sub-matrices in Algorithm 2 and 3 are all indexed by the local indices.

The communication-optimized SMG2S is implemented based on MPI and C++. The submatrix on each process is stored in ELLPACK format, using the key-value map containers provided by C++. The key-value map implementation facilitates the indexing and moving of the rows and columns. We did not implement a GPU version of SMG2S with this kind of communication optimization since its core is the data movement among different computing units, which is not well suitable for the multi-GPU architecture.

5 Performance Evaluation

5.1 Hardware Environment

In experiments, we install SMG2S on the supercomputers *Tianhe-2* and *Romeo*. *Tianhe-2* is installed at National Super Computer Center in Guangzhou, China, with 16000 compute nodes. Each node composes 2 Intel Ivy Bridge 12 cores @ 2.2 GHz. *Romeo* is located at University of Reims Champagne-Ardenne, France, which is a heterogeneous system with 130 BullX R421 nodes. Each node composes 2 Intel Ivy Bridge 8 cores @ 2.6 GHz and 2 NVIDIA Tesla K20x GPUs.

5.2 Strong and Weak Scalability Results and Analysis

Table 1: Details for weak scaling and speedup evaluation.
(a) Matrix size for the CPU weak scaling tests on $Tianhe$-2.

CPU number	48	96	192	384	768	1536
matrix size	1×10^6	2×10^6	4×10^6	8×10^6	1.6×10^7	3.2×10^7

(b) Matrix size for the CPU weak scaling on $ROMEO$.

CPU number	16	32	64	128	256
matrix size	4×10^5	8×10^5	1.6×10^6	3.2×10^6	6.4×10^6

(c) Matrix size for the GPU weak scaling and speedup evaluation on $ROMEO$.

CPU or GPU number	16	32	64	128	256
matrix size	2×10^5	4×10^5	8×10^5	1.6×10^6	3.2×10^6

In this section, we will use double-precision real and complex values to evaluate the strong and weak scalability of SMG2S's different implementations on CPU and multiple GPUs. All the test matrices in this paper are generated with the h set to be 10 and d to be 7. The details of the weak scaling experiments are given in Table 1. The matrix size of the strong scaling experiments on $Tianhe-2$ with CPUs, $ROMEO$ with CPUs and $ROMEO$ with GPUs are respectively 1.6×10^7, 3.2×10^6 and 8.0×10^5. The results are given in Fig. 5. The weak

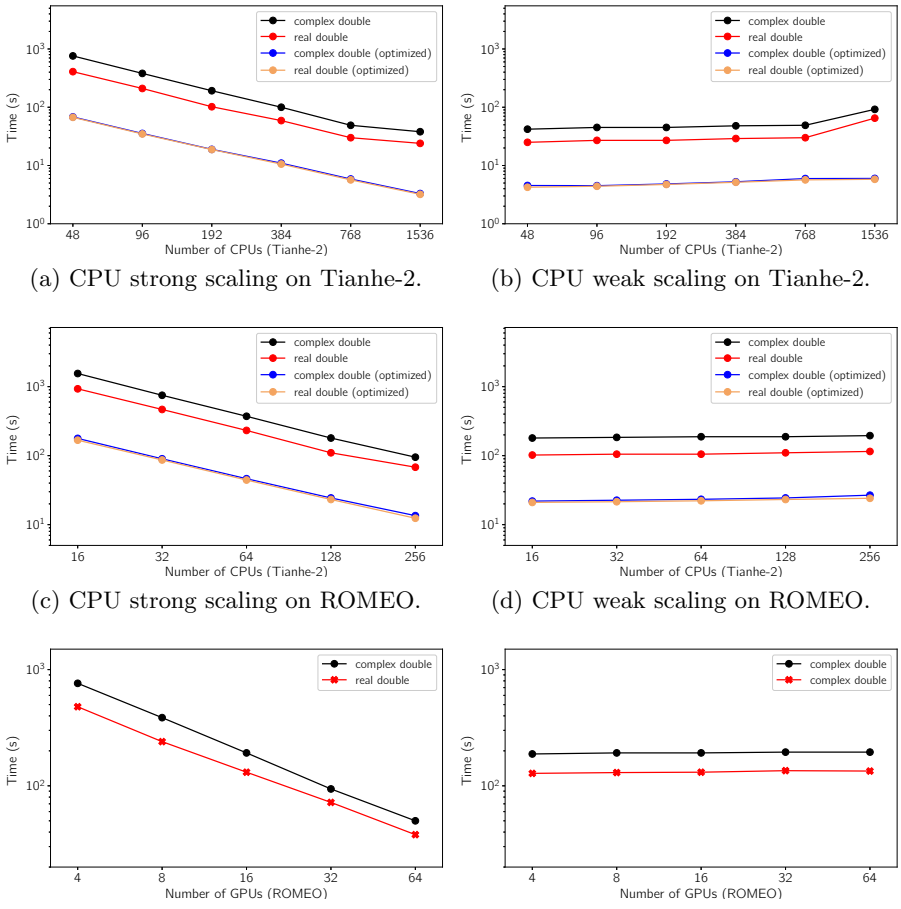
scaling for the PETSc implementation of SMG2S on *Tianhe-2* trends to be bad when MPI processes number is larger than 768, where the communication overhead becomes dominant for computation. But for the communication optimized SMG2S, both the strong and weak scaling perform well when the MPI process number is larger than 768. The experiments show that SMG2S implemented with GPUs can still have good strong and weak scalability. In conclusion, SMG2S has always good strong scaling performance when d and h are much smaller than the dimension of the matrix n, because it turns to be a $\mathcal{O}(n)$ problem. The weak scalability is good enough for most cases. The reason is that the nilpotent matrix A in SpGEMM is simple with not many non-zero elements, therefore there is not enormous communication among different computing units. The weak scalability has its drawback in case that the computing unit number come to be huge for the SMG2S implementation based on PETSc, where the communication overhead become dominant. The special implementation of communication-optimized SMG2S makes his strong and weak scalability better. It is also shown that the double precision complex type matrix generation takes almost two times time over the double precision real type for the basic SMG2S implementation, but the time consumption of complex and real type matrix generation of optimized SMG2S seems similar. The reason is that there is no numerical values multiplication anymore in the optimized implementation of SMG2S.

5.3 Speedup Results and Analysis

The speedup of both SMG2S on multi-GPU and communication-optimized SMG2S on the CPUs compared with the PETSc-based implementation on CPU are also tested on *Romeo*. According to the previous evaluation that complex and real value types have always good scalability, we select the double precision complex values for the speedup evaluation. The details of experiments are also given in Table 1c. The results are shown in Fig. 6. We can find that the GPU version of SMG2S has almost 1.9× speedup over the PETSc CPU version. The communication-optimized SMG2S on CPUs has about 8× speedup over the basic PETSc CPU version.

6 Accuracy Verification

In the last section, we presented the good parallel performance of SMG2S, then it is necessary to verify if the generated matrices are able to keep the given spectra with enough accuracy. Generally, the iterative eigenvalue solvers such as the Arnoldi or other Krylov methods [9] are applied to approximate the dominant eigenvalues. But the accuracy verification is an opposite case. Now there exists a value, we want to check if it is an eigenvalue of a given matrix. These iterative methods cannot directly and efficiently deal with this kind of verification. In this section, we present a method for the accuracy verification using the Shifted Inverse Power method, which was easily implemented in parallel.



(a) CPU strong scaling on Tianhe-2.

(b) CPU weak scaling on Tianhe-2.

(c) CPU strong scaling on ROMEO.

(d) CPU weak scaling on ROMEO.

(e) GPU strong scaling on ROMEO.

(f) GPU weak scaling on ROMEO.

Fig. 5: Strong and weak scaling results of SMG2S on different platforms. A base 2 logarithmic scale is used for X-axis, and a base 10 logarithmic scale for Y-axis.

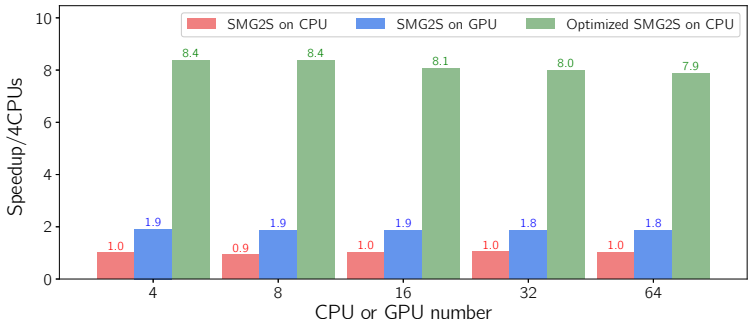


Fig. 6: Weak scaling speedup comparison of GPUs on ROMEO.

6.1 Verification Method

The Power method is an algorithm to approximate the greatest eigenvalue. Meanwhile, the Inverse Power method is a similar iterative algorithm to find the smallest eigenvalue. The middle eigenvalues can be obtained by the Shifted Inverse Power method [7]. The Shifted Inverse Power method is able to compute the eigenpair whose eigenvalue is the nearest to a given value in a few iterations.

In details, for checking if the given value λ is the eigenvalue of a matrix, we select a shifted value σ which is close enough to λ. An eigenpair (λ', v') with the relation $Av' = \lambda' v'$ can be approximated in very few steps by Shifted Inverse Power method, with λ' is the closest eigenvalue to σ. Since σ is very close to λ, it should be that λ and λ' are the same eigenvalue of a system, and v' should be the eigenvector related to λ. In reality, even if the computed eigenvalue is very close to the true one, the related eigenvector may be quite inaccurate. For the right eigenpairs, the formula $Av' \approx \lambda v'$ should be satisfied. Based on this relation, we define the relative error as Formula (2) to quantify the accuracy.

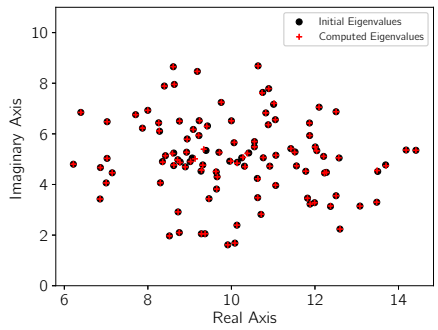
$$error = \frac{||Av' - \lambda v'||_2}{||Av'||_2} \tag{2}$$

If $\lambda' = \lambda$, this *error* should be 0, if not, this generated matrix will not have an exact eigenvalue as λ. In real experiments, the exact solution cannot always be guaranteed with the arithmetic rounding errors of floating operations during the generation. A threshold could be set for accepting it or not.

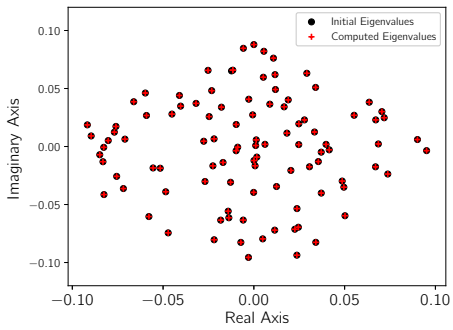
6.2 Experimental Results

In the experiments, we test the accuracy of SMG2S with four selected cases among the various tests of different spectral distributions. Fig. 7a and Fig. 7b are cases of clustered eigenvalues with different scales. Fig. 7c is a special case with the dominant part of eigenvalues clustered in a small region. Fig. 7d is a case that composes the conjugate and closest pair eigenvalues. These figures compare the difference between the given spectra (noted as initial eigenvalues in the figures) and the approximated ones (noted as computed eigenvalues) by the Shifted Inverse Power Method. Clearly, the matrices generated by SMG2S can keep almost all the given eigenvalues in the four cases even they are very clustered and close. The acceptance threshold is set to be 1.0×10^{-3}.

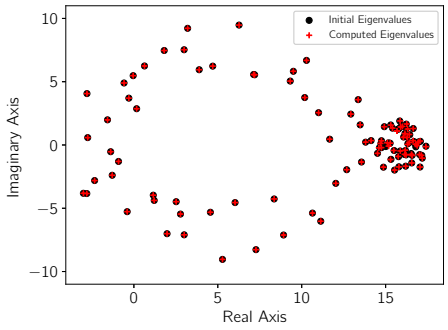
This acceptance for cases of Fig. 7a, Fig. 7b, Fig. 7c and Fig. 7d are respectively 93%, 100%, 94% and 100%. The maximum *error* for them are respectively 3×10^{-2}, 7×10^{-5}, 3×10^{-2} and 3×10^{-7}. After the tests, we conclude that SMG2S is able to keep accurately the given spectra even for the very clustered and closest eigenvalues. In some cases, a very little number of too clustered eigenvalues may result in the inaccuracy of given ones, but in general, the generated matrix can fufil the need to evaluate the linear system and eigenvalue solvers.



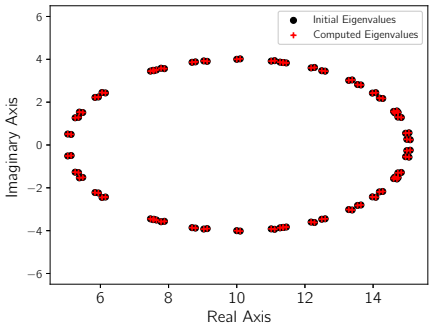
(a) Clustered Eigenvalues I: acceptance = 93%, max $error = 2 \times 10^{-2}$



(b) Clustered Eigenvalues II: acceptance = 100%, max $error = 7 \times 10^{-5}$



(c) Dominant Clustered Eigenvalues: acceptance = 94%, max $error = 3 \times 10^{-2}$



(d) Conjugate and Closest Eigenvalues: acceptance = 100%, max $error = 3 \times 10^{-7}$

Fig. 7: Verification using Different Types of Spectra.

6.3 Arithmetic Precision Analysis

Any floating operations will introduce rounding errors, which is not negligible for the generation of large matrices. Regarding the non-Hermitian matrix, its eigenvalues may be extremely sensitive to perturbation. This sensibility is bounded by $bound(\lambda) \leq ||E||_2 Cond(\lambda)$, with $Cond(\lambda)$ the condition number of related eigenvalue λ which can be excessively high for the non-Hermitian matrix and $||E||_2$ the Euclidean norm of errors [12]. One solution is to use the integer values for the matrix generation, since only integers and the operations $+$, $-$, and $\times$ on the microprocessor can make absolutely exact computations. As shown in Algorithm 1, most of the operations in SMG2S are $+$, $-$ and $\times$, except the step 7 with a division operation. Without step 7, we could introduce a special SMG2S fully using integers to avoid the risks of rounding errors. The spectra of the generated matrix will be $(2d)!$ times scaled up over the given one.

7 Conclusion and Pespectives

In this paper, we presented a scalable matrix generator and its parallel implementation on homogeneous and heterogeneous clusters. It allows generating large-scale non-Hermitian matrices with customized eigenvalues to evaluate the impact of spectra on the linear/eigenvalue solvers on large-scale platforms. The experiments proved its good scalability and the ability to keep the given spectra with acceptable accuracy. For large matrices, the I/O operation on supercomputers is always a bottleneck even with the high bandwidth. The matrices generated in parallel by SMG2S, with data already allocated on different processes, can be used directly to evaluate the numerical methods without concerning the I/O operation. The interfaces of SMG2S to C, Python and scientific libraries PETSc and Trilinos are provided. Interface to Fortran will be implemented in future.

References

1. Bai, Z., Day, D., Demmel, J., Dongarra, J.: A test matrix collection for non-hermitian eigenvalue problems. Dept. of Mathematics **751**, 40506–0027 (1996)
2. Boisvert, R.F., Pozo, R., Remington, K., Barrett, R.F., Dongarra, J.J.: Matrix market: a web resource for test matrix collections. In: Quality of Numerical Software, pp. 125–137. Springer (1997)
3. Chu, M.T., Golub, G.H.: Structured inverse eigenvalue problems. Acta Numerica **11**, 1–71 (2002)
4. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS) **38**(1), 1 (2011)
5. Demmel, J., McKenney, A.: A test matrix generation suite. In: Courant Institute of Mathematical Sciences. Citeseer (1989)
6. Galicher, H., Boillod-Cerneux, F., Petiton, S., Calvin, C.: Generate very large sparse matrices starting from a given spectrum. in lecture notes in computer science, 8969, springer (2014)
7. Hernandez, V., Roman, J., Tomas, A., Vidal, V.: Single vector iteration methods in slepc. Scalable Library for Eigenvalue Problem Computations (2005)
8. Liesen, J., Strakos, Z.: Krylov subspace methods: principles and analysis. Oxford University Press (2013)
9. Petiton, S.G.: Parallel subspace method for non-hermitian eigenproblems on the connection machine (cm2). Applied Numerical Mathematics **10**(1), 19–35 (1992)
10. Saad, Y.: Sparsekit: a basic tool kit for sparse matrix computation (version2), university of illinois (1994)
11. Saad, Y.: Least squares polynomials in the complex plane and their use for solving nonsymmetric linear systems. SIAM Journal on Numerical Analysis **24**(1), 155–169 (1987)
12. Saad, Y.: Numerical Methods for Large Eigenvalue Problems: Revised Edition. SIAM (2011)
13. Wu, X.: SMG2S Manual v1.0. Maison de la Simuation, France (2018), https://smg2s.github.io/files/smg2s-manual.pdf
14. Wu, X., Petiton, S.G.: A distributed and parallel asynchronous unite and conquer method to solve large scale non-hermitian linear systems. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. pp. 36–46. HPC Asia 2018, ACM, New York, NY, USA (2018)