Parallel Computing 86 (2019) 57–65

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

# A high performance implementation of Zolo-SVD algorithm on distributed memory systems☆

Shengguo Li[a,*], Jie Liu[a], Yunfei Du[b,*]

[a] College of Computer Science, National University of Defense Technology, Changsha 410073, China
[b] Department of Computer Science, Sun Yat-Sen University, Guangzhou 510006, China

ARTICLE INFO

*Article history:*
Received 15 June 2018
Revised 22 February 2019
Accepted 17 April 2019
Available online 22 April 2019

*MSC:*
65F15
68W10

*Keywords:*
ScaLAPACK
Polar decomposition
Zolotarev
QDWH
Distributed parallel algorithm

ABSTRACT

This paper introduces a high performance implementation of the `Zolo-SVD` algorithm on distributed memory systems, which is based on the polar decomposition (PD) algorithm via the Zolotarev's function (`Zolo-PD`), originally proposed by Nakatsukasa and Freund [SIAM Review, 2016]. Our implementation highly relies on the routines of ScaLAPACK and therefore it is portable. Compared with the other PD algorithms such as the QR-based dynamically weighted Halley method (`QDWH-PD`), `Zolo-PD` is naturally parallelizable and has better scalability though performs more floating-point operations. When using many processors, `Zolo-PD` is usually 1.20 times faster than the `QDWH-PD` algorithm, and `Zolo-SVD` can be about two times faster than the ScaLAPACK routine `PDGESVD`. These numerical experiments are performed on Tianhe-2A supercomputer, one of the fastest supercomputers in the world, and the tested matrices include some sparse matrices from particular applications and some randomly generated dense matrices with different dimensions. Our `QDWH-SVD` and `Zolo-SVD` implementations are freely available at https://github.com/shengguolsg/Zolo-SVD.

© 2019 Elsevier B.V. All rights reserved.

## 1. Introduction

Computing the SVD of a matrix is an important problem in scientific computing and many applications. For example, SVD has been used for information retrieval [10], principal component analysis (PCA) in Statistics [24], and signal processing [29]. How to compute it in an efficient and scalable way has gathered much attention.

State-of-the-art SVD solvers are based on the bidiagonal reduction (BRD) strategy, consisting of the following three stages. First, a general matrix is reduced to an upper bidiagonal form by a sequence of (two-sided) orthogonal transformations [11], and this step is called *bidiagonal reduction*. Second, the bidiagonal SVD problem is solved by any standard method such as DC [17], QR [13] or MRRR [39]. Finally, the singular vectors are computed by accumulating the orthogonal transformations from the bidiagonal reduction, which is called *back transformation*. The reduction step is the most time-consuming phase, and takes 75%–99% of the total time on homogeneous multicore architecture [27]. Some recent works have been focused on accelerating the bidiagonal reduction phase, see references [15,19,27]. To be efficient, much effort is required to develop an efficient and scalable algorithm. Every detail has to be considered including synchronization of multi-threads, vectorization, cache issues, and so on.

There are four main types of bidiagonalization methods. One is the classical one-stage approach, used in LAPACK and ScaLAPACK, which directly reduces a matrix to its bidiagonal form through sequences of orthogonal transformations (Householder or Givens) from two sides. Another is the two-stage approach, proposed in [14], which first reduces a matrix to its banded form and then a banded matrix is bidiagonalized. Another approach is obtained by means of the Lanczos algorithm, see [16] and [35]. The fourth approach is the so called *one-sided bidiagonalization*, first proposed by Ralha in [34], and stabilized by Barlow, Bosner, and Drmač [4] and a block parallel version is proposed in [6].

In this work we exploit a different approach. We try out some new algorithms firstly proposed in the numerical linear algebra community. Higham and Papadimitriou [23] introduce a new framework for computing the SVD which is based on the polar decomposition and combines with the eigendecomposition algorithms. Note that any rectangular matrix $A \in \mathbb{C}^{m\times n} (m \geq n)$ has a polar decomposition (PD)

$$A = Q_p H, \tag{1}$$

where $Q_p \in \mathbb{C}^{m\times n}$ is a (tall) matrix with orthogonal columns and $H \in \mathbb{C}^{n\times n}$ is Hermitian positive semidefinite [21]. The SVD of $A$ can

☆ **Conflict of interest.** None.

* Corresponding authors.
*E-mail addresses:* nudtlsg@gmail.com (S. Li), duyunfei@mail.sysu.edu.cn (Y. Du).

https://doi.org/10.1016/j.parco.2019.04.004
0167-8191/© 2019 Elsevier B.V. All rights reserved.

be obtained by further computing the eigendecomposition of $H$, .i.e, $A = Q_p(V\Lambda V^*) = (Q_pV)\Lambda V^* := U\Lambda V^*$.

One advantage of this framework is that it can be accelerated by many efficient PD algorithms and some well-developed scalable eigensolvers (such as ELPA [28]) without implementing the complicated bidiagonalization codes. Therefore, its implementation is relatively simple. While, its main drawback is that it requires much more floating-point operations than the bidiagonal reduction approach, see [31,32] for details. This framework is well-known in the numerical linear algebra community, but there are few or no results on its performance on supercomputers compared with existing parallel SVD algorithms, for example, the algorithms in ScaLAPACK [5,7], the most famous parallel numerical linear package.

The SVD problem is reduced to an eigenvalue problem via the polar decomposition, and therefore the recent well-developed scalable eigenvalue packages are usable. The remaining problem is how to compute the polar decomposition in a scalable and efficient way. There exist many algorithms such as the scaled Newton (SN) method [21], and the QR-based dynamically weighted Halley (QDWH) method [30]. In [31], Nakatsukasa and Freund proposed a variant of the QDWH-PD algorithm with higher order of convergence for the polar decomposition by using Zolotarev's function, and call it Zolo-PD. It is shown that the convergence order of Zolo-PD can be 17, and it usually requires one or two iterations (while QDWH requires less than 6 iterations). As in [31], we still name this Zolo-PD based SVD algorithm as Zolo-SVD.

The QDWH-SVD algorithm which is used to compute the SVD in [32], has been implemented on multicore architecture enhanced with multiple GPUs [36,37]. The results there show that QDWH-SVD can outperform the standard methods. Sukkari et al. [38] further present a comprehensive performance of QDWH-SVD on a large-scale distributed-memory platform, based on ScaLAPACK [7]. With QDWH-PD as a preprocessing step and using the eigensolver ELPA [28] for computing the eigendecomposition, the distributed parallel QDWH-SVD algorithm [38] achieves up to five-fold and two-fold than the ScaLAPACK routine PDGESVD on ill and well-conditioned matrices, respectively. A more complete performance study is included their more recent paper [8].

In this paper we mainly exploit the Zolo-SVD algorithm, and introduce a high performance Zolo-PD implementation on distributed-memory platforms based on ScaLAPACK. We discuss a way to take advantage of the sparsity structure in the QR factorization, which can be done by modifying ScaLAPACK routines, see Section 3.1. We further compare the performance of QDWH-PD and Zolo-PD. It turns out that Zolo-PD has better scalability than QDWH, though it requires more floating-point operations. When using many MPI processes, Zolo-PD can be much faster than QDWH-PD. Combining with ELPA, we show that Zolo-SVD can be much faster than the ScaLAPACK routine PDGESVD and QDWH-SVD. We use some sparse matrices from University of Florida sparse matrix collection [9], and some randomly constructed matrices to test the Zolo-SVD algorithm. Our QDWH-SVD and Zolo-SVD software library are freely available at https://github.com/shengguolsg/Zolo-SVD.

Note that a similar implementation of Zolo-SVD is discussed in [18], which is also based on ScaLAPACK.[1] Their experiments are conducted on a Cray XC40 system by using randomly generated ill-conditioned dense matrices, and draw a similar conclusion that Zolo-PD outperforms QDWH when using many processes. Comparing with it, we further exploit the sparsity structure and use some sparse matrices from particular applications to do the experiments.

[1] We know this later after the first review round of this paper. Our manuscript was first motivated by Sukkari et al. [37] and mostly finished before the end of July 2017.

## 2. QDWH-PD and Zolo-PD

The polar decomposition is an important problem in numerical linear algebra community, and the well-known PD algorithms include the scaled Newton (SN) method [21], QDWH-PD [30], and Zolo-PD [31], etc.

### 2.1. QDWH-PD

QDWH is a QR-based dynamically weighted Halley iterative method for computing the polar decomposition [30]. QDWH computes the polar factor $Q$ as the limit of the sequence $X_k$ defined by

$$X_{k+1} = X_k(a_kI + b_kX_k^*X_k)(I + c_kX_k^*X_k)^{-1}, \quad X_0 = A/\alpha, \tag{2}$$

where $\alpha$ is an upper bound of the maximum singular value of $A$, $\alpha \geq \|A\|_2$. In QDWH the parameters $a_k$, $b_k$ and $c_k$ are chosen dynamically to speed up the convergence. If $a_k = 3, b_k = 1, c_k = 3$ are fixed, it gives the Halley iteration, which is cubically convergent [21].

The iteration (2) requires explicit matrix inversion and thus it may have potential numerical stability issue. It is shown in [30] that (2) is mathematically equivalent to a QR-based implementation, which is inverse-free. The practical QDWH iteration is : $X_0 = A/\alpha$,

$$X_{k+1} = \frac{b_k}{c_k}X_k + \frac{1}{\sqrt{c_k}}\left(a_k - \frac{b_k}{c_k}\right)Q_1Q_2^*, \tag{3}$$

where $\begin{bmatrix}\sqrt{c_k}X_k \\ I\end{bmatrix} = \begin{bmatrix}Q_1 \\ Q_2\end{bmatrix}R$, $k \geq 0$. The main cost lies in computing the QR factorization of an $(m+n)\times n$ matrix and a matrix multiplication, both of which can be done in a communication-optimal manner [3,12].

Iteration (2) is also mathematically equivalent to the following form,

$$\begin{aligned} Z_k &= I + c_kX_k^*X_k, \quad L_k = \mathrm{chol}(Z_k), \\ X_{k+1} &= \tfrac{b_k}{c_k}X_k + \left(a_k - \tfrac{b_k}{c_k}\right)(X_kL_k^{-1})L_k^{-*}, \end{aligned} \tag{4}$$

where $\mathrm{chol}(Z_k)$ denotes the Cholesky factorization of $Z_k$. The starting point is that when the absolute value of $c_k$ is small, matrix $Z_k$ is probably well-conditioned and computing the Cholesky factorization is cheaper than QR factorization. As suggested in [32], we switch from (3) to (4) as long as $c_k \leq 100$. In general, the QR iteration (3) is usually required only once or twice. Our implementation of the QDWH algorithm is based on ScaLAPACK, and the main procedure is quite similar to that one in [38].

### 2.2. Zolo-PD

The QDWH parameters $a$, $b$, and $c$ in (2) are computed as the solution of the rational max-min optimization problem in [30],

$$\max_{a,b,c}\min_{\ell\leq x\leq 1} x\frac{a+bx^2}{1+cx^2}, \tag{5}$$

subject to the constraint $f(x) = x\frac{a+bx^2}{1+cx^2} \leq 1$ on [0, 1]. The parameters in the QDWH iteration can also be obtained by finding the best type-(3, 2) rational approximation to the sign function in the infinity norm, see [31].

Nakatsukasa and Freund [31] extend $f(x) \in \mathcal{R}_{3,2}$ to higher order rational polynomials $f(x) \in \mathcal{R}_{2r+1,2r}$ for general $r \geq 1$. The obtained optimal rational function is called $Z_{2r+1}(x;\ell)$ *the type*$(2r+1, 2r)$ *Zolotarev function* corresponding to $\ell$, and the solution is given by

$$Z_{2r+1}(x;\ell) = Mx\prod_{j=1}^{r}\frac{x^2+c_{2j}}{x^2+c_{2j-1}}. \tag{6}$$

Here, the constant $M > 0$ is uniquely determined by the condition

$$1 - Z_{2r+1}(1;\ell) = -(1 - Z_{2r+1}(\ell;\ell)),$$

and the coefficients $c_1, c_2, \ldots, c_{2r}$ are given by

$$c_i = \ell^2 \frac{sn^2\left(\frac{iK'}{2r+1};\ell'\right)}{cn^2\left(\frac{iK'}{2r+1};\ell'\right)}, \quad i = 1, 2, \ldots, 2r, \tag{7}$$

where $sn(u;\ell')$ and $cn(u;\ell')$ are the Jacobi elliptic functions (see, e.g., [1, Ch. 5]), $\ell' = \sqrt{1-\ell^2}$ and $K' = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1-(\ell')^2\sin^2\theta}}$. It is more convenient to use the scaled Zolotarev function

$$\hat{Z}_{2r+1}(x;\ell) := \frac{Z_{2r+1}(x;\ell)}{Z_{2r+1}(1;\ell)} = \hat{M}x\prod_{j=1}^{r}\frac{x^2+c_{2j}}{x^2+c_{2j-1}}, \tag{8}$$

where $\hat{M} = \prod_{j=1}^{r}\frac{1+c_{2j-1}}{1+c_{2j}}$.

Combining the iterative process of QDWH with the scaled Zolotarev functions, we arrive at an algorithm of computing the polar factorization, called Zolo-PD [31]. The convergence order is $2r+1$, and $r$ is related to the condition number of matrix $A$. For ill-conditioned matrices, $r$ should be large for fast convergence.

The following theorem shows that Eq. (8) can be written in partial fraction form, which endows Zolo-PD with good parallelism. Each QR factorization in (10) can be computed simultaneously, and therefore the computation time can be reduced significantly. For example, when $r = 8$, we divide all processes into 8 groups, and each group computes one QR factorization in (10), instead of using all processes to compute 8 QR factorizations sequentially. Compared with QDWH-PD, another advantage of Zolo-PD is that it requires much fewer iterations, about two-thirds fewer, see [30], since QDWH requires six iterations for ill-conditioned matrices.

**Theorem 2.1** ([31]). *For the function $\hat{Z}_{2r+1}(x;\ell)$ as in (8), and a matrix $X$ with SVD $X = U\cdot\mathrm{diag}(\sigma_i)\cdot V^*$, the matrix $\hat{Z}_{2r+1}(X;\ell) := U\cdot\mathrm{diag}(\hat{Z}_{2r+1}(\sigma_i;\ell))\cdot V^*$ is equal to*

$$\hat{Z}_{2r+1}(X;\ell) = \hat{M}\left(X + \sum_{j=1}^{r} a_j X(X^*X + c_{2j-1}I)^{-1}\right), \tag{9}$$

*where $a_j = -\left(\prod_{k=1}^{r}(c_{2j-1}-c_{2k})\right)\cdot\left(\prod_{k=1,k\neq j}^{r}(c_{2j-1}-c_{2k-1})\right)$. Moreover, $\hat{Z}_{2r+1}(X;\ell)$ can be computed in an inverse-free manner as*

$$\begin{cases} \begin{bmatrix} X \\ \sqrt{c_{2j-1}}I \end{bmatrix} = \begin{bmatrix} Q_{j1} \\ Q_{j2} \end{bmatrix} R_j, \\ \hat{Z}_{2r+1}(X;\ell) = \hat{M}\left(X + \sum_{j=1}^{r}\frac{a_j}{\sqrt{c_{2j-1}}}Q_{j1}Q_{j2}^*\right). \end{cases} \tag{10}$$

A table is given in [31] to guide the choice of $r$. It is suggested in [31] to choose $r$ such that it requires at most two iterations. This strategy is not suitable for the distributed parallel environment, which may require much more computational resources. We will further discuss it below in Section 4. Another problem with using the QDWH-PD and Zolo-PD algorithms is to estimate two parameters $\alpha$ and $\beta$ such that $0 < \beta \le \sigma_{\min}(A) \le \sigma_{\max}(A) \le \alpha$. It can be computed easily by $\alpha = \|A\|_F$ and $\beta = (\gamma\sqrt{n})^{-1}$, where $\gamma$ is the 1-norm estimate of $A^{-1}$, as suggested in [30, Section 4].

### 2.3. Zolo-SVD

A framework for computing the SVD via the polar decomposition and the eigendecomposition has been proposed in [22,33]. It first computes the polar decomposition of $A$, $A = Q_pH$ and then computes the symmetric eigendecomposition of $H$, $H = V\Sigma V^*$. Finally, the SVD of $A$ is obtained from $A = (Q_pV)\Sigma V^*$.

Many algorithms have been proposed for this approach, and their differences lie in how to compute the polar decomposition and the symmetric eigendecomposition. In [22], it suggests using a method based on Padé iteration for the polar decomposition and any standard method for the symmetric eigendecomposition. In [33], it computes the polar decomposition by the QDWH algorithm and the symmetric eigendecomposition by QDWH-EIG which is a spectral divide-and-conquer algorithm based on the polar decomposition, see [33] for details. In [37], it is shown that QDWH-EIG is not as efficient as the symmetric eigendecomposition routines in MAGMA on GPUs. The results in [38] also show that QDWH-EIG would be slower than ELPA for ill- and well-conditioned matrices on distributed memory systems. Therefore, we also use ELPA to compute the eigendecomposition of $H$ in our implementation. The SVD algorithm in [31] is based on Zolo-PD which is implemented in Matlab and there are no results about its performance on supercomputers.

The framework for computing SVD used in this paper is summarized in Algorithm 1. We use Zolo-PD to compute the polar decomposition and use ELPA [2] to compute the symmetric eigendecomposition on the distributed memory parallel computers.

One of the main differences between ELPA and the DC algorithm in ScaLAPACK is that ELPA uses a two-stage approach for tridiagonalization. Compared with ScaLAPACK, ELPA has better scalability and can be up to two times faster on Tianhe-2A supercomputer, see [28] for results on other supercomputers. Unfortunately, there are few packages that have efficiently implemented the two-stage bidiagonal reduction (BRD) algorithm. We guess that the two-stage BRD extended to the distributed environment systems can achieve similar speedups as the tridiagonal reduction (TRD), which will be our future work. For multicore architectures, PLASMA has provided a high-performance BRD, which achieves up to 30-fold speedup [26] on a 16-core Intel Xeon machine against the LAPACK implementation in Intel MKL version 10.2.

In this work, we are concerned with computing the SVD of (nearly) square matrices. For highly rectangular matrices, many efficient algorithms are based on randomized techniques [20]. Another approach is to compute the SVD *incrementally*, such as the SVD-updating algorithm in [40]. A hierarchically incremental SVD approach is proposed in [25], which is similar in spirit to the CAQR factorization [12]. It divides the matrix $A$ into many chunks along the column dimension, $A = [A_1|A_2|\cdots|A_m]$. The SVD of each $A_i$ can be computed independently by using different process groups and then merged hierarchically together. Similar to Zolo-PD, it is naturally parallelizable.

**Algorithm 1.** [Zolo-SVD]Input: A general matrix $A \in \mathbb{R}^{m\times n}$ with $m \ge n$.

Output: the SVD $A = U\Sigma V^*$.

1. Compute the polar decomposition $A = Q_pH$ via Zolo-PD.
2. Compute the symmetric eigendecomposition $H = V\Sigma V^*$ via ELPA.
3. Compute $U = Q_pV$.

## 3. Implementation details

The serial Zolo-PD algorithm requires more flops than QDWH, see [31], but it is naturally parallelizable. We can use many MPI processes and implement Zolo-PD in parallel. We use $r$ process groups to compute $r$ QR factorizations or Cholesky factorizations simultaneouly. Therefore, the operations of each iteration cost by each process group is about the same as QDWH. By comparing with the number of iterations required, the parallel version of Zolo-PD is supposed to be about three times faster than QDWH, which is also confirmed by the results in Section 4. Note that the idea of exploiting the sparsity structure is not new, which has been mentioned in [32, Appendix A.1] and implemented in [36] on manycore architectures. In this section, we firstly introduce how

**Table 1**
Comparison of the structured QR algorithms with ScaLAPACK.

| No.Proc | Matrix $10000 \times 5000$ | | | | | |
|---|---|---|---|---|---|---|
| | PDGEQRF | MPDGEQRF | Speedup | PDORGQR | MPDORGQR | Speedup |
| 256 | 0.91 | 0.69 | 1.32 | 0.39 | 0.27 | 1.43 |
| 512 | 0.89 | 0.66 | 1.34 | 0.30 | 0.20 | 1.51 |
| 1024 | 0.76 | 0.64 | 1.23 | 0.21 | 0.16 | 1.26 |
| 2048 | 0.87 | 0.75 | 1.18 | 0.15 | 0.13 | 1.21 |
| | Matrix $20000 \times 10000$ | | | | | |
| 256 | 3.36 | 2.46 | 1.36 | 1.98 | 1.37 | 1.45 |
| 512 | 2.91 | 2.14 | 1.36 | 1.45 | 0.97 | 1.49 |
| 1024 | 2.08 | 1.60 | 1.30 | 0.88 | 0.61 | 1.46 |
| 2048 | 2.00 | 1.57 | 1.28 | 0.69 | 0.47 | 1.47 |

to exploit the sparsity structure by modifying the routines in ScaLAPACK. It turns out that our structured QR algorithm can be up to 1.5 times faster than the one in ScaLAPACK. We then introduce our implementation of Zolo-SVD based on ScaLAPACK in Section 3.2.

### 3.1. Fast structured QR algorithms

In the Zolo-PD alorithm (see (10)), we need to compute the QR factorization of a tall matrix $\mathcal{M} \equiv \begin{bmatrix} X \\ I \end{bmatrix}$, where $X \in \mathbb{R}^{m\times n}$ is a general matrix, and $I \in \mathbb{R}^{n\times n}$ is an identity matrix. The main idea of the structured QR (Struct-QR) algorithm is to take advantage of the sparsity of the bottom part of matrix $\mathcal{M}$, i.e., the identity matrix $I$. While, the classical QR will ignore all the zeros in matrix $I$ and treat it as a dense matrix. By exploiting this special sparsity structure, each Householder reflector in the Struct-QR algorithm has at most $m+1$ nonzero elements instead of $m+n$ in the classical one. Therefore, it could save a lot of floating-point operations by exploiting this sparsity structure. We introduce how to exploit the sparsity structure of $\mathcal{M}$ block column by block column below. Different from the method suggested in [32], our implementation can be seen as a block version of it, and every NB columns are transformed together. Therefore, each Householder reflector in our implementation has at most $m$+NB nonzero elements.

In this work, we implement it by modifying the routines in ScaLAPACK, PDGEQRF and PDORGQR. The routine PDGEQRF computes a QR factorization of a general matrix $A$ by computing the QR factorization of the first NB columns and then the next NB columns, and so on, where NB is the block size. The process of Struct-QR is similar to that shown in Fig. 1, which is denoted by MPDGEQRF, for simplicity. The difference between MPDGEQRF and PDGEQRF is that the row dimensions of the panels in MPDGEQRF are at most $m$+NB, where NB $\ll n$. Note that we use "row dimension" to denote the number of rows of a matrix. While, the row dimensions of panels in PDGEQRF could be much larger than MPDGEQRF. For example, the row dimension of the first panel of PDGEQRF is $m+n$. The ScaLAPACK routine PDORGQR can be similarly modified to generate the orthogonal matrix $Q$ computed by MPDGEQRF, and the structured version is denoted by MPDORGQR.

To compare the structured QR factorization with ScaLAPACK routines, we use two random matrices $\mathcal{M}$ with different dimensions which are $10000 \times 5000$ and $20000 \times 10000$, respectively. The experiments are done on Tianhe-2A super computer, located in Guangzhou, China. Each compute node has two Intel Xeon E5 2692-v2 CPUs and 24 cores in total. This experiment is done in MPI-only mode, and it runs 24 MPI processes on each node in principle. We use 256, 512, 1024 and 2048 processes to test these four routines MPDGEQRF, MPDORGQR, PDGEQRF and PDORGQR. The execution times are shown in Table 1, from which we can see that the speedups over ScaLAPACK are from 1.18 to 1.51.

### 3.2. Implementation based on ScaLAPACK

Our implementation highly depends on the ScaLAPACK and BLACS routines, and therefore it is portable. We use BLACS routines to split the communicators and perform the communications among all the processes. The algorithm proposed here works for both sparse and dense matrices. Note that for sparse matrices our algorithm does not exploit their sparsity properties when comput-

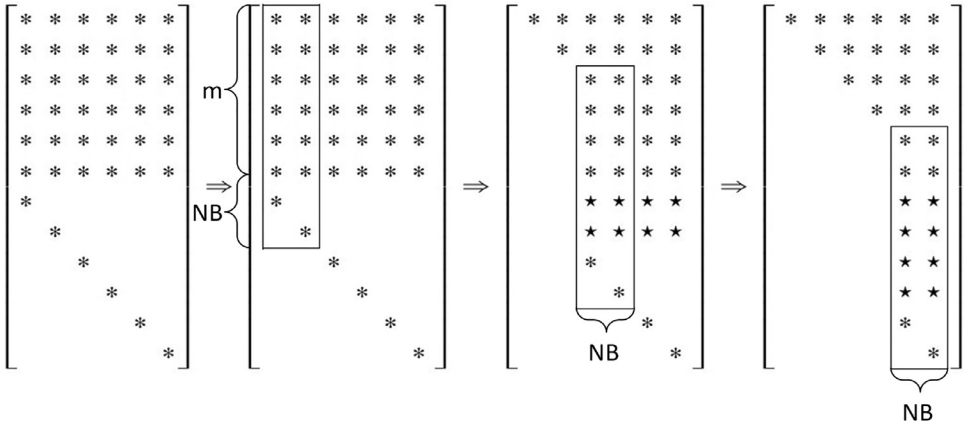



**Fig. 1.** The procedure of structured QR factorization.

ing its SVD, as done in ScaLAPACK, since Zolo-SVD can be seen as a direct method for computing the SVD.

**Algorithm 2.** [Distributed Parallel Zolo-SVD] Input: A sparse or dense matrix $A \in \mathbb{R}^{n\times n}$. Assume $A$ is distributed among $np$ processes, organized as $nprow \times npcol$, and their corresponding BLACS context is ALL_CONTXT.

Output: The SVD of $A = U\Sigma V^T$.

1. Compute $\alpha$, and $X_1 = A/\alpha$, involving all the processes in ALL_CONTXT;
2. Compute $\beta$ and let $\ell = \beta$, involving all the processes in ALL_CONTXT;
3. Choose $r$ based on $\kappa = \ell^{-1}$ or fix $r = 2$ or 3.
   - Use BLACS routine BLACS_GRIDMAP to divide all the processes into $r$ subgroups, $np = r \times sep_np$, and their BLACS context is TOP_CONTXT;
   - The $sep_np$ processes in the same row of TOP_CONTXT are further organized as $sprow \times spcol$, and associated with another context SEP_CONTXT;
4. While not convergent, involving all the processes in each SEP_CONTXT
   (a) Compute $c_i$ and $a_j$ as defined in (7) and (9);
   (b) Compute $\hat{M} = \prod_{j=1}^{r}(1 + c_{2j-1})/(1 + c_{2j})$;
   (c) The processes in the same SEP_CONTXT compute the QR or Cholesky factorization

$$\begin{bmatrix} X_1 \\ \sqrt{c_{2j-1}}I \end{bmatrix} = \begin{bmatrix} Q_{j1} \\ Q_{j2} \end{bmatrix} R_j,\ T_j = \frac{a_j}{\sqrt{c_{2j-1}}} Q_{j1} Q_{j2}^*, \tag{11}$$

   or

$$\begin{cases} Z_{2j-1} = X_1^* X_1 + c_{2j-1} I, \\ T_j = a_j (X_1 L_{2j-1}^{-1}) L_{2j-1}^{-*}), \end{cases} \tag{12}$$

   where $L_{2j-1} = \mathrm{Chol}(Z_{2j-1})$.
   (d) The $r$ subgroups together compute,

$$X_2 := \hat{M}\left(X_1 + \sum_{j=1}^{r} T_j\right), \tag{13}$$

   involving all the processes in the same column of TOP_CONTXT.
   (e) Verify that $\frac{\|X_2 - X_1\|_F}{\|X_2\|_F} \le \epsilon^{1/(2r+1)}$ holds. If not, $X_1 \leftarrow X_2$ and update $\ell := \hat{M}\ell \prod_{j=1}^{r}(\ell + c_{2j})/(\ell^2 + c_{2j-1})$;
1. End while
5. Choose one SEP_CONTXT to redistribute $X_2$ to all the processes in ALL_CONTXT;
6. Compute $H = \frac{1}{2}(X_2^* A + (X_2^* A)^*)$, and its EVD $H = V\Sigma V^*$ via ELPA or ScaLAPACK, involving all the processes in ALL_CONTXT;
7. Compute the singular vectors $U = X_2 V$ via PDGEMM, involving all the processes in ALL_CONTXT.

The parallel Zolo-SVD algorithm is summarized in Algorithm 2, which can be easily changed to an executable ScaLAPACK routine. There are three different MPI process grids, and two BLACS contexts are associated with all the processes, which are ALL_CONTXT and TOP_CONTXT. Their process grids are $nprow \times npcol$ and $r \times sep_np$, respectively. The processes in the same row of TOP_CONTXT are further organized into a $sprow \times spcol$ 2D grid, and the associated BLACS context is SEP_CONTXT. We assume the whole matrix is firstly distributed among all the processes, and then it is redistributed to the processes in each SEP_CONTXT by calling the BLACS routine PDGEMR2D. Then, each individual QR factorization in (11) or Cholesky factorization in (12) can be computed by the processes in each SEP_CONTXT simultaneously. Note that the main computational part of Zolo-PD lies in computing the QR and Cholesky factorizations.

The computation of $X_2$, defined in equation (13), requires a summation of $r$ terms, and each term is computed by a individual subgroup of processes. To compute the summation, communications are required. In our implementation, the summation is done by using the BLACS routine DGSUM2D. We only need to add the corresponding data in the same column of the process grid associated with TOP_CONTXT. This is because the data distribution of $\{T_j\}'s$ among the processes in every subgroup is the same. The times of data redistribution are very small, see the results in Table 6. For a sparse matrix $A$, we can also let each process have one copy of $A$ when its storage is not large, stored in its sparse form. This can avoid one time of communication.

We test the performance of Zolo-PD by using two different ways of choosing $r$. Firstly, $r$, the number of subcommunicators, is obtained as suggested in [31] based on the condition number of $A$. This strategy works well for the well-conditioned matrices. However, for the ill-conditioned matrices it may require too many computational resources, for example, when $r = 8$. Another strategy is to choose a small $r$ but use more iterations. In our implementation, we choose $r = 2$ or 3. Comparing with the first strategy, the number of iterations of the second one would increase by one or two. The convergence rate of Zolo-PD is reduced, but it is still of order 5 or 7.

## 4. Numerical results

Our experiments are performed on Tianhe-2A supercomputer located in Guangzhou, China, which is one of the fastest supercomputers in the world, having a peak performance of 100.7 petaflops in theory and 61.44 petaflops in Linpack benchmark. It has a total of 16,000 compute nodes. Each compute node is equipped with two Intel E5-2692 CPUs, 64GB of DDR3 main memory. The interconnect network topology is an opto-electronic hybrid, hierarchical fat tree. For compilation we used Intel fortran compiler (ifort) and the optimization flag -O3 -mAVX, and linked the codes to Intel MKL (composer_xe_2015.1.133). As suggested in [38], we only investigate pure MPI implementation, and set NB = 64 for the two-dimensional block cyclic data distribution (BCDD). Each compute node uses 24 MPI processes in principle.

**Example 1** We first use three sparse matrices with medium size from real applications, which are obtained from the University of Florida sparse matrix collection [9]. The names of these matrices are illustrated in Table 2.

The number of iterations of Zolo-PD depends on $r$. The last column of Table 2 shows the values of $r$ suggested by [31] based on the condition numbers. When $r$ is larger, Zolo-PD has higher convergence rate. Table 3 shows the number of iterations by Zolo-PD in our experiments. From it we can see that when $r = 3$ or 4 Zolo-PD requires two fewer iterations than QDWH-PD, which ex-

**Table 2**
Summary of basic matrix characteristics.

| Matrix | $n$ | $nnz$ | Condition number | $r$ |
|---|---|---|---|---|
| nemeth03 | 9506 | 202,157 | 1.29e+00 | 2 |
| fv1 | 9604 | 47,434 | 1.40e+01 | 3 |
| linverse | 11,999 | 53,988 | 9.06e+03 | 4 |

**Table 3**
The number of iterations required by QDWH-PD and Zolo-PD.

| Matrix | QDWH | $r = 2$ | $r = 3$ | $r = 4$ |
|---|---|---|---|---|
| nemeth03 | 4 | 3 | 3 | 3 |
| fv1 | 5 | 4 | 3 | 3 |
| linverse | 5 | 4 | 3 | 3 |

**Table 4**
Time comparisons of PDGESVD with Zolo-SVD.

| Matrix | $n$ | Method | Number of processes | | | | |
|---|---|---|---|---|---|---|---|
| | | | 256 | 512 | 1024 | 2048 | 4096 |
| nemeth03 | 9,506 | PDGESVD | 61.45 | 15.97 | 15.14 | 14.85 | 15.57 |
| | | Zolo-SVD | 24.73 | 15.11 | 10.91 | 11.83 | 10.01 |
| | | Speedup | 2.48 | 1.06 | 1.39 | 1.26 | 1.56 |
| fv1 | 9,604 | PDGESVD | 95.60 | 20.18 | 19.06 | 18.11 | 19.03 |
| | | Zolo-SVD | 30.65 | 17.58 | 12.82 | 12.65 | 11.54 |
| | | Speedup | 3.12 | 1.15 | 1.49 | 1.43 | 1.65 |
| linverse | 11,999 | PDGESVD | 144.37 | 29.87 | 26.43 | 25.01 | 32.38 |
| | | Zolo-SVD | 50.29 | 28.64 | 19.62 | 18.64 | 15.83 |
| | | Speedup | 2.87 | 1.04 | 1.35 | 1.34 | 2.05 |

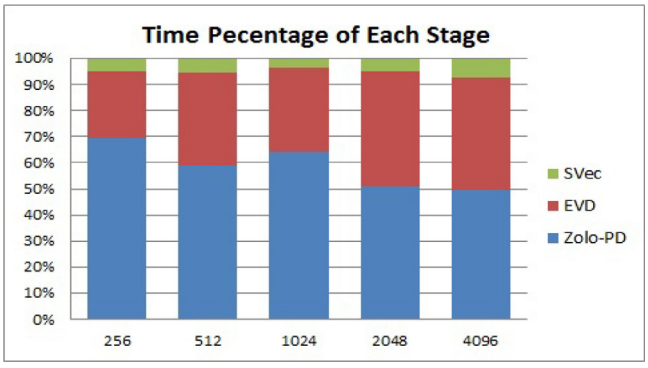



**Fig. 2.** The time percentages spent in each step of Algorithm 2.

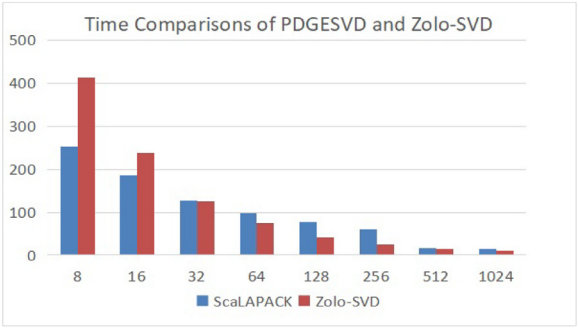



**Fig. 3.** The time comparisons of PDGESVD and Zolo-SVD for nemeth03.

plains the speedups of Zolo-PD over QDWH-PD in some sense when implemented in parallel.

The comparison results of Zolo-SVD and PDGESVD are shown in Table 4. Note that the results in Table 4 are obtained by using the same number of processes and choosing $r = 2$ in Zolo-SVD for all these three matrices. It shows that Zolo-SVD can be 3.12x faster than PDGESVD for matrix fv1 when using 256 processes. Note that the number of processes used by PDGESVD and Zolo-SVD are shown in the first rows of Table 4. Fig. 2 shows the time percentages spent in each step of Algorithm 2, where Zolo-PD denotes the polar decomposition step, EVD denotes computing the eigendecomposition of $H$ by using ELPA, and SVec denotes computing the singular vectors via PDGEMM. From it we can see that most work lies in computing Zolo-PD, and that as the number of processes increases, EVD takes more percentage of time. The comparisons of QDWH-PD and Zolo-PD are shown in Table 5. It turns out that Zolo-PD is about 1.20x times faster than QDWH-PD when using many processes.

The results in Table 4 show that the timings of PDGESVD and Zolo-SVD both decrease as the number of processes increases and the decrease of Zolo-SVD is faster than PDGESVD. It shows that Zolo-SVD has better strong scalability than PDGESVD, for example, the timings of PDGESVD for matrix nemeth03 decrease very slowly from 512 to 4096 processes. When looking into the scalability of PDGESVD, we find that superlinear speedup happens for PDGESVD, for example, see the timings of matrix nemeth03 in Table 4 from 256 to 512 processes. Note that the results shown in the rows denoted by 'Speedup' in Table 4 are defined as $\frac{Time(PDGESVD)}{Time(Zolo-SVD)}$ in this example. It makes the speedup of Zolo-SVD over PDGESVD go down first and then up. We have performed this experiment for many times, include using different versions of Intel MKL. The routine PDGESVD uses the implicit zero-shift QR algorithm [13] to compute the SVD of a bidiagonal matrix. It calls the LAPACK routine DBDSQR, which becomes much faster when using 512 processes than using 256 processes. Since Zolo-SVD requires more floating-point operations than PDGESVD, Zolo-SVD can be slower than PDGESVD when using few processes. It is verified by Fig. 3 for matrix nemeth03 and processes from 8 to 1024, and Zolo-SVD is slower than PDGESVD when using 8 and 16 processes.

In this example we further use the matrix nemeth03 to show the benefits of the Struct-QR algorithm proposed in Section 3.1. For this matrix, Zolo-PD takes one QR factorization and two Cholesky factorizations. The time comparisons of Struct-QR and QR in ScaLAPACK are shown in Fig. 4, where the speedups of Struct-QR over

**Table 5**
Times of QDWH-PD and Zolo-PD ($r = 2$) when using the same number of processes.

| Matrix | Method | Number of processes | | | | |
|---|---|---|---|---|---|---|
| | | 256 | 512 | 1024 | 2048 | 4096 |
| nemeth03 | QDWH-PD | 16.51 | 12.02 | 8.14 | 7.36 | 5.55 |
| | Zolo-PD | 16.56 | 8.55 | 6.42 | 4.79 | 4.44 |
| | Speedup | 1.00 | 1.41 | 1.27 | 1.70 | 1.25 |
| fv1 | QDWH-PD | 16.16 | 12.75 | 8.62 | 7.23 | 5.97 |
| | Zolo-PD | 22.26 | 10.87 | 7.40 | 6.02 | 5.82 |
| | Speedup | 0.73 | 1.17 | 1.16 | 1.20 | 1.03 |
| linverse | QDWH-PD | 37.63 | 20.25 | 12.97 | 10.15 | 9.35 |
| | Zolo-PD | 37.40 | 19.07 | 12.43 | 8.75 | 7.92 |
| | Speedup | 1.01 | 1.06 | 1.04 | 1.16 | 1.18 |

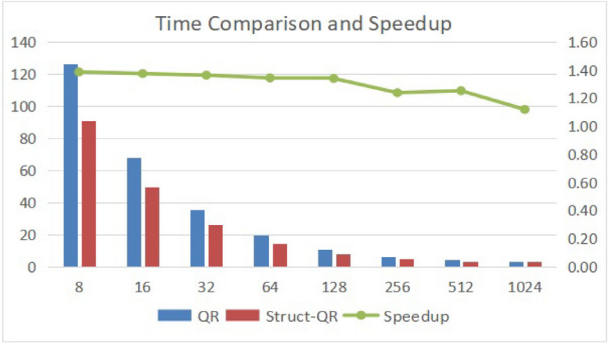



**Fig. 4.** The time comparisons of QR and Struct-QR for nemeth03.

**Table 6**
Profiling the computational stages of Zolo-PD for matrix fv1. Zolo-PD requires three iterations and the times are in second.

| No. Proc | 256 | 512 | 1024 |
|---|---|---|---|
| QR | 3.42 | 3.03 | 2.07 |
| Combin. | 6.64e-02 | 7.21e-03 | 6.26e-02 |
| FormX2 | 2.82e-02 | 1.98e-02 | 1.47e-02 |
| Chol. | 1.95 | 1.38 | 0.89 |
| Combin. | 1.74e-02 | 1.84e-02 | 3.80e-02 |
| FormX2 | 1.96e-02 | 1.38e-02 | 1.26e-02 |
| Chol. | 1.89 | 1.22 | 0.75 |
| Combin. | 1.34e-02 | 1.20e-02 | 1.03e-03 |
| FormX2 | 1.87e-02 | 1.33e-02 | 1.38e-02 |

**Table 7**
Summary of basic matrix characteristics.

| Matrix | $n$ | $nnz$ | Condition number |
|---|---|---|---|
| bcsstk18 | 11,948 | 80,519 | 3.46e+11 |
| c-47 | 15,343 | 113,372 | 3.16e+08 |
| c-49 | 21,132 | 89,087 | 6.02e+08 |
| cvxbqp1 | 50,000 | 199,984 | 1.09e+11 |
| rand1 | 10,000 | dense | 3.97e+07 |
| rand2 | 30,000 | dense | 1.24e+07 |

QR are also included. The results are similar to those in Table 1, and Struct-QR is usually faster than QR by about 20%–40%. Struct-QR is always faster than the QR in ScaLAPACK. Since Struct-QR requires much fewer floating-point operations than QR, the execution time is reduced significantly when using few processes. When using eight processes, Struct-QR is faster than QR by 35 s. While, the benefit of using Struct-QR is not much when using many processes. Struct-QR is only faster than QR by 0.5 s when using 1024 processes.

**Example 2** The main drawback of Zolo-PD is that it requires too many floating-point operations for ill-conditioned matrices when requiring at most TWO iterations. One approach to fix this problem is to choose a small $r$. In this example we let $r$ be 2 and let QDWH-PD and Zolo-PD use *the same number* of processes, and the results are shown in Table 5. The first rows of Table 5 show the number of processes used. From it, we can get that Zolo-PD is usually faster than QDWH-PD. Because Zolo-PD decomposes all MPI processes into $r$ relatively independent groups, it is more loosely coupled and therefore more scalable than QDWH-PD. Zolo-PD becomes faster than QDWH-PD when using many processes.

Compared with QDWH-PD, Zolo-PD further requires communications among different subcommunicators. We profile the Zolo-PD algorithm and Table 6 shows the times cost by each stage of Zolo-PD, where the rows Combin. show the communication time cost by DGSUM2D, which computes the summation of $\sum_{j=1}^{r} T_j$ in Algorithm 2. The rows FormX2 show the maximum time of computing $X_2$ of all subgroups. The results were obtained for the matrix fv1 when let $r = 3$, and Zolo-PD took three iterations, the first one used QR factorization and the other two used Cholesky factorization. From it we can see that most time lies in computing the QR and Cholesky factorizations and the communication times between different communicators are negligible.

**Example 3** We use some larger and more ill-conditioned matrices to further test Zolo-SVD and compare it with PDGESVD. The properties of these matrices are shown in Table 7, where rand1 and rand2 are two matrices with random Gaussian entries. These matrices include symmetric ones and nonsymmetric ones, sparse ones and dense ones. Some matrices are very ill-conditioned and their condition numbers are in the order of $10^{11}$.

In this example we let PDGESVD and Zolo-SVD use the same number of processes and let $r = 2$, and the numerical results are shown in Table 8, where the first row contains the number of processes used and the other rows contain the times cost by PDGESVD and Zolo-SVD, and the speedups of Zolo-SVD over PDGESVD. It turns out that Zolo-SVD is always faster than PDGESVD for these matrices when using many MPI processes, for example using 4096 processes. Since the speedups of QDWH-SVD over PDGESVD are similar to those of Zolo-SVD, the results are not included in Table 8.

About how to choose $r$, it depends on the computational resources you have and the condition number of matrix. The number of iterations by Zolo-PD when choosing different $r$ are illustrated in Table 9, which are consistent with the results in [31]. It only decreases by one iteration when $r$ is increased from 2 to 5, and therefore $r = 2$ or 3 is probably a good choice. Another reason is that taking $r$ too large can lead to numerical instability [31].

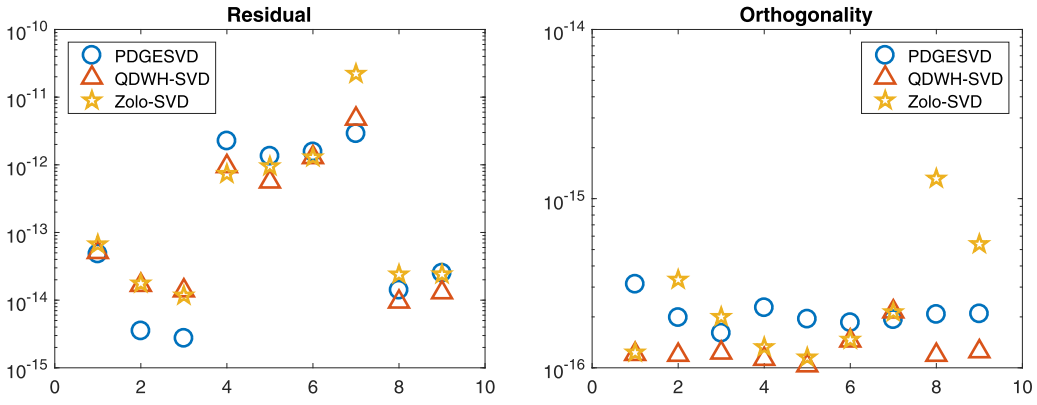
### 4.1. Numerical accuracy

The backward error of the overall SVD is computed as

$$Residual = \frac{\|A - U\Sigma V^*\|_F}{\|A\|_2}, \tag{14}$$

**Table 8**
Time comparisons of PDGESVD with Zolo-SVD ($r = 2$).

| Matrix | $n$ | Methods | Number of processes | | | | |
|---|---|---|---|---|---|---|---|
| | | | 256 | 512 | 1024 | 2048 | 4096 |
| bcsstk18 | 11,948 | PDGESVD | 71.2 | 18.5 | 17.7 | 16.8 | 17.1 |
| | | Zolo-SVD | 48.9 | 27.8 | 20.9 | 14.1 | 13.3 |
| | | Speedup | 1.45 | 0.67 | 0.85 | 1.19 | 1.29 |
| c-47 | 15,343 | PDGESVD | 125.6 | 38.5 | 27.5 | 27.0 | 26.1 |
| | | Zolo-SVD | 89.7 | 49.3 | 38.1 | 23.8 | 21.4 |
| | | Speedup | 1.40 | 0.78 | 0.72 | 1.13 | 1.22 |
| c-49 | 21,132 | PDGESVD | 356.4 | 337.9 | 82.6 | 73.3 | 66.7 |
| | | Zolo-SVD | 279.0 | 111.8 | 76.7 | 47.6 | 38.6 |
| | | Speedup | 1.28 | 3.02 | 1.08 | 1.54 | 1.73 |
| cvxbqp1 | 50,000 | PDGESVD | 2.6e3 | 1.5e3 | 9.1e2 | 6.2e2 | 4.2e2 |
| | | Zolo-SVD | 1.9e3 | 1.1e3 | 7.8e2 | 4.1e2 | 3.3e2 |
| | | Speedup | 1.37 | 1.36 | 1.17 | 1.51 | 1.28 |
| rand1 | 10,000 | PDGESVD | 98.3 | 20.1 | 18.3 | 18.2 | 18.6 |
| | | Zolo-SVD | 30.6 | 18.2 | 14.8 | 10.9 | 10.4 |
| | | Speedup | 3.21 | 1.10 | 1.24 | 1.70 | 1.79 |
| rand2 | 30,000 | PDGESVD | 1024.7 | 916.5 | 228.5 | 168.5 | 127.5 |
| | | Zolo-SVD | 465.3 | 266.1 | 189.0 | 105.4 | 79.0 |
| | | Speedup | 2.20 | 3.44 | 1.21 | 1.60 | 1.61 |




(a) Residual (b) $\frac{\|I-UU^*\|_F}{n}$

**Fig. 5.** The residual of the computed SVD, and the orthogonality of the computed left singular vectors.

**Table 9**
The number of iterations by Zolo-PD when choosing different $r$.

| Matrix | $r$ | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| bcsstk18 | 4 | 4 | 3 | 3 |
| c-47 | 4 | 4 | 3 | 3 |
| c-49 | 4 | 4 | 3 | 3 |
| linverse | 4 | 3 | 3 | 3 |
| nemeth03 | 3 | 3 | 3 | 3 |
| fv1 | 4 | 3 | 3 | 3 |

where $U \in \mathbb{C}^{n\times n}$, $V \in \mathbb{C}^{n\times n}$ are orthogonal and $\Sigma \in \mathbb{C}^{n\times n}$ is diagonal and its diagonals are the singular values, $\|X\|_F$ denotes the Frobenius norm of matrix $X$ and $\|X\|_2$ denotes the 2-norm of $X$. We test the orthogonality of the computed singular vectors by measuring

$$Orthogonality = \frac{\|I - WW^*\|_F}{n},$$

where $W$ equals to $U$ or $V$, and $n$ is the dimension of matrix $A$.

We compare these three methods, PDGESVD, QDWH-SVD and Zolo-SVD, and the numerical results are shown in Fig. 5. The matrices tested are bcsstk18, c-47, c-49, fv1, linverse, nemeth03, cvxbqp1, rand1, and rand2, which are numbered from 1 to 9 in the subfigures of Fig. 5, respectively. The results illustrate that the algorithm is numerically stable, and the computed left and right singular vectors are highly orthogonal. The relative residuals, defined in (14), of the computed SVD by Zolo-SVD and QDWD-SVD are as accurate as those computed by PDGESVD for these matrices, and the results are shown in Fig. 5(a). Zolo-SVD is comparable to QDWH-SVD, and their accuracy are in the same order. The orthogonality of the computed singular vectors by these three methods are always in machine precision, less than $1.0e-15$. The results for the right and left singular vectors are similar, and only the results for the left singular vectors are included in Fig. 5(b).

## 5. Conclusions

A new distributed parallel SVD algorithm, called *Zolo-SVD*, is implemented in this work, which is based on the Zolo-PD [31] algorithm and the symmetric eigenvalue decomposition algorithm. The main advantage of this algorithm is that it is highly scalable. When using many computational resources, numerical results show that Zolo-SVD is about two times faster than PDGESVD. Compared with QDWH-PD, Zolo-PD has higher convergence rate and the drawback is that Zolo-PD requires more floating-point operations. To be faster, it is better to implement Zolo-SVD in parallel. We use many sparse matrices from the University of Florida sparse matrix collection and some random dense matrices to conduct the numerical experiments. Our implementations of the QDWH-SVD, Zolo-SVD and Struct-QR algorithms are freely available at https://github.com/shengguolsg/Zolo-SVD.

## Acknowledgments

The authors are very grateful to the anonymous referees for their valuable suggestions, which greatly improve the presentation. This work is supported by National Natural Science Foundation of China (no. 11401580, U1811464 and 91530324), Program for Guangdong Introducing Innovative and Enterpreneurial Teams (no. 2016ZT06D211), NSF of Hunan (no. 806175290082) and NSF of NUDT (no. ZK18-03-01).

## References

[1] N.I. Akhiezer., Elements of the Theory of Elliptic Functions, AMS, Providence, RI, 1990.
[2] T. Auckenthaler, V. Blum, H.J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, P.R. Willems, Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations, Parallel Comput. 37 (12) (2011) 783–794.
[3] G. Ballard, J. Demmel, O. Holtz, O. Schwartz, Minimizing communication in numerical linear algebra, SIAM J. Matrix Anal. Appl. 21 (2) (2011) 562–580.
[4] J.L. Barlow, N. Bosner, Z. Drmač, A new stable bidiagonal reduction algorithm, Linear Algebra Appl. 397 (2005) 35–84.
[5] J.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK Users' Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.
[6] N. Bosner, J.L. Barlow, Block and parallel versions of one-sided bidiagonalization, SIAM J. Matrix Anal. Appl. 29 (3) (2007) 927–953.
[7] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, Scalapack: a portable linear algebra library for distributed memory computers-design issues and performance, Comput. Phys. Commun. 97 (1996) 1–15.
[8] D. Sukkari, H. Ltaief, A. Esposito, D. Keyes, A QDWH-based SVD software framework on distributed-memory manycore systems. submitted to ACM transactions on mathematical software (under revision, 2017, https://repository.kaust.edu.sa/handle/10754/626212.
[9] T. Davis, Y. Hu, The Univeristy of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011). 1:1–1:25.
[10] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, R. Harshman, Indexing by latent semantic analysis, J. Soc. Inf. Sci. 41 (1990) 391–407.
[11] J. Demmel, Applied numerical linear algebra, 1997, SIAM, Philadelphia.
[12] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal parallel and sequential QR and LU factorizations, SIAM J. Sci. Comput. 34 (2012) A206–A239.
[13] J.W. Demmel, W.M. Kahan, Accurate singular values of bidiagonal matrices, SIAM J. Sci. Comput. 11 (1990) 873–912.

[14] B. Großer, B. Lang, Efficient parallel reduction to bidiagonal form, Parallel Comput. 25 (1999) 969–986.
[15] M. Faverge, J. Langou, Y. Robert, J. Dongarra, Bidiagonalization with parallel tiled algorithms, 2016, ArXiv: 1611.06892v1.
[16] G. Golub, W. Kahan, Calculating the singular values and pseudo-inverse of a matrix, SIAM J. Numer. Anal. 2 (1965) 205–224.
[17] M. Gu, S.C. Eisenstat, A divide-and-conquer algorithm for the bidiagonal SVD, SIAM J. Matrix Anal. Appl. 16 (1) (1995) 79–92.
[18] H. Ltaief, D. Sukkari, A. Esposito, Y. Nakatsukasa, D. Keyes, Massively parallel polar decomposition on distributed-memory systems. submitted to ACM transactions on parallel computing under revision, 2018, https://repository.kaust.edu.sa/handle/10754/626212.
[19] A. Haidar, P. Luszczek, J. Kurzak, J. Dongarra, An Improved Parallel Singular Value Algorithm and Its Implementation for Multicore Hardware, in: In William Gropp, editor, *Proceedings of SC13*, pages 90: 1–90:14, Denver, CO, USA, 2013.
[20] N. Halko, P.G. Martinsson, J.A. Tropp, Finding structure with randomness probabilistic algorithms for constructing approximate matrix decompositions, SIAM Rev. 53 (2011) 217–288.
[21] N.J. Higham, Functions of Matrices: Theory and Computation, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2008.
[22] N.J. Higham, P. Papadimitriou, Parallel Singular Value Decomposition via the Polar Decomposition, Technical Report Numerical Analysis Report 239, Manchester Centre for Computational Mathematics, Manchester, England, 1993.
[23] N.J. Higham, P. Papadimitriou, A New Parallel Algorithm for Computing the Singular Value Decomposition, in: J.G. Lewis (Ed.), The Fifth SIAM Conference on Applied Linear Algebra, SIAM, Philadelphia, 1994, pp. 80–84.
[24] H. Hotelling, Simplified calculation of principal components, Psychometrica 1 (1935) 27–35.
[25] M.A. Iwen, B.W. Ong, A distributed and incremental SVD algorithm for agglomerative data analysis on large networks, SIAM J. Matrix Anal. Appl. 37 (4) (2016) 1699–1718.
[26] H. Ltaief, P. Luszczek, J. Dongarra, High-performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures, ACM Trans. Math. Softw. 39 (3) (2013). 16.
[27] H. Ltaif, P. Luszczek, J. Dongarra, High performance bidiagonal reduction using tile algorithm on homogeneous multicore architectures, ACM Trans. Math. Softw. 39 (3) (2013) 1–22. 16.
[28] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H. Bungartz, H. Lederer, The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science, J. Phys.: Condens. Matter 26 (2014) 1–15.
[29] B.C. Moore, Principal component analysis in linear systems: controllability, observability, and model reduction, IEEE Trans. Autom. Control 1 (1981). AC–26.
[30] Y. Nakatsukasa, Z. Bai, F. Gygi, Optimizing Halley's iteration for computing the matrix polar decomposition, SIAM J. Matrix Anal. Appl. 31 (2010) 2700–2720.
[31] Y. Nakatsukasa, R.W. Freund, Computing fundamental matrix decompositions accurately via the matrix sign function in two iterations: the power of Zolotarev's functions, SIAM Rev. xx(x) (2016) xx–xxx.
[32] Y. Nakatsukasa, N. Higham, Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD, SIAM J. Sci. Comput. 35 (3) (2013) A1325–A1349.
[33] Y. Nakatsukasa, N. Higham, Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD, SIAM J. Sci. Comput. 35 (3) (2013) A1325–A1349.
[34] R. Ralha, One-sided reduction to bidiagonal form, Linear Algebra Appl. 358 (2003) 219–238.
[35] H.D. Simon, H. Zha, Low-rank matrix approximation using the Lanczos bidiagonalization process with applications, SIAM J. Sci. Comput. 21 (2000) 2257–2274.
[36] D. Sukkari, H. Ltaief, M. Faverge, D. Keyes, Asynchronous task-based polar decomposition on single node manycore architectures, IEEE Trans. Parallel Distrib. Syst. 29 (2) (2018) 312–323.
[37] D.E. Sukkari, H. Ltaief, D.E. Keyes, A high performance QDWH-SVD solver using hardware accelerators, ACM Trans. Math. Softw. 43 (1) (2016) 1–25.
[38] D.E. Sukkari, H. Ltaief, D.E. Keyes, High Performance Polar Decomposition on Distributed Memory Systems, in: P.F. Dutot, D. Trystram (Eds.), Euro-Par 2016, LNCS, volume 9833, Springer, Switzerland, 2016, pp. 605–616.
[39] P.R. Willems, B. Lang, C. Vömel, Computing the bidiagonal SVD using multiple relatively robust representations, SIAM J. Matrix Anal. Appl. 28 (4) (2006) 907–926.
[40] H. Zha, H.D. Simon, On updating problems in latent semantic indexing, SIAM J. Sci. Comput. 21 (1999) 782–791.