A parallel structured divide-and-conquer algorithm for symmetric tridiagonal eigenvalue problems

Xia Liao[1], Shengguo Li[1], Yutong Lu[2,3], and Jose E. Roman[4]

[1] *College of Computer Science, National University of Defense Technology, Changsha, China*
[2] *National Supercomputer Center in Guangzhou, China*
[3] *School of Data and Computer Science, Sun Yatsen University, Guangzhou, China, 510006*
[4] *D. Sistemes Informàtics i Computació, Universitat Politècnica de València, Camí de Vera s/n, 46022 València, Spain*

July 2020

Abstract

In this paper, a parallel structured divide-and-conquer (PSDC) eigensolver is proposed for symmetric tridiagonal matrices based on ScaLAPACK and a parallel structured matrix multiplication algorithm, called PSMMA. Computing the eigenvectors via matrix-matrix multiplications is the most computationally expensive part of the divide-and-conquer algorithm, and one of the matrices involved in such multiplications is a rank-structured Cauchy-like matrix. By exploiting this particular property, PSMMA constructs the local matrices by using generators of Cauchy-like matrices without any communication, and further reduces the computation costs by using a structured low-rank approximation algorithm. Thus, both the communication and computation costs are reduced. Numerical results show that both PSMMA and PSDC are highly scalable and scale to 4096 processes at least. PSDC has better scalability than PHDC that was proposed in [J. Comput. Appl. Math. 344 (2018) 512–520] and only scaled to 300 processes for the same matrices. Comparing with `PDSTEDC` in ScaLAPACK, PSDC is always faster and achieves 1.3x–1.6x speedup for some matrices with few deflations. PSDC is also comparable with ELPA, with PSDC being faster than ELPA when using few processes and a little slower when using many processes.

1 Introduction

Computing the eigendecomposition of a symmetric tridiagonal matrix is an important linear algebra problem and is widely used in many fields of science and engineering. It is usually solved by divide-and-conquer (DC) [1, 2], QR [3], MRRR [4], and some other methods. The DC algorithm is now the default method in LAPACK [5] and ScaLAPACK [6] when the eigenvectors are required. DC is usually very efficient in practice, requiring only $O(N^{2.3})$ flops on average [7], but its complexity can still be $O(N^3)$ for some matrices with few deflations. The performance of an algorithm not only depends on the number of floating point operations, but also other ingredients such as communication and data movements, etc. Some communication-avoiding algorithms have been developed recently [8, 9]. In this work, we aim to accelerate the parallel DC algorithm on distributed memory machines by reducing both the *computation* complexity and *communication* complexity. Our algorithm can be much faster than the ScaLAPACK routine `PDSTEDC` for those difficult matrices, and can scale to thousands of processes. Numerical results are included in section 4.

It is known that some Cauchy-like matrices with off-diagonally low-rank properties appear in the DC algorithm [1, 2], which can be approximated by hierarchically semiseparable (HSS) matrices [10, 11]. Then, the worst case complexity of DC can be reduced from $O(N^3)$ to $O(N^2r)$, where r is a modest number and is usually much smaller than a large N, see [12]. This technique was extended for the bidiagonal and banded DC algorithms for the SVD problem on a shared memory multicore platform in [13, 14]. For distributed memory machines, a parallel DC algorithm is similarly proposed in [15] by using STRUMPACK (STRUctured Matrices PACKage) [16], which provides some distributed parallel HSS algorithms. The accelerated DC proposed in [15] was called *PHDC*. However, numerical results show that for the simple matrix-matrix multiplication operations, STRUMPACK is not as scalable as `PDGEMM`, and may become slower than `PDGEMM` when using 300 or more processes on Tianhe-2 supercomputer. See [15] for details.

Instead of using HSS, this work exploits a much simpler type of rank-structured matrix, called BLR (Block low-rank format [17]). Compared to HSS, BLR abandons the hierarchy but compresses the off-diagonal blocks. It loses the near-linear complexity of other hierarchical matrices such as $\mathcal{H}$-matrix [18], $\mathcal{H}^2$-matrix [19], and HSS matrix. Because of its simple structure, BLR is easy to implement in parallel, and often improves the scalability of corresponding algorithms, see [17, 20] for more details.

In this paper, we propose a parallel structured matrix-matrix multiplication algorithm for Cauchy-like matrices, which will be named *PSMMA*. It exploits the off-diagonal low-rank property of matrices like BLR, and it can further reduce the communication cost by constructing local submatrices using the *generators*, which will be explained in section 3. Our main contributions include the following:

- We propose a parallel structured matrix multiplication algorithm (PSMMA) for structured matrices including Cauchy-like, Toeplitz, Hankel, Vandermonde, etc. PSMMA can reduce both the communication cost and computation cost by using low-rank approximations. To the best of the authors' knowledge, none of the matrix multiplication algorithms has been developed to reduce the communication cost by exploiting the structure of matrices.

- PSMMA works for matrices both in the block cyclic data distribution (BCDD) form (like ScaLAPACK) and block data distribution (BDD) form (2D block partitioning). It also works for general process grids.

- Combining PSMMA with the DC algorithm in ScaLAPACK, we propose a parallel structured DC algorithm (PSDC), which can be much faster than `PDSTEDC` in ScaLAPACK. PSDC is also comparable with ELPA [21].

The process of PSMMA is similar to Cannon [22] and Fox [23] algorithms. However, PSMMA works for matrices in BCDD form and works for any rectangular process grids. From this perspective, PSMMA is more like PUMMA [24], a generalized Fox algorithm. PSMMA is more efficient than PUMMA for structured matrices and details are shown in section 3.1.2. It has three advantages compared with PUMMA. One advantage is that PSMMA constructs the required submatrix locally by using the generators without communication and thus requires less *communication*. Another one is that PSMMA combines with low-rank approximations and therefore the *computation complexity* is also reduced. The third one is that PSMMA requires less workspace and the size of local matrix multiplications is also larger than PUMMA. In this paper, SRRSC [13, 25] is used to compute the low-rank approximations of Cauchy-like matrices in PSMMA, which only requires linear storage. Note that `PDGEMM` implements an algorithm similar to SUMMA [26]. Compared with SUMMA [26], which is based on the outer product form of matrix multiplication, PSMMA can naturally exploit the off-diagonal low-rank property of matrices.

By incorporating PSMMA into the DC algorithm in ScaLAPACK [7], we obtain a highly scalable DC algorithm, which has much better scalability than the previous PHDC algorithm [15]. To distinguish from PHDC, we call the newly proposed algorithm *parallel structured DC* algorithm (PSDC). Numerical results show that PSDC is always faster than `PDSTEDC` in ScaLAPACK, and scales to 4096 processes at least. That is because PSDC requires both less computations and communications than `PDSTEDC`. The speedups of PSDC over `PDSTEDC` can be up to 1.3x-1.6x for some matrices with dimension $30,000$ on Tianhe-2 supercomputer. Note that PHDC in [15] can only scale to 300 processes for the same matrices.

The remaining sections of this paper are organized as follows. Section 2 introduces the DC algorithm, the SRRSC algorithm for constructing low-rank approximations of Cauchy-like matrices, and some classical parallel matrix multiplication algorithms. Section 3 presents the newly proposed parallel matrix multiplication algorithm, PSMMA, and describes the implementation details of PSDC, which combines PSMMA with the parallel tridiagonal DC algorithm in ScaLAPACK. All the numerical results are reported in section 4, and some future works are included in section 4.2. Conclusions are drawn in section 5.

2 Preliminaries

Assume T is a symmetric tridiagonal matrix,

$$T = \begin{bmatrix} a_1 & b_1 & & & \\ b_2 & \ddots & \ddots & & \\ & \ddots & a_{N-1} & b_{N-1} \\ & & b_{N-1} & a_N \end{bmatrix}. \tag{1}$$

We briefly introduce some formulae of Cuppen's divide-and-conquer algorithm [1, 7]. The ScaLAPACK routine also implements this version of DC algorithm [7] based on rank-one update.

ALGORITHM 1: DC(T, Q, Λ) algorithm for computing the eigendecomposition of a symmetric tridiagonal matrix

```
Input: $T \in \mathbb{R}^{N\times N}$
Output: eigenvalues $\Lambda$, eigenvectors $Q$
if the size of $T$ is small enough then
    apply the QR algorithm and compute $T = Q\Lambda Q^T$ ;
    return $Q$ and $\Lambda$;
else
    form $T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + b_k v v^T$;
    call DC$(T_1, Q_1, \Lambda_1)$;
    call DC$(T_2, Q_2, \Lambda_2)$;
    form $M = D + b_k u u^T$ from $Q_i, \Lambda_i$ and $v$ $(i = 1, 2)$, where $D = \mathrm{diag}(\Lambda_1, \Lambda_2)$;
    find eigenvalues $\Lambda$ and eigenvectors $\widehat{Q}$ of $M$;
    compute the eigenvectors of $T$ as $Q = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \widehat{Q}$;
    return $Q$ and $\Lambda$;
end
```

Firstly, T is decomposed into the sum of two matrices,

$$T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + b_k v v^T, \tag{2}$$

where $T_1 \in \mathbb{R}^{k\times k}$, b_k is the off-diagonal element at the kth row of T and $v = [0, \ldots, 1, 1, \ldots, 0]^T$ with ones at the kth and $(k+1)$th entries. If $T_1 = Q_1 \Lambda_1 Q_1^T$ and $T_2 = Q_2 \Lambda_2 Q_2^T$, then T can be written as

$$T = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \left(\begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix} + b_k u u^T \right) \begin{bmatrix} Q_1^T & \\ & Q_2^T \end{bmatrix}, \tag{3}$$

where $u = \begin{bmatrix} Q_1^T & \\ & Q_2^T \end{bmatrix} v = \begin{bmatrix} \text{last col. of } Q_1^T \\ \text{first col. of } Q_2^T \end{bmatrix}$.

Since Q_1 and Q_2 are orthogonal matrices, the problem is reduced to computing the spectral decomposition of a diagonal plus rank-one matrix,

$$M \equiv D + b_k u u^T = \widehat{Q} \Lambda \widehat{Q}^T, \tag{4}$$

where $D = \operatorname{diag}(\Lambda_1, \Lambda_2)$ is a diagonal matrix, Λ is a diagonal matrix whose diagonal elements are the eigenvalues of matrix M, and $\widehat{Q}$ is the eigenvector matrix of M. Then, the eigenvector matrix of T is computed as

$$\begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \widehat{Q}. \tag{5}$$

The eigenvalues λ_i of $D + b_k u u^T$ are the roots of the secular equation

$$f(\lambda) = 1 + b_k \frac{u_k^2}{d_k - \lambda} = 0, \tag{6}$$

where d_k is the kth diagonal entry of D, u_k is the kth component of u. Then, the eigenvector is computed as

$$\hat{q}_i = (D - \lambda_i I)^{-1} u. \tag{7}$$

The main observation of works [15, 12] is that

$$\widehat{Q} = \left(\frac{u_i v_j}{d_i - \lambda_j} \right)_{i,j}, \tag{8}$$

where $v_j = 1 / \sqrt{\sum_{k=1}^{N} \frac{u_k^2}{(d_k - \lambda_j)^2}}$ is a Cauchy-like matrix, and the vectors $u, v \in \mathbb{R}^N$ and $d = (d_1, \cdots, d_N)^T, \lambda = (\lambda_1, \cdots, \lambda_N)^T$ are called *generators*.

The whole classical DC algorithm is shown in Algorithm 1. The main computational task of DC lies in computing the eigenvectors via matrix-matrix multiplications (MMM) (5), which costs $O(N^3)$ flops. Since $\widehat{Q}$ is a Cauchy-like matrix and off-diagonally low-rank, MMM (5) can be accelerated by using HSS matrix algorithms, and the computational complexity can be reduced significantly, see [15, 12] for more details. The aim of this work is not only to reduce the computation cost of MMM (5) but also its communication cost in the distributed memory environment. For simplicity, we do not consider *deflation* in (5). About the *deflation* process, we refer the interested readers to [1, 27] and section 3.2.

2.1 SRRSC low-rank approximation

A novel low-rank approximation method for Cauchy-like matrix is proposed in [13, 25], which only requires linear storage. For completeness, this method is introduced briefly in this section, which only works on the generators.

Assume that A is an $n \times n$ Cauchy-like matrix, $A = (\frac{u_i v_j}{d_i - w_j})_{i,j}$. The following factorization is called the kth *Schur complement factorization* of A,

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & \\ A_{21} & A^{(k)} \end{bmatrix} \begin{bmatrix} I & Z^{(k)} \\ & I \end{bmatrix}, \tag{9}$$

where $A_{11} \in \mathbb{R}^{k\times k}$ and $A^{(k)}$ is called the kth Schur complement.

One good property of a Cauchy-like matrix is that its kth Schur complement $A^{(k)}$ is also Cauchy-like [28, 29], and its generators can be computed recursively as follows.

Theorem 1 *The kth Schur complement $A^{(k)}$ satisfies*

$$D_k A^{(k)} - A^{(k)} W_k = u^{(k)}(k+1:n) \cdot v^{(k)T}(k+1:n),$$

with $D_{k+1} = \mathrm{diag}(d_{k+1}, \cdots, d_n)$ and $W_{k+1} = \mathrm{diag}(w_{k+1}, \ldots, w_n)$. Then

$$u^{(k)}(k+\ell) = u^{(k-1)}(k+\ell) \cdot \frac{d_{k+\ell} - d_k}{d_{k+\ell} - w_k}, \tag{10}$$

$$v^{(k)}(k+\ell) = v^{(k-1)}(k+\ell) \cdot \frac{w_{k+\ell} - w_k}{w_{k+\ell} - d_k}, \tag{11}$$

with $1 \le \ell \le n-k, k \ge 1$, $A^{(0)} = A$, $u^{(0)} = u$ and $v^{(0)} = v$.

Corollary 1 *For $1 \le \ell \le n-k, k \ge 1$, the generators of $A^{(k)}$ are*

$$u^{(k)}(k+\ell) = \prod_{j=1}^{k} \frac{d_{k+\ell} - d_j}{d_{k+\ell} - w_j} \cdot u^{(0)}(k+\ell), \tag{12}$$

$$v^{(k)}(k+\ell) = \prod_{j=1}^{k} \frac{w_{k+\ell} - w_j}{w_{k+\ell} - d_j} \cdot v^{(0)}(k+\ell). \quad \square \tag{13}$$

It is easy to prove that $Z^{(k)}$ is also Cauchy-like, and its generators can also be computed recursively.

Theorem 2 *The generators of $Z^{(k)}$ satisfy the displacement equation*

$$W_1^{(k)} Z^{(k)} - Z^{(k)} W_2^{(k)} = y^{(k)}(1:k) \cdot v^{(k)}(k+1:n)^T,$$

where $W_1^{(k)} = \mathrm{diag}(w_1, \ldots, w_k)$ and $W_2^{(k)} = \mathrm{diag}(w_{k+1}, \ldots, w_n)$, and

$$y^{(k)}(i) = \prod_{1 \le j \le k, j \ne i} \frac{d_j - w_i}{w_j - w_i} \cdot \frac{d_i - w_i}{v^{(0)}(i)}, \tag{14}$$

and $v^{(k)}(k+1:n)$ are computed recursively by equation (13). Furthermore,

$$y^{(k)}(i) = \begin{cases} y^{(k-1)}(i) \cdot \frac{d_k - w_i}{w_k - w_i}, & \text{if } 1 \le i \le k-1, \\ \prod_{j=1}^{k-1} \frac{w_k - d_j}{w_k - w_j} \cdot \frac{d_k - w_k}{v^{(0)}(k)}, & \text{if } i = k. \quad \square \end{cases}$$

Since permutation does not destroy the structure of Cauchy-like matrices, we can permute generators $u^{(k)}, v^{(k)}, d$ and w to make the first entry of $A^{(k)}$, $A^{(k)}(1,1) = \frac{u^{(k)}(k+1)v^{(k)}(k+1)}{d_{k+1}-w_{k+1}}$, large. A complete pivoting strategy could be used. Some efficient pivoting strategies have been proposed in [29] and [30]. We used the pivoting strategy proposed in [13].

If the entries of $A^{(k)}$ are negligible, and then by ignoring $A^{(k)}$, we get a low-rank approximation to A,

$$AP \approx A(1:n, \mathscr{T}) \begin{bmatrix} I & Z^{(k)} \end{bmatrix}, \tag{15}$$

where $\mathscr{T} = \{i_1, i_2, \ldots, i_k\} \subset \{1, 2, \ldots, n\}$ and P is a permutation matrix which records the column permutations during the pivotings. To be more specific, $A(1:n, \mathscr{T})$ consists of a subset of columns of A, and from Theorem 2,

$$Z_{ij}^{(k)} = \frac{u^{(k)}(k+i)y^{(k)}(j)}{w(i) - w(k+j)}, \tag{16}$$

where w is the array after permutation.

2.2 Parallel Matrix-Matrix multiplications

Matrix-matrix multiplication is a very important computational kernel of many scientific applications. In this subsection, we briefly introduce some parallel matrix multiplication algorithms, which compute $C = A \times B$.

Cannon algorithm [22] was the first efficient algorithm for parallel matrix multiplication providing theoretically optimal communication cost. However it requires the process grid to be square, which limits its practical usage. Fox algorithm proposed in [23] has the same problem. The PUMMA algorithm [24] is a generalized Fox algorithm, and it works for a general $P \times Q$ processor grid. PUMMA was designed for ScaLAPACK and used the block-cyclic distributed matrix data structure.

The current version of ScaLAPACK implements SUMMA [26], which was proposed in the mid-1990s and designed for a general processor grid. It also uses the block-cyclic distributed matrix data structure. It implements the outer product form of matrix multiplication, and allows to pipeline them. PUMMA implements the inner product form, and requires the largest possible matrices for computation and communication. Some more efficient matrix multiplication algorithms have been proposed recently, like 2.5D algorithm [31], CARMA [32], CTF [33], and COSMA [34], and many others. Since they are not quite related to this current work, we do not attempt to give a complete literature review.

PUMMA is more appropriate for the rank-structured matrices than SUMMA, since the large off-diagonal blocks can be compressed by low-rank approximations. So are Cannon and Fox algorithms. It is not obvious for SUMMA to exploit the rank-structured property of the input matrices.

In this paper we propose a parallel structured matrix multiplication algorithm, which is called PSMMA for short. We assume at least one of the two matrices is a structured matrix. By 'structured matrices' we mean those matrices which can be expressed using $O(n)$ parameters where n is the dimension of the matrix, e.g. Cauchy-like, Toeplitz, Hankel, and Vandermonde matrices [35]. PSMMA can be based on block-cyclic distributed data (BCDD) structure like PUMMA and ScaLAPACK, and it can also be based on block distributed data (BDD) structure like Cannon or Fox algorithms. As shown later, the second approach is more efficient to exploit the off-diagonal low-rank structure. Its drawback is that it requires to redistribute the matrix from the BCDD form to BDD form, since the matrices are initially stored in BCDD form in ScaLAPACK routines. The PSMMA in BCDD form fits well with ScaLAPACK routines and does not need any data redistribution.

By exploiting the special structure of the matrix, PSMMA can reduce both the computation and communication costs. To the best of the authors' knowledge, this work is the first one proposing a reduction of the communication cost of matrix multiplication algorithms by exploiting the matrix structure.

3 Algorithm Proposed

The main contributions of this paper consist of two parts. First, we design a parallel structured matrix multiplication algorithm for structured matrices in section 3.1, which is called PSMMA and can reduce both the computation and communication costs.

Secondly, section 3.2 shows how to combine PSMMA with the parallel tridiagonal DC algorithm in ScaLAPACK. It illustrates how to modify several routines in ScaLAPACK in order to use PSMMA. The parallel structured DC algorithm is called PSDC, and its whole procedure is summarized in Algorithm 3. A cartoon is included in Fig. 4 to show the whole workflow of PSDC.

3.1 PSMMA for structured matrices

In this subsection, we introduce PSMMA to compute $C = A \times B$, where $A \in \mathbb{R}^{m\times k}$ is a general matrix, $B \in \mathbb{R}^{k\times n}$ is a structured matrix, and its entries can be represented by $O(n+k)$ parameters. The case that A is a structured matrix and B is a general matrix is similar. When both A and B are structured matrices, all operations can be performed locally without any communication, which will be explained later. In the following sections, we assume A is stored in block-cyclic form and B is stored in its generators.

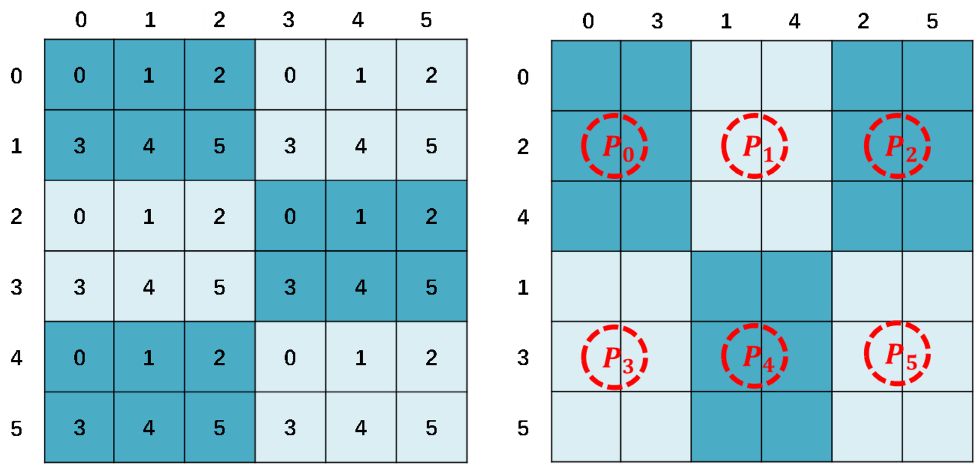
Suppose the matrix A has M block rows and K block columns, and the matrix B has K block rows and N block columns. Block (I,J) of C is then computed by

$$C(I,J) = \sum_{\ell=0}^{K-1} A(I,\ell)\cdot B(\ell,J), \tag{17}$$

where $I = 0,1,\cdots,M-1$, $J = 0,1,\cdots,N-1$. Cannon and Fox algorithms initially considered only the case of matrices in which each processor contains a single row or a single column of blocks [22, 23]. PUMMA considered the matrix multiplication algorithms with BCDD form [26]. Fig. 1 shows a 6×6 block matrix stored in a 2×3 process grid. It is easy to see that the matrix in the block cyclic form is obtained from the original matrix by performing row and column block permutations.

Since the matrix B can be represented by its generators, any submatrices of B can be formed easily. This fact enables us to treat the submatrices of A in each process as a whole continuous block, and we only need to construct the proper submatrices of B correctly. This is our **main observation**. After discovering this fact, the proposal of the PSMMA algorithm becomes very natural and simple, just following equation (17). The whole procedure is shown in Algorithm 2.

To illustrate the algorithm from the process point of view, we show how the submatrices of C stored at process P_0 (located at position $(0,0)$ in the process grid) are computed for the matrix shown in Fig. 1(b). This consists of three steps, and the process is depicted in Figure 2. The column indexes of B are fixed, and its row indexes are determined by the column indexes of A. After each step, matrix A would be shifted leftward, and the local matrix A on process P_0 is updated by the matrix on process P_1. As shown in Fig. 1(b), the local matrix A of process P_0 at step 1 is updated by that of process P_1, which is located at the right of process P_0. After process P_0 has received the matrix A from all other processes, the algorithm stops.



(a) Matrix point-of-view (b) Process point-of-view

Figure 1: A matrix with 6×6 blocks is distributed over a 2×3 process grid.

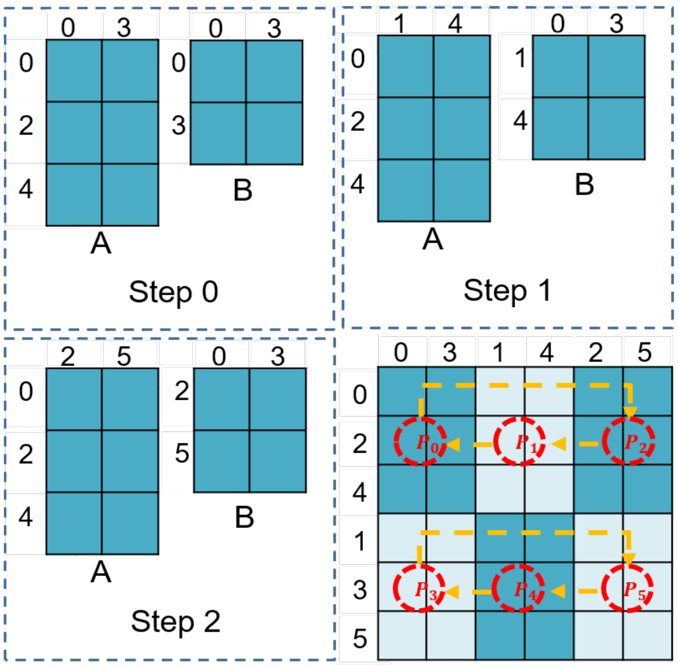


Figure 2: The process for computing the submatrices of C located at process $(0, 0)$.

```
ALGORITHM 2: PSMMA for a structured matrix $B$
Input: $A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}$, where $A$ is distributed over a $p \times q$ process grid, $B$ is a
       structured matrix and all processes have a copy of its generators;
Output: $C = A \times B$.
  1. Each process constructs the column indexes (CIndex) of matrix $B$ based on its process
     column in the process grid;
  2. Set $C = 0$.
  3. do $k = 0, q-1$
       (a) For each process $(i, j)$, construct the column indexes (RIndex) of matrix $A$ based on
           the process column $\mod (j+k, q)$;
       (b) Calculate the required $B$ subblock $B$(RIndex,CIndex), and construct its low-rank
           approximation (if needed) by using its generators, $B \approx U_B V_B$;
       (c) Multiply the copied $A$ subblock with the currently residing $B$ subblock:
           $C = C + (A \cdot U_B) \cdot V_B$;
       (d) Shift matrix $A$ leftward cyclically along each process row;
     end do
```

Algorithm 2 works both for block cyclic data distribution and block data distribution. It only depends on the distribution of A to determine the row indexes of local matrix B. In step 3(b) of Algorithm 2, we construct a low rank approximation only when the local matrix B is probably low rank. For the tridiagonal DC algorithm in section 2, matrix B is a Cauchy-like matrix and we can check whether the intersection of *CIndex* and *RIndex* is empty or not. If the intersection is empty, the B submatrix is probably numerically low-rank. Otherwise, the B submatrix is probably full rank. For Cauchy-like matrices, we use SRRSC discussed in section 2.1 to construct a low-rank approximation to matrix B.

Remark 1. The PSMMA algorithm introduced in this section is also suitable for other structured matrices, such as Toeplitz, Vandermonde, and DFT (Discrete Fourier Transform) matrices. Some results will be shown in our future works.

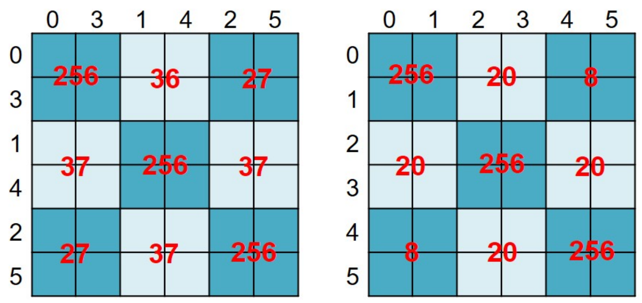
Remark 2. Only step *3(d)* of Algorithm 2 requires point-to-point communications. It can be overlapped with other computations if implemented carefully. During our numerical experiments, we did not implement this techniques.

3.1.1 Storage form affects the off-diagonally low-rank property

For our problem, the eigenvector matrix $\widehat{Q}$ in equation (4) is a Cauchy-like matrix, see equation (8). It is further off-diagonally low rank, since $\{d_i\}_{i=1}^N$ and $\{\lambda_i\}_{i=1}^N$ are interlacing. The storage form of matrix A affects the low-rank property of matrix B in Fig. 2. We use the following example to show the main points.

Example 0. Assume that matrix A is defined as $A_{ij} = \frac{u_i v_j}{d_i - w_j}$, where u_i and v_j are random numbers, $d_i = i \cdot \frac{b-a}{N}$, $w_j = d_j + \frac{b-a}{2*N}$, $a = 1.0, b = 9.0$, for $i, j = 1, ..., N$. It is known that A is a rank-structured Cauchy-like matrix, see [13]. Let $N = 768$ and assume A is distributed over a 2×3 process grid. Suppose that $N_B = 128$ (the parameter of block size for distribution), we get the BCDD form of matrix A, as shown in Fig. 1. By choosing $N_B = 256$, we get the BDD form of A.

Fig. 3 shows the block partitions of matrix B when choosing different N_B. The numbers in Fig. 3 are the ranks of the corresponding blocks. Fig. 3(a) shows the ranks of blocks of B when A is in the BCDD form. From it we can see the off-diagonal ranks in Fig. 3(a) are larger than those when A is in BDD form, which are shown in Fig. 3(b). Many flops can be saved when the ranks are small. When A is initially in BCDD form with small N_B, we can transform it to BDD form by using ScaLAPACK routine `PDGEMR2D`. We compare these two cases in Example 1 in section 4.



(a) Block cyclic distribution of matrix B when $N_B = 128$.

(b) Block distribution of B when $N_B = 256$.

Figure 3: The ranks of B blocks when A is in block cyclic and block data distribution forms, respectively.

3.1.2 Comparison with other algorithms

Algorithm 2 works on the whole block of local matrices, like block partitioned based algorithms such as Cannon and Fox. The advantages of PSMMA over Cannon and Fox algorithms are that Algorithm 2 also works for block cyclic data distribution and also works for any rectangular process grids.

Comparing with PUMMA algorithm, which works for block cyclic data distribution and rectangular process grids, Algorithm 2 requires less workspace and the size of local matrix multiplications is larger than in PUMMA. Based on the *LCM* concept, PUMMA needs extra workspace to permute block columns of A and block rows of B together, which can be multiplied simultaneously. Therefore, PUMMA requires roughly as much extra workspace to store another copy of the local matrices A and B, which makes it impractical in real applications, see [24, 36]. Furthermore, only partial block columns (rows) can be merged together in PUMMA, while PSMMA always multiplies the whole local matrix A with B and it does not need to permute the block columns of A or the block rows of B.

Comparing with SUMMA, which is based on the outer product form of matrix multiplication, PSMMA can further exploit the low-rank structure of matrix B. It is not easy for SUMMA to exploit this property.

3.2 Parallel Structured DC Algorithm

The excellent performance of the DC algorithm is partially due to *deflation* [37, 1], which happens in two cases. If the entry z_i of z is negligible or zero, the corresponding $(\lambda_i, \hat{q}_i)$ is already an eigenpair of T. Similarly, if two eigenvalues in D are identical then one entry of z can be transformed to zero by applying a sequence of plane rotations. All the deflated eigenvalues are moved to the end of D by a permutation matrix, and so are the corresponding eigenvectors.

Then, after deflation, (3) reduces to

$$T = Q(GP) \begin{pmatrix} \bar{D} + b_k \bar{z}\bar{z}^T & \\ & \bar{D}_d \end{pmatrix} (GP)^T Q^T, \tag{18}$$

where G is the product of all rotations, P is a permutation matrix, and $\bar{D}_d$ are the deflated eigenvalues.

According to (4), the eigenvectors of T are computed as

$$U = \left[\begin{pmatrix} Q_1 & \\ & Q_2 \end{pmatrix} GP \right] \begin{pmatrix} \widehat{Q} & \\ & I_d \end{pmatrix}. \tag{19}$$

To improve efficiency, Gu [38] suggested a permutation strategy for reorganizing the data structure of the orthogonal matrices, which has been used in ScaLAPACK. The matrix in square brackets is permuted as $\begin{pmatrix} Q_{11} & Q_{12} & 0 & Q_{14} \\ 0 & Q_{22} & Q_{23} & Q_{24} \end{pmatrix}$, where the first and third block columns contain the eigenvectors that have not been affected by deflation, the fourth block column contains the deflated eigenvectors, and the second block column contains the remaining columns. Then, the computation of U can be done by two parallel matrix-matrix products (calling `PDGEMM`) involving parts of $\widehat{Q}$ and the matrices $\begin{pmatrix} Q_{11} & Q_{12} \end{pmatrix}$, $\begin{pmatrix} Q_{22} & Q_{23} \end{pmatrix}$. Another factor that contributes to the excellent performance of DC is that most operations can take advantage of highly optimized matrix-matrix products.

When there are few deflations, the size of matrix $\widehat{Q}$ in (19) will be large, and most of the time spent by DC would correspond to the matrix-matrix multiplication in (19). Furthermore, it is well-known that matrix $\widehat{Q}$ defined as in (4) is a Cauchy-like matrix with off-diagonally low rank property, see [2, 12]. Therefore, we simply use the parallel structured matrix-matrix multiplication algorithm to compute the eigenvector matrix U in (19). Since PSMMA requires much fewer floating point operations and communications than the plain matrix-matrix multiplication, `PDGEMM`, this approach makes the DC algorithm in ScaLAPACK much faster.

As mentioned before, the central idea is to replace `PDGEMM` by PSMMA. The eigenvectors are updated in the ScaLAPACK routine `PDLAED1`, and therefore we modify it and call PSMMA in it instead of `PDGEMM`. The whole procedure of PSDC accelerated by PSMMA is summarized in Algorithm 3. Comparing with the classical DC algorithm (Algorithm 1), the only difference is that PSMMA is used when the size of matrix M is large. In Fig. 4 the stages of PSDC are graphically represented.

Note that after applying permutations to Q in (19), matrix $\widehat{Q}$ should also be permuted accordingly. From the results in [2, 12, 13], we know that $\widehat{Q}$ is a Cauchy-like matrix and off-diagonally low-rank, the numerical rank is usually around 50-100. When combining with PSMMA, we would not use Gu's idea [38] since permutation may destroy the off-diagonally low-rank structure of $\widehat{Q}$ in (19). We need to modify the ScaLAPACK routine `PDLAED2`, and only when the size of deflated matrix $\bar{D}$ in (18) is large enough, PSMMA would be used, otherwise use Gu's idea. In section 4, we denote the size of $\bar{D}$ by K, whose value depends on the matrix as well as the architecture of the particular parallel computer used, and may be different for different computers. In Example 2, PSMMA is used when $K \geq 20,000$.

Remark 3. To keep the orthogonality of $\widehat{Q}$ (see equation (8)), $d_i - \lambda_j$ must be computed by

$$d_i - \lambda_j = \begin{cases} (d_i - d_j) - \gamma_j & \text{if } i \leq j \\ (d_i - d_{j+1}) + \mu_j & \text{if } i > j \end{cases}, \tag{20}$$

where $\gamma_i = \lambda_i - d_i$ (the distance between λ_i and d_i), and $\mu_i = d_{i+1} - \lambda_i$ (the distance between λ_i and d_{i+1}), which can be returned by calling the LAPACK routine `DLAED4`. In our implementation, $\widehat{Q}$ is represented by using *five* generators, $\{d_i\}, \{\gamma_i\}, \{\mu_i\}, \{u_i\}$ and $\{v_i\}$.

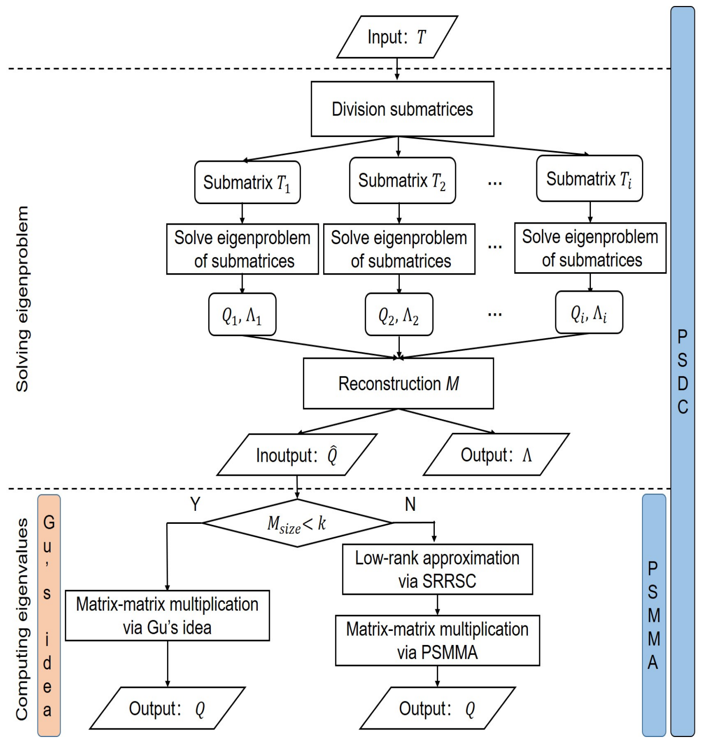


Figure 4: The stages of the PSDC method for solving the symmetric tridiagonal eigenvalue problem

4 Experimental results

All the numerical results are obtained on the Tianhe-2A supercomputer [39, 40], located in Guangzhou, China. Each compute node is equipped with two 12-cores Intel Xeon E5-2692 v2 CPUs and our experiments only use CPU cores. The details of the test platform and environment of compute nodes are shown in Table 1. For all these numerical experiments, we only used plain MPI, run 24 MPI processes per node in principle, and one process per core. For example, we used 171 compute nodes for testing 4096 processes.

Example 1. Assume that a Cauchy-like matrix A is defined as $A_{ij} = \frac{u_i v_j}{d_i - w_j}$, where u_i and v_j are random numbers, $d_i = i \cdot \frac{b-a}{N}$, $w_j = d_j + \frac{b-a}{2*N}$, $a = 1.0, b = 9.0$, for $i, j = 1, ..., N$. It is known that A is a rank-structured matrix, see [13, 12]. We let $N = 8192$, 16384 and 32768, respectively, and choose different number of processes, $N_P = 16$, 64, 256, 1024 and 4096, to compare `PDGEMM` with PSMMA. To avoid performance variance during multiple executions, we evaluated the performance of `PDGEMM` and PSMMA twice in the same program and called that program three times, and chose the best results among these six executions. Our codes will be released on Github (available at https://github.com/shengguolsg/).

It is well-known that the performance of `PDGEMM` depends on the block size N_B. We tested the performances of `PDGEMM` by choosing $N_B = 64$, 128 and 256, and we found that their differences are very small. But they are better than choosing $N_B \leq 32$. Therefore, we chose $N_B = 64$ and $N/\sqrt{N_P}$, which corresponds to the BCDD form and BDD form, respectively. Note that a large N_B is better for PSMMA since the ranks of off-diagonal blocks after permutation are smaller, see Table 2.

As shown in section 3.1.1, we can redistribute matrix A from BCDD to BDD to exploit the off-diagonal low-rank property of matrix B. In this example, we tested four versions of PSMMA, which are

ALGORITHM 3: PSDC(T, Q, Λ) algorithm for computing the eigendecomposition of a symmetric tridiagonal matrix

```
Input: $T \in \mathbb{R}^{N\times N}$
Output: eigenvalues $\Lambda$, eigenvectors $Q$
if the size of T is small enough then
    apply the QR algorithm and compute $T = Q\Lambda Q^T$ ;
    return $Q$ and $\Lambda$;
else
    form $T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + b_k vv^T$;
    call PSDC$(T_1, Q_1, \Lambda_1)$;
    call PSDC$(T_2, Q_2, \Lambda_2)$;
    form $M = \bar{D} + b_k \bar{z}\bar{z}^T$ from $Q_1, Q_2, \Lambda_1, \Lambda_2$, and $v$, $\bar{D} = \mathrm{diag}(\Lambda_1, \Lambda_2)$ after deflations;
    if the size of matrix M is small then
        find eigenvalues $\Lambda$ and eigenvectors $\widehat{Q}$ of $M$;
        use Gu's idea to calculate $Q = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \widehat{Q}$;
    else
        find eigenvalues $\Lambda$ and eigenvectors $\widehat{Q}$ of $M$;
        use PSMMA (via SRRSC) to calculate $Q = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \widehat{Q}$;
    end
    return $Q$ and $\Lambda$;
end
```

- PSMMA_BCDD: $N_B = 64$ and with low-rank approximation;
- PSMMA_BDD: $N_B = N/\sqrt{N_P}$ and with low-rank approximation;
- PSMMA_WRedist: $N_B = 64$ and with data redistribution and low-rank approximation (matrix A is transformed from BCDD to BDD with $N_B = N/\sqrt{N_P}$ and then back);
- PSMMA_NLowrank: $N_B = 64$ and without low-rank approximation;

The speedups of PSMMA over `PDGEMM` are shown in Fig. 5(a), 5(b) and 5(c) with dimensions $N = 8192, 16384, 32768$, respectively. From the results, we can see that PSMMA_BCDD is always faster than `PDGEMM` except for $N_P = 16$. It is because the ranks of B blocks are very large when using the BCDD form, and most of the time is spent in computing the low-rank

Table 1: The test platform and environment of one node

Items	Values
2*CPU	Intel Xeon CPU E5-2692 v2@2.2GHz
Memory size	64GB (DDR3)
Operating System	Linux 3.10.0
Complier	Intel ifort 2013_sp1.2.144
Optimization	-O3 -mavx

approximations. Table 2 shows the ranks of the off-diagonal blocks in the first block column, see Fig. 3 for the structure of matrix B. Without using low-rank approximation, PSMMA can be faster than `PDGEMM`. It is interesting to see that PSMMA_NLowrank is always faster than `PDGEMM` for these three matrices. Note that PSMMA_NLowrank requires more floating point operations than `PDGEMM` since it needs to construct the local B submatrices. However, PSMMA_NLowrank requires fewer communications than `PDGEMM`.

PSMMA_WRedist is only slower than `PDGEMM` when the size of the matrix is small ($N = 8192$) and the number of processes is large. From the last row of Table 2, we can see that the ranks of off-diagonal submatrices without permutation are much smaller than the size (4096) of submatrices. PSMMA_WRedist is generally faster than both PSMMA_BCDD and PSMMA_NLowrank. The disadvantage of PSMMA_WRedist is that it requires data redistribution (communication). When the number of processes is large, the data redistribution represents a large portion of the overall time. Fig. 6 shows the percentages of transforming matrix A from BCDD to BDD and transforming it back. It takes over 70% of total time when N is small and N_P is large.

PSMMA_BDD is always the best and can be more than ten times faster than `PDGEMM`. This is because it assumes that matrix A is initially stored in BDD form and the low-rank property of matrix B is not destroyed either.

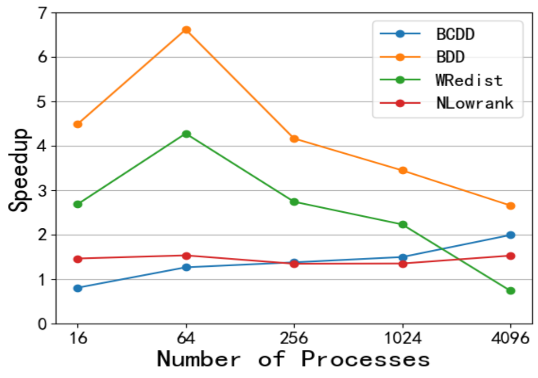
It is difficult to know exactly why the speedups of PSMMA_BDD and PSMMA_WRedist over `PDGEMM` firstly increase and then decrease. It depends on the properties of these two algorithms. Comparing with `PDGEMM`, PSMMA_BDD and PSMMA_WRedist save both computations and communications. When the number of processes increases, these two contributions make the speedups first increase. As the number of processes increases, the size of the local submatrix located on each process decreases, and therefore the percentage of floating point operations saved from using low-rank approximations decreases. Since PSMMA_BDD does not save many floating point operations compared to `PDGEMM`, the speedups decrease when using more processes. For PSMMA_WRedist, the cost of data redistribution also increases as the number of processes increases. Meanwhile, as the number of processes grows, the percentage of saved communication by PSMMA_BDD and PSMMA_WRedist increases. Therefore, PSMMA_BDD can be always faster than `PDGEMM` since it not only reduces computations but also communications. It is better to use PSMMA_NLowrank instead of PSMMA_WRedist when the number of processes is large.

Remark 4. We adaptively choose a particular PSMMA algorithm based on the size of the matrix and the number of processes. It is better to use PSMMA_WRedist when the number of processes is small, and use PSMMA_BCDD or PSMMA_NLowrank when the number of processes is large and/or the size of matrix is large. It is best to use PSMMA_BDD when it is available.

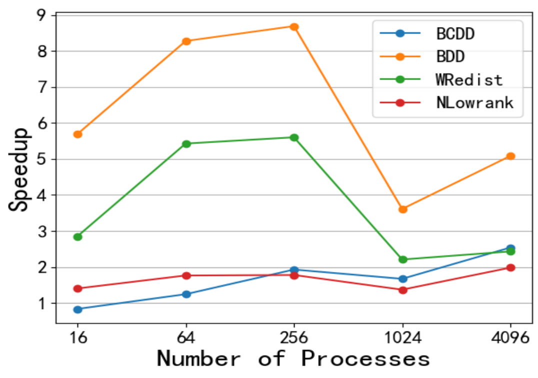
Table 2: The ranks of off-diagonal blocks of B in the first block column when stored on 4×4 processes in BCDD form. Each block is a 4096×4096 submatrix.

N_B	$B(2,1)$	$B(3,1)$	$B(4,1)$
64	1260	892	1252
128	711	445	699
256	401	221	390
4096	34	11	9

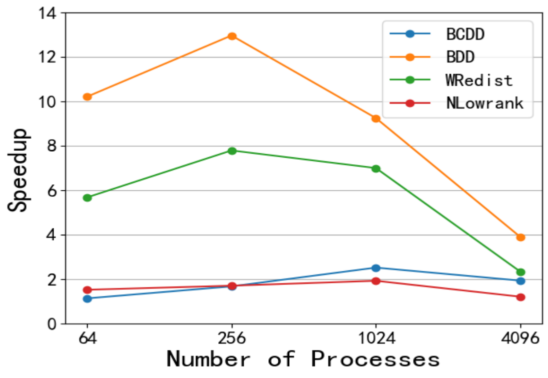
Example 2. We use some 'difficult' matrices [41] for the DC algorithm, for which few or no eigenvalues are deflated. Examples include the Clement-type, Hermite-type and Toeplitz-type matrices, which are defined as follows.



(a) Dimension $N = 8192$



(b) Dimension $N = 16384$



(c) Dimension $N = 32768$

Figure 5: The speedup of PSMMA over `PDGEMM`

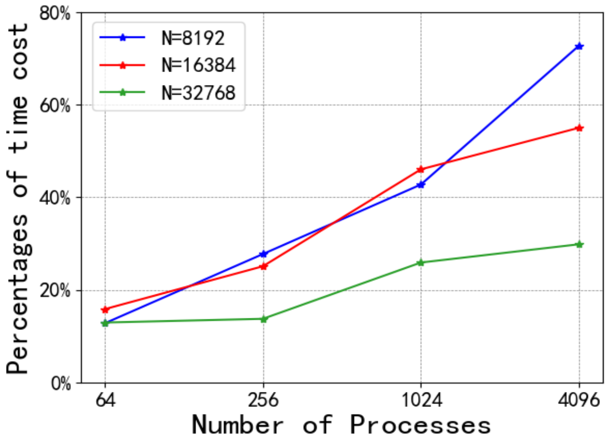


Figure 6: The percentages of time required by data redistribution

The Clement-type matrix [41] is given by

$$T = \text{tridiag}\begin{pmatrix} & \sqrt{n} & & \sqrt{2(n-1)} & & & \sqrt{n\cdot 1} & \\ 0 & & 0 & & \dots & 0 & & 0 \\ & \sqrt{n} & & \sqrt{2(n-1)} & & & \sqrt{n\cdot 1} & \end{pmatrix},$$

where the off-diagonal entries are $\sqrt{i(n+1-i)}, i = 1, \dots, n$.

The Hermite-type matrix is given as [41],

$$T = \text{tridiag}\begin{pmatrix} & \sqrt{1} & & \sqrt{2} & & & \sqrt{n-1} & \\ 0 & & 0 & & \dots & 0 & & 0 \\ & \sqrt{1} & & \sqrt{2} & & & \sqrt{n-1} & \end{pmatrix}.$$

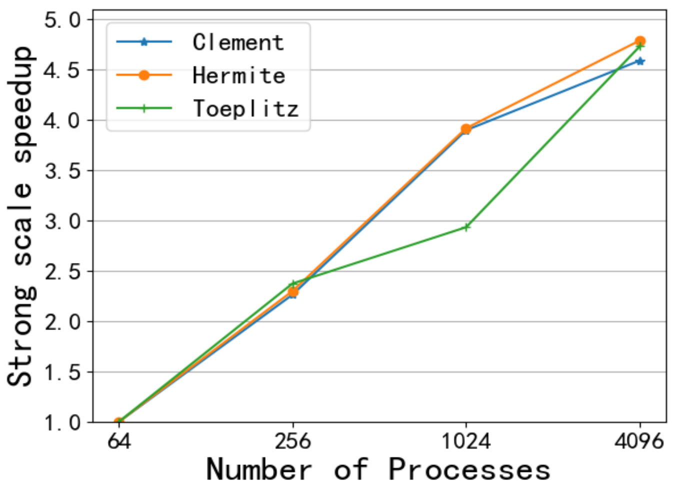
The Toeplitz-type matrix is defined as [41],

$$T = \text{tridiag}\begin{pmatrix} & 1 & & 1 & & 1 & & 1 & \\ 2 & & 2 & & \dots & & 2 & & 2 \\ & 1 & & 1 & & 1 & & 1 & \end{pmatrix}.$$

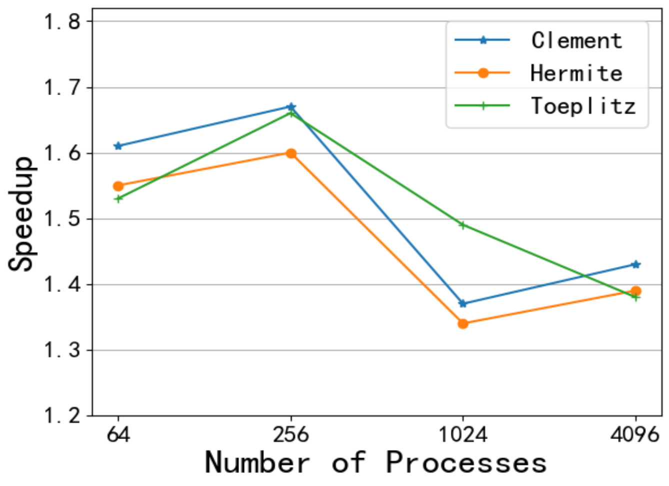
For the results of strong scaling, we let the dimension n be $30,000$, and use rank-structured techniques only when the size of the secular equation is larger than $K = 20,000$. We used PSMMA_WRedist and chose $N_B = 64$. The results for strong scaling of PSDC are shown in Fig. 7(a). The speedups of PSDC over ScaLAPACK are reported in Fig. 7(b). We can see that PSDC is about 1.4x–1.6x times faster than `PDSTEDC` in ScaLAPACK for all cases. Because of deflations, the performances of these three matrices can be different even though they have the same dimensions.

The orthogonality of the eigenvectors computed by PSDC is in the same order as those by ScaLAPACK, as shown in Table 3. The orthogonality of matrix Q is defined as $\|I - QQ^T\|_{\max}$, where $\|\cdot\|_{\max}$ is the maximum absolute value of entries of $(\cdot)$. We confirm that the residuals of eigenpairs computed by PSDC are in the same order as those computed by ScaALAPACK though the results are not included here.

Furthermore, we compare PSDC with PHDC which was introduced in [15] and used STRUMPACK to accelerate the matrix-matrix multiplications. The results are shown in Fig. 8. For these three matrices, PSDC is faster than PHDC when using many processes. It is better to use STRUMPACK when using few processes since HSS-based multiplications can save more floating point operations than BLR-based multiplications.



(a) The strong scaling of PSDC



(b) The speedup of PSDC over ScaLAPACK

Figure 7: The results for the matrices of Example 2

Table 3: The orthogonality of the computed eigenvectors by PSDC

Matrix	Number of Processes			
	64	256	1024	4096
Clement	3.02e-14	3.73e-14	3.80e-14	3.65e-14
Hermite	2.49e-14	2.75e-14	2.96e-14	3.01e-14
Toeplitz	2.88e-14	3.01e-14	3.03e-14	2.97e-14

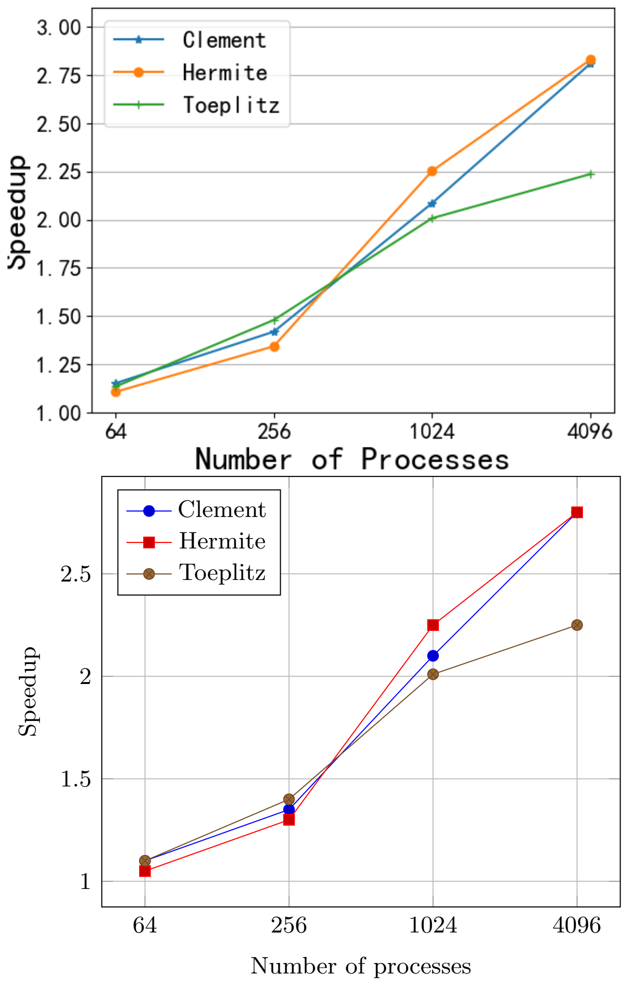


Figure 8: The speedup of PSDC over PHDC

4.1 Results from real applications

In this subsection, we use three matrices that come from real applications to test PSDC. One comes from the spherical harmonic transform (SHT) [42], which has been used in [15]. One symmetric tridiagonal matrix is defined as follows, which will be denoted by SHT,

$$A_{jk} = \begin{cases} c_{m+2j-2}, & k = j-1 \\ d_{m+2j}, & k = j \\ c_{m+2j}, & k = j+1 \\ 0, & \text{otherwise}, \end{cases} \tag{21}$$

for $j, k = 0, 1, \ldots, n-1$, where $\xi_l = l - m$,

$$c_l = \sqrt{\frac{(\xi_l+1)(\xi_l+2)(l+m+1)(l+m+2)}{(2l+1)(2l+3)^2(2l+5)}},$$

$$d_l = \frac{2l(l+1) - 2m^2 - 1}{(2l-1)(2l+3)},$$

for $l = m, m+1, m+2, \ldots$. We assume the dimension of this matrix is $n = 30,000$ and $m = n$.

The other two are sparse matrices obtained from the SuiteSparse matrix collection [43], called SiO and Si5H12. We first reduce each matrix into its tridiagonal form by calling ScaLAPACK routines and then call PSDC to compute its eigendecomposition. It is also the general process for computing the eigenvalue decomposition of any symmetric (sparse) matrices. These matrices are real and symmetric and their dimensions are $n = 33,401$ and $19,896$, respectively.

Example 3. We use matrices SHT, SiO and Si5H12 to compare PSDC with `PDSTEDC`. In this example, we use the rank-structured techniques, i.e. calling PSMMA, whenever the size of the secular equation is larger than $K = 15,000$, since the largest K for matrix Si5H12 is $15,489$ when $N_B = 64$. The speedups of PSDC over `PDSTEDC` are reported in Table 4. The backward errors of the computed eigenpairs are also included in the third column, which are computed as

$$\mathit{Residual} = \frac{\|A - Q\Lambda Q^*\|_c}{\|A\|_2}, \tag{22}$$

where $Q \in \mathbb{R}^{n\times n}$ is orthogonal, $\Lambda \in \mathbb{R}^{n\times n}$ is diagonal with the eigenvalues as its diagonal elements, $\|X\|_c$ denotes the maximum Frobenius norm of each column of X and $\|X\|_2$ denotes the 2-norm of X, its largest singular value.

Table 4: The speedups of PSDC over `PDSTEDC` for matrices from real applications

Matrix	K	Residual	Number of Processes			
			64	256	1024	4096
SHT	$27,136$	1.10e-14	1.20	1.48	1.42	1.41
Si5H12	$15,489$	3.54e-15	1.27	1.16	1.15	1.05
SiO	$24,981$	1.55e-14	1.46	1.31	1.10	1.15

Example 4. It is known that ELPA (Eigenvalue soLver for Petascale Applications [21, 44]) has better scalability and is faster than the MKL version of ScaLAPACK. As opposed to ScaLAPACK, ELPA routines do not rely on BLACS and PBLAS, and it can overlap the computations with communications, and the computation is also optimized by using OpenMP

and even GPU. For the tridiagonal eigensolver, ELPA rewrites the DC algorithm and implements its own matrix-matrix multiplications and does not use the PBLAS routine `PDGEMM`. In contrast, PSDC follows the main procedure of `PDSTEDC`, and only uses PSMMA to accelerate the expensive matrix-matrix multiplication part.

We compare PSDC with ELPA and find that it competes with ELPA (with version 2018.11.001). We use the Clement matrix to do experiments. Fig. 9 shows the execution times of PSDC and ELPA when using different processes. It shows that PSDC is faster than ELPA when using few processes, but PSDC becomes slower when using more than 1024 processes. It is because matrix multiplication is no longer the dominant factor when using many processes. It is shown in [15] that the percentage of time dedicated to the matrix multiplications can be less than 10% of the total time. The gains of ELPA are obtained from other optimization techniques.

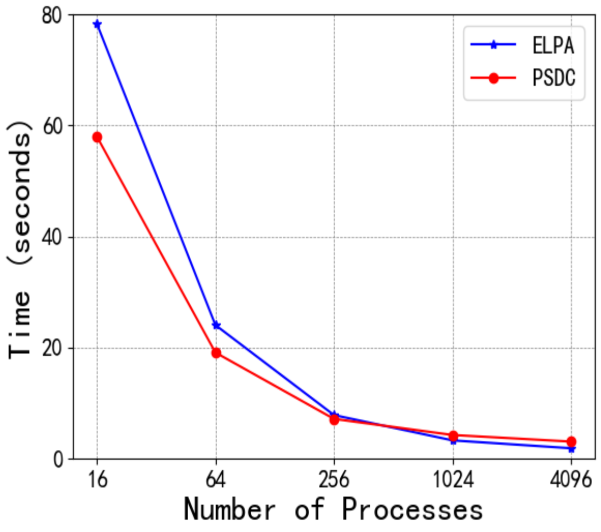


Figure 9: The comparison of PSDC with ELPA

4.2 Future works

In this subsection we discuss the current implementation bottlenecks and the future works. A restriction of our current code is that PSMMA was only used at the top level of the DC tree. It should be used at any level as long as the size of the matrix is large. This will be modified in the near future. We did not use OpenMP or vectorization to optimize our routines. The routine for constructing the local submatrices from generators can be optimized by using vectorization techniques and OpenMP.

Following our current work, there are some interesting research projects to do in the future. First, PSMMA can be used to extend the banded DC algorithms proposed in [14, 45] to distributed memory platforms in a similar manner. Secondly, the structured matrix-matrix multiplication techniques can be used in heterogeneous architectures, which can reduce the data movements from CPU to the accelerators such as GPU[1]. We only transform the generators to GPU once instead of many submatrices. Last but not least, PSMMA can be adapted for Toeplitz, Hankel, DFT (Discrete Fourier Transform) and other structured matrices [46]. Some results will be included in our future works.

[1]The authors would like to thank the referee for pointing out this research direction.

5 Conclusions

The starting point of this paper is trying to accelerate the tridiagonal DC algorithm implemented in ScaLAPACK [7] for some difficult matrices. It is known that the main task lies in multiplying a general matrix with a rank-structured Cauchy-like matrix, see [1, 2, 12]. The main contribution of this paper is that a highly scalable parallel matrix multiplication algorithm is proposed for rank-structured Cauchy-like matrices, which fits well for the parallel tridiagonal DC algorithm.

The matrix multiplication problem is known to be very compute intensive. However, as HPC moves towards exascale computing, the development of communication-avoiding or communication-decreasing algorithms becomes more and more important. By taking advantage of the particular structures of Cauchy-like matrices, we proposed a parallel structured matrix multiplication algorithm PSMMA, which can reduce both computation and communication costs. The workflow of PSMMA is similar to PUMMA [26] and further exploits the rank-structured property of input matrices. Numerical results show that PSMMA can be much faster than `PDGEMM` for rank-structured Cauchy-like matrices, and the speedups over `PDGEMM` can be up to 12.96.

By combing PSMMA with the parallel tridiagonal DC algorithm, we propose a parallel structured DC algorithm (PSDC). For these difficult matrices for which DC deflates very few eigenvalues, PSDC is always much faster than the classical DC algorithm implemented in ScaLAPACK. Unlike PHDC which is proposed in [15], PSDC does not have scalability problem and it can scale to 4096 processes at least.

Acknowledgment

The authors would like to thank the referees for their valuable comments which greatly improve the presentation of this paper. This work is supported by National Key RD Program of China (2018YFB0204303), National Natural Science Foundation of China (No. 11401580, 91530324, U1611261, 61872392 and U1811461), NSF of Hunan (No. 806175290082), NSF of NUDT (No. ZK18-03-01), Guangdong Natural Science Foundation (2018B030312002), and the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant (No. 2016ZT06D211). Jose E. Roman was supported by the Spanish Agencia Estatal de Investigación (AEI) under project SLEPc-DA (PID2019-107379RB-I00).

References

[1] J. J. M. Cuppen, "A divide and conquer method for the symmetric tridiagonal eigenproblem," *Numer. Math.*, vol. 36, pp. 177–195, 1981.

[2] M. Gu and S. C. Eisenstat, "A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem," *SIAM J. Matrix Anal. Appl.*, vol. 16, pp. 172–191, 1995.

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

[4] I. S. Dhillon and B. N. Parlett, "Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices," *Linear Algebra Appl.*, vol. 387, pp. 1–28, 2004.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[6] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. Whaley, "Scalapack: A portable linear algebra library for distributed memory computers-design issues and performance," *Computer Physics Communications*, vol. 97, pp. 1–15, 1996.

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

[8] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM J. Matrix Anal. Appl.*, vol. 21, no. 2, pp. 562–580, 2011.

[9] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM J. Sci. Comput.*, vol. 34, pp. A206–A239, 2012.

[10] S. Chandrasekaran, M. Gu, and T. Pals, "Fast and stable algorithms for hierarchically semi-separable representations," University of California, Berkeley, CA, Tech. Rep., 2004.

[11] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals, "A fast solver for HSS representations via sparse matrices," *SIAM J. Matrix Anal. Appl.*, vol. 29, pp. 67–81, 2006.

[12] S. Li, X. Liao, J. Liu, and H. Jiang, "New fast divide-and-conquer algorithm for the symmetric tridiagonal eigenvalue problem," *Numer. Linear Algebra Appl.*, vol. 23, pp. 656–673, 2016.

[13] S. Li, M. Gu, L. Cheng, X. Chi, and M. Sun, "An accelerated divide-and-conquer algorithm for the bidiagonal SVD problem," *SIAM J. Matrix Anal. Appl.*, vol. 35, no. 3, pp. 1038–1057, 2014.

[14] X. Liao, S. Li, L. Cheng, and M. Gu, "An improved divide-and-conquer algorithm for the banded matrices with narrow bandwidths," *Comput. Math. Appl.*, vol. 71, pp. 1933–1943, 2016.

[15] S. Li, F.-H. Rouet, J. Liu, C. Huang, X. Gao, and X. Chi, "An efficient hybrid tridiagonal divide-and-conquer algorithm on distributed memory architectures," *Journal of Computational and Applied Mathematics*, vol. 344, pp. 512–520, 2018.

[16] F. Rouet, X. Li, P. Ghysels, and A. Napov, "A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization," *ACM Transactions on Mathematical Software*, vol. 42, no. 4, pp. 27:1–35, 2016.

[17] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker, "Improving multifrontal methods by means of block low-rank representations," *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. A1451–A1474, 2015.

[18] W. Hackbusch, "A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices," *Computing*, vol. 62, pp. 89–108, 1999.

[19] W. Hackbusch, B. Khoromskij, and S. Sauter, "On $\mathcal{H}^2$-matrices," in *Lecture on Applied Mathematics*, Z. C. Bungartz H, Hoppe RHW, Ed. Berlin: Springer, 2000, pp. 9–29.

[20] I. Yamazaki, A. Ida, R. Yokota, and J. Dongarra, "Distributed-memory lattice $\mathcal{H}$-matrix factorization," *International Journal of High Performance Computing Applications*, vol. 33, no. 5, pp. 1046–1063, 2019.

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

[22] L. E. Cannon, “A cellular computer to implement the kalman filter algorithm,” Ph.D. dissertation, Montana State Univesity, 1969.

[23] G. C. Fox, S. W. Otto, and A. J. G. Hey, “Matrix algorithms on a hypercube I: matrix multiplication,” *parallel computing*, vol. 4, no. 1, pp. 17–31, 1987.

[24] J. Choi, D. W. Walker, and J. J. Dongarra, “Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers,” *Concurrency and Computation: Practice and Experience*, vol. 6, no. 7, pp. 543–570, 1994.

[25] M. Gu and J. Xia, “A multi-structured superfast Toeplitz solver,” preprint, 2009.

[26] R. A. V. De Geijn and J. Watts, “SUMMA: Scalable universal matrix multiplication algorithm,” *Concurrency and Computation: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[27] M. Gu and S. C. Eisenstat, “A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem,” *SIAM J. Matrix Anal. Appl.*, vol. 15, pp. 1266–1276, 1994.

[28] I. Gohberg, T. Kailath, and V. Olshevsky, “Fast Gaussian elimination with partial pivoting for matrices with displacement structure,” *Mathematics of Computation*, vol. 64, no. 212, pp. 1557–1576, 1995.

[29] M. Gu, “Stable and efficient algorithms for structured systems of linear equations,” *SIAM J. Matrix Anal. Appl.*, vol. 19, no. 2, pp. 279–306, 1998.

[30] C.-T. Pan, “On the existence and computation of rank revealing LU factorizations,” *Linear Algebra Appl.*, vol. 316, pp. 199–222, 2000.

[31] E. Solomonik and J. Demmel, “Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms,” in *Proc. 17th international conference on Parallel processing (Euro-Par 2011), Part II*, 2011, pp. 90–109.

[32] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, “Communication-optimal parallel recursive rectangular matrix multiplication,” in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing.* IEEE, 2013, pp. 261–272.

[33] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, “Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions,” in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing.* IEEE, 2013, pp. 813–824.

[34] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, “Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–22.

[35] V. Y. Pan, *Structured Matrices and polynomials, unified superfast algorithms.* New York: Birkhäuser, Springer, 2001.

[36] J. Choi, “A new parallel matrix multiplication algorithm on distributed-memory concurrent computers,” *Concurrency: practice and experience*, vol. 10, no. 8, pp. 655–670, 1998.

[37] J. R. Bunch, C. P. Nielsen, and D. C. Sorensen, “Rank one modification of the symmetric eigenproblem,” *Numer. Math.*, vol. 31, pp. 31–48, 1978.

[38] M. Gu, “Studies in numerical linear algebra,” Ph.D. dissertation, Yale University, New Haven, CT, 1993.

[39] X. Liao, L. Xiao, C. Yang, and Y. Lu, “Milkyway-2 supercomputer: System and application,” *Frontiers of Computer Science*, vol. 8, no. 3, pp. 345–356, 2014.

[40] Y. Liu, C. Yang, F. Liu, X. Zhang, Y. Lu, Y. Du, C. Yang, M. Xie, and X. Liao, “623 Tflop/s HPCG run on Tianhe-2: Leveraging millions of hybrid cores,” *International Journal of High Performace Computing Applications*, vol. 30, no. 1, pp. 39–54, 2016.

[41] O. A. Marques, C. Voemel, J. W. Demmel, and B. N. Parlett, “Algorithm 880: A testing infrastructure for symmetric tridiagonal eigensolvers,” *ACM Trans. Math. Softw.*, vol. 35, pp. 8:1–13, 2008.

[42] M. Tygert, “Fast algorithms for spherical harmonic expansions, II,” *Journal of Computational Physics*, vol. 227, pp. 4260–4279, 2008.

[43] T. Davis and Y. Hu, “The Univeristy of Florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[44] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H. Bungartz, and H. Lederer, “The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science,” *J. Phys.: Condens. Matter*, vol. 26, pp. 1–15, 2014.

[45] P. Arbenz, “Divide-and-conquer algorithms for the bandsymmetric eigenvalue problem,” *Parallel Comput.*, vol. 18, pp. 1105–1128, 1992.

[46] D. Bini and V. Pan, *Polynomial and Matrix Computations, Volume I Fundamental Agorithms*, ser. Process in Theoretical Computer Science. Birkhäuser, 1994.