THE SINGULAR VALUE DECOMPOSITION: ANATOMY OF OPTIMIZING AN ALGORITHM FOR EXTREME SCALE[*]

JACK DONGARRA[†], MARK GATES[‡], AZZAM HAIDAR[‡], JAKUB KURZAK[‡], PIOTR LUSZCZEK[‡], STANIMIRE TOMOV[‡], AND ICHITARO YAMAZAKI[‡]

Abstract. The computation of the Singular Value Decomposition, or SVD, has a long history, with many improvements over the years, both in implementations and algorithmically. Here, we survey the evolution of SVD algorithms for dense matrices, discussing the motivation and performance impact of changes. There are two main branches of dense SVD methods: bidiagonalization and Jacobi. Bidiagonalization methods started with the implementation by Golub and Reinsch in Algol60, which was subsequently ported to Fortran in the EISPACK library, and was later more efficiently implemented in the LINPACK library, targeting contemporary vector machines. To address cache-based memory hierarchies, the SVD algorithm was reformulated to use Level 3 BLAS in the LAPACK library. To address new architectures, ScaLAPACK was introduced to take advantage of distributed computing, and MAGMA was developed for accelerators such as GPUs. Algorithmically, the divide and conquer and MRRR algorithms were developed to reduce the number of operations. Still, these methods remained memory bound, so two-stage algorithms were developed to reduce memory operations and increase the computational intensity, with efficient implementations in PLASMA, DPLASMA, and MAGMA. Jacobi methods started with the two-sided method of Kogbetliantz and the one-sided method of Hestenes. They have likewise had many developments, including parallel and block versions, and preconditioning to improve convergence. In this paper, we investigate the impact of these changes by testing various historical and current implementations on a common, modern multicore machine and a distributed computing platform. We show that algorithmic and implementation improvements have increased the speed of the SVD by several orders of magnitude, while using up to 40 times less energy.

Key words. singular value decomposition, SVD, bidiagonal matrix, QR iteration, divide and conquer, bisection, MRRR, Jacobi method, Kogbetliantz method, Hestenes method

AMS subject classifications. 15A18 15A23 65Y05

1. Introduction.

The *singular value decomposition*, or SVD, is a very powerful technique for dealing with matrix problems in general. The practical and theoretical importance of the SVD is hard to overestimate, and it has a long and fascinating history. A number of classical mathematicians are associated with the theoretical development of the SVD [107], including Eugenio Beltrami (1835–1899), Camille Jordan (1838–1921), James Sylvester (1814–1897), Erhard Schmidt (1876–1959), and Hermann Weyl (1885–1955).

In recent years, the SVD has become a computationally viable tool for solving a wide variety of problems that arise in many practical applications. The use of the SVD in these applications is centered on the fact that they require information about the rank of a matrix, or a low rank approximation of a matrix, or orthogonal bases for the row and column spaces of a matrix. Applications are as diverse as least squares data fitting [53], image compression [3], facial recognition [111], principal

[*]Submitted to the editors February 20, 2017.

Funding: This research is based upon work supported by the National Science Foundation under Grant No. 1339822, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We also received support from NVIDIA and Intel. Furthermore, we would like to thank Intel for access to their distributed computing platform for testing ScaLAPACK and DPLASMA.

[†]University of Tennessee, Oak Ridge National Laboratory, University of Manchester (dongarra@icl.utk.edu

[‡]University of Tennessee (mgates3@icl.utk.edu, haidar@icl.utk.edu, kurzak@icl.utk.edu, luszczek@icl.utk.edu, tomov@icl.utk.edu, iyamazak@icl.utk.edu)

component analysis [92], latent semantic analysis [28], and computing the 2-norm, condition number, and numerical rank of a matrix.

The SVD of an m-by-n matrix A is given by:

$$A = U\Sigma V^T \text{ } (A = U\Sigma V^H \text{ in the complex case}), \tag{1}$$

where U and V are orthogonal (unitary) matrices and Σ is an m-by-n matrix with real diagonal elements, σ_i, conventionally ordered such that:

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min(m,n)} \geq 0.$$

The σ_i are the *singular values* of A and the first $\min(m, n)$ columns of U and V are the *left* and *right singular vectors* of A, respectively.

Theoretically, the SVD can be characterized by the fact that the singular values are the square roots of the eigenvalues of $A^T A$, the columns of V are the corresponding eigenvectors, and the columns of U are the eigenvectors of AA^T, assuming distinct singular values. However, this is not a satisfactory basis for computation because roundoff errors in the formulation of $A^T A$ and AA^T often destroy pertinent information.

The key to using the SVD is the fact that it can be computed very effectively. There are two dominant categories of SVD algorithms for dense matrices: bidiagonalization methods and Jacobi methods. The classical bidiagonalization method proceeds in the following three stages:

1. The matrix A is reduced to bidiagonal form: $A = U_1 B V_1^T$ if A is real ($A = U_1 B V_1^H$ if A is complex), where U_1 and V_1 are orthogonal (unitary if A is complex), and B is real and upper-bidiagonal when $m \geq n$ or lower bidiagonal when $m < n$, so that B is nonzero on only the main diagonal and either the first superdiagonal (if $m \geq n$) or the first subdiagonal (if $m < n$).
2. The SVD of the bidiagonal matrix B is computed: $B = U_2 \Sigma V_2^T$, where U_2 and V_2 are orthogonal and Σ is diagonal as described above. Several algorithms exist for the bidiagonal SVD, the original being QR iteration.
3. If desired, the singular vectors of A are then computed as $U = U_1 U_2$ and $V = V_1 V_2$.

This is the basic, efficient, and stable algorithm as posed by Golub and Kahan in 1965 [53]. Golub and Reinsch [54] realized the first implementation of the SVD algorithm in Algol60, the programming language of the time. Their paper was later reproduced in the Wilkinson-Reinsch Handbook [117]. Bidiagonalization methods are covered in sections 3 to 11, with additional tests of accuracy and performance on various matrix types in sections 13 and 14.

In contrast, Jacobi methods apply plane rotations to the entire matrix A, without ever reducing it to bidiagonal form. Two-sided Jacobi methods, first proposed by Kogbetliantz in 1955 [76], iteratively apply rotations on both sides of A to bring it to diagonal form, while one-sided Jacobi methods, proposed by Hestenes in 1958 [68], apply rotations on one side to orthogonalize the columns of A, implicitly bring $A^T A$ to diagonal. While Jacobi methods are often slower than bidiagonalization methods, there remains interest due to their simplicity, easy parallelization, and potentially better accuracy for certain classes of matrices. Jacobi methods are covered in section 12, with additional tests in sections 13 and 14.

This manuscript traces the development of the SVD algorithm over the past 50 years, using various historical implementations. This development includes algorithmic improvements such as blocking, the divide and conquer and MRRR algorithms, a two-stage reduction, as well as adapting to new computer architectures such as distributed memory, accelerators, and multicore CPUs. We compare the performance of all the implementations on a common multicore computer. Our focus is on computing all singular values and optionally singular vectors, for both square and tall dense matrices. For bisection and MRRR methods we also compute a subset of the singular values and vectors.

2. Experimental Setup.

To test the various implementations, we ran six different tests:

1. Square matrices, singular values only (no vectors).
2. Square matrices, singular values and vectors.
3. Tall matrices, $m = 3n$, singular values only (no vectors).
4. Tall matrices, $m = 3n$, singular values and vectors.
5. Tall matrices, $m = 1000n$, singular values only (no vectors).
6. Tall matrices, $m = 1000n$, singular values and vectors.

When computing singular vectors, we computed the *reduced SVD* consisting of the first $\min(m, n)$ columns of U and V, and $\min(m, n)$ rows and columns of Σ. This is the most useful part computationally, sufficient for many applications such as solving least squares problems, and we subsequently identify U, V, and Σ with those of the reduced SVD, which still satisfy (1). For LAPACK, the reduced SVD corresponds to job="s" for both U and V. We store U and V separately from A, i.e., they do not overwrite A. Where applicable, we query for the optimal workspace size; otherwise, we use the maximum documented workspace size. This ensures that we always use the "fast" path in codes, including blocking and other optimizations.

Unless indicated, matrices have random entries from a uniform distribution on $(0, 1)$. For some tests, we generate singular values Σ according to one of the distributions below, then form $A = U\Sigma V^T$ where U and V are random orthogonal matrices from the Haar distribution [106]. Where given, κ is the condition number of A.

- Σ random: singular values are random uniform on $(0, 1)$. The condition number is not determined *a priori*.
- arithmetic: $\sigma_i = 1 - \frac{i-1}{n-1}\left(1 - \frac{1}{\kappa}\right)$ for $i = 1, \dots, n$.
- geometric: $\sigma_i = \kappa^{-(i-1)/(n-1)}$ for $i = 1, \dots, n$.
- log-random: singular values are random in $(\frac{1}{\kappa}, 1)$ such that their logarithms are random uniform on $(\log \frac{1}{\kappa}, \log 1)$.
- cluster at $\frac{1}{\kappa}$: $\Sigma = \left[1, \frac{1}{\kappa}, \dots, \frac{1}{\kappa}\right]$.
- cluster at 1: $\Sigma = \left[1, \dots, 1, \frac{1}{\kappa}\right]$.

All tests were performed in double-precision real arithmetic. Except for PLASMA and MPI-based implementations, which initialize memory in parallel, we used `numactl --interleave=all` to distribute memory across CPU sockets, and the CPU cache was flushed before the SVD function call. To avoid repeating minor differences, we shall generally assume that A is real and $m \ge n$. Operations for complex or $m < n$ are analogous.

We conducted experiments on a two-socket Intel Sandy Bridge Xeon E5-2670 running at 2.6 GHz, with 8 cores per socket, a theoretical double-precision peak of 333 Gflop/s, and 64 GiB of main memory. The measured practical dgemm peak is 313.6 Gflop/s and dgemv peak is 13.9 Gflop/s (55.8 GB/s). The STREAM triad benchmark [91] measured the memory bandwidth as 57.8 GB/s with 16 OpenMP threads. All CPU implementations were compiled with gcc and linked against Intel's Math Kernel Library (MKL) version 11.2.3 [71].

GPU results used an NVIDIA Kepler K40c with 15 multiprocessors, each containing 192 CUDA cores. The theoretical double precision peak performance is 1682 Gflop/s. On the GPU, 12 GiB of device memory can be accessed at a theoretical bandwidth of 288 GB/s. The measured practical dgemm peak is 1243.1 Gflop/s and dgemv peak is 45.3 Gflop/s (181.2 GB/s). For the GPU implementation, we used CUDA version 7.0 [94].

3. EISPACK Implementation.

The EISPACK project was an effort to develop a software library for numerical computation of eigenvalues and eigenvectors of matrices based on algorithms and ideas that were mainly contained in the Wilkinson-Reinsch Handbook. EISPACK was a transliteration of these Algol programs into Fortran. It contains subroutines for calculating the eigenvalues of nine classes of matrix problems: complex general, complex Hermitian, real general, real symmetric, real symmetric banded, real symmetric tridiagonal, special real tridiagonal, generalized real, and generalized real symmetric. In addition, it includes subroutines to perform a singular value decomposition [50]. Some routines were updated to implement improvements in the numerical accuracy and achieve portability across different computing systems. However, the basic organization and access to matrix elements was kept in the Algol style.

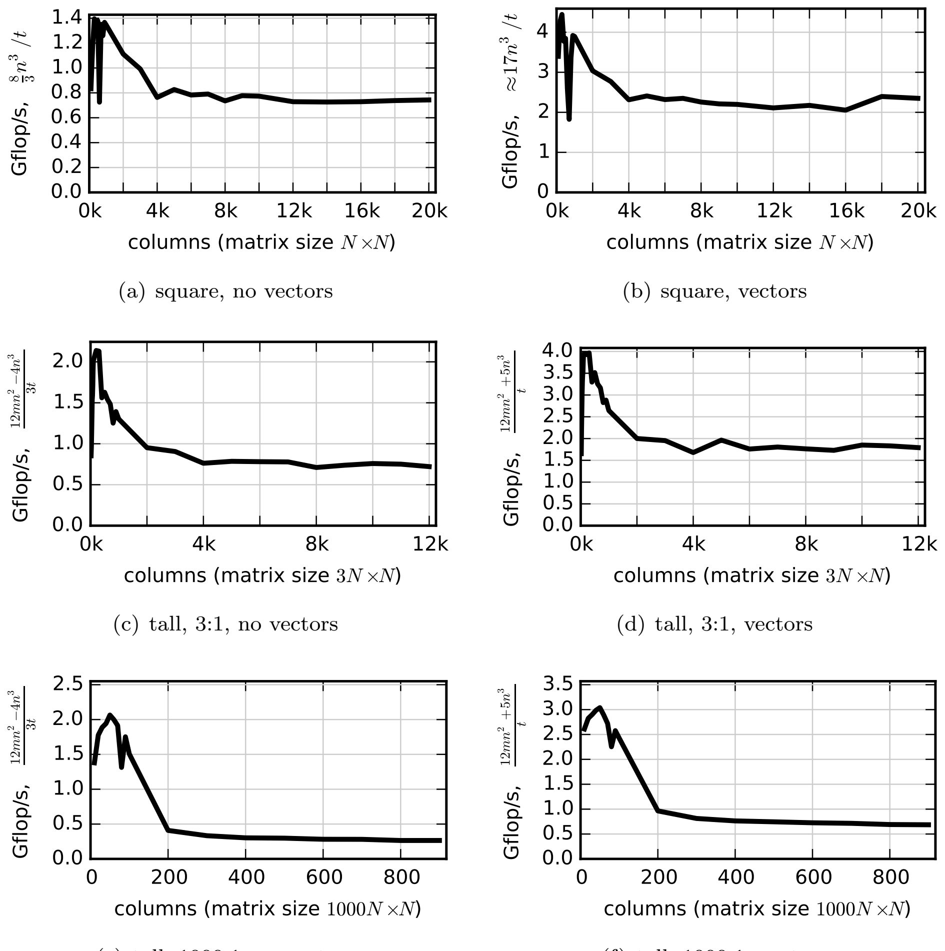
To arrange multidimensional arrays in linear storage such as memory, Algol uses row-major order (each row is contiguous in memory), while Fortran uses column-major order (each column is contiguous in memory). Array layout is critical for correctly passing arrays between programs written in different languages. It is also important for performance when traversing an array since accessing array elements that are contiguous in memory is usually much faster than accessing elements which are not, due to the structure of the memory cache hierarchy. In the Algol routines, and subsequently the Fortran routines of EISPACK, matrix elements were referenced by row, thus causing great inefficiencies in the Fortran EISPACK software on modern cache based computer systems.

Written in standard Fortran 77, with no outside dependencies, EISPACK still compiles with a modern Fortran compiler. Figure 1 shows its performance results on a modern computer with the six test problems described in section 2. EISPACK has no notion of parallelism, so the code runs on only a single core. The operation count formulas here assume two QR iterations per singular value, and that an initial QR reduction is not done [23].

For square matrices without computing singular vectors, asymptotic performance is limited to 0.74 Gflop/s for one core, while when computing singular vectors, performance nearly triples to 2.17 Gflop/s. As is common, small sizes perform better because the entire matrix fits into L2 cache. Performance for the tall 3:1 and 1000:1 cases is less than the square case, but exhibits a similar improvement when computing singular vectors compared with no vectors. For comparison, the practical peak using matrix-multiply on one core is 20 Gflop/s.

4. LINPACK Implementation Using BLAS.

In the 1970s, the Level 1 BLAS (Basic Linear Algebra Subroutines) [79] were introduced as a standard set of interfaces to perform common linear algebra operations. The Level 1 BLAS includes operations with $O(n)$ floating-point operations (flops), such as vector sum ($y = \alpha x + y$, called `daxpy`). The LINPACK project [39] reimplemented the SVD algorithm, along with other linear algebra algorithms, using Level 1 BLAS for efficient execution on the vector supercomputers of the 1970s and 1980s. It uses Fortran's native column-major order, which makes better use of cache and memory bandwidth. However,



(a) square, no vectors

(b) square, vectors

(c) tall, 3:1, no vectors

(d) tall, 3:1, vectors

(e) tall, 1000:1, no vectors

(f) tall, 1000:1, vectors

FIG. 1. *Results for EISPACK, which uses only one core.*

using Level 1 BLAS, LINPACK is limited by the memory bandwidth and receives little benefit from multiple cores. We see in Figure 2 that LINPACK achieves up to 3.9× speedup over EISPACK for the square, no vectors case, and 2.7× speedup for the square, vectors case. When computing a tall $m \times n$ matrix with $m = 1000n$, using multithreaded BLAS on 16 cores yields some benefit, with speedups of 22.5× and 13.5× over EISPACK for the no vectors and vectors cases, respectively, compared with speedups of 7.6× and 3.9×, respectively, with single-threaded BLAS. In some instances, for large matrices such as $n = 16,000$, the code hung, appearing in a "sleep" state in `ps`, so we were unable to collect all data points.

5. LAPACK Implementation Based on Blocked Householder Transformations.

While successful for vector-processing machines, Level 1 BLAS were not a good fit for the cache-based machines that emerged later in the 1980s. For cache-based machines, it is preferable to use higher-level operations such as matrix-matrix multiply, which is implemented by splitting a matrix into small blocks that fit into cache memory and performing small matrix-matrix multiplies on these blocks. This

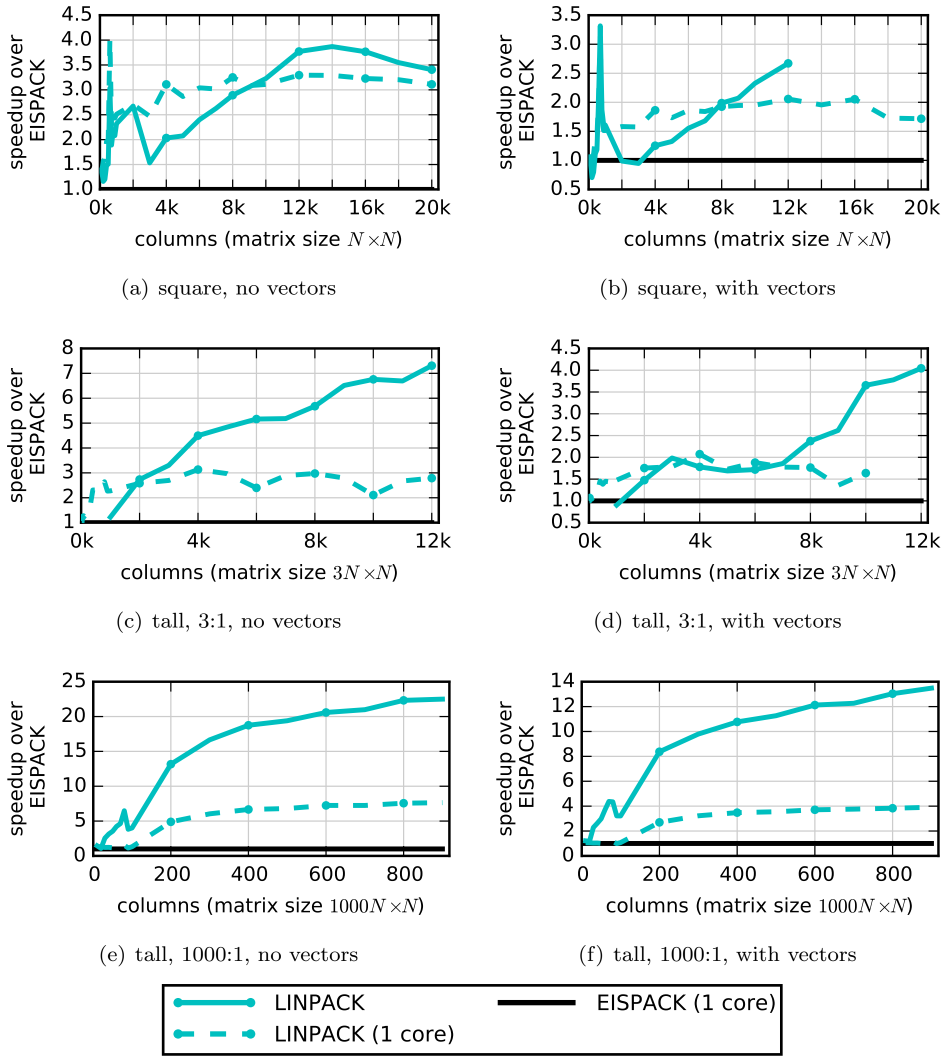


FIG. 2. *Comparison of LINPACK to EISPACK.*

avoids excessive data movement between cache and main memory. This led to the Level 2 BLAS [41] for operations with $O(n^2)$ flops, such as general matrix-vector multiply ($y = \alpha Ax + \beta y$, called `dgemv`); and Level 3 BLAS [40] for operations with $O(n^3)$ flops on $O(n^2)$ data, such as general matrix-matrix multiply ($C = \alpha AB + \beta C$, called `dgemm`). Level 1 and 2 BLAS access $O(1)$ elements per operation, and are thus limited in performance by the memory bandwidth. Level 3 BLAS benefit from the *surface-to-volume* effect of having only $O(n^2)$ elements to access for $O(n^3)$ operations. The performance of Level 1, 2, and 3 BLAS are compared in Figure 3, showing the significant benefit of Level 3 BLAS. The BLAS provides a means to write high-level, high-performance, portable numerical software. Optimized BLAS libraries are available, both from commercial vendors such as the Intel Math Kernel Library (MKL) [71] and the IBM Engineering and Scientific Subroutine Library (ESSL) [70],

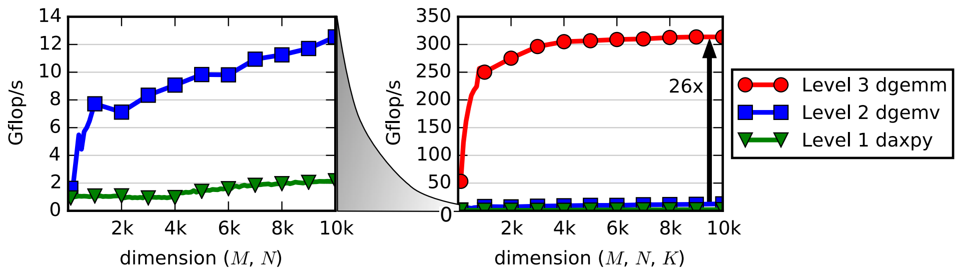
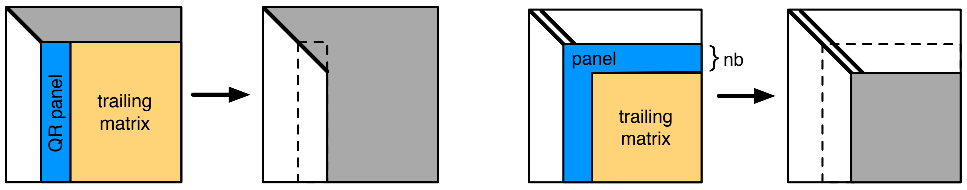


FIG. 3. *Comparison of Level 1, 2, and 3 BLAS performance.*

and in open-source libraries such as OpenBLAS [96] and ATLAS [115]. These math libraries often also included optimized versions of LAPACK, ScaLAPACK, and other numerical libraries. Our tests used the optimized routines available in Intel MKL.

5.1. Blocked Householder Transformations. With the introduction of Level 3 BLAS, algorithms were recast using matrix multiplies, and LINPACK was redesigned into LAPACK [2] to use Level 3 BLAS where possible. The redesign for one-sided factorizations such as QR, LU, and Cholesky is relatively easier than reductions for eigenvalue problems and the SVD because the transformations used in QR, LU, and Cholesky are applied from only the left side [40]. Consecutive elementary transformations are restricted to a block of columns at a time, referred to as the panel (depicted in Figure 4(a)), and updates to the rest of the matrix, referred to as the trailing matrix, are delayed. The transformations used for a panel are blocked together [14, 104] and applied to the trailing matrix as Level 3 BLAS.



(a) QR factorization (b) Bidiagonal reduction

FIG. 4. *Comparison of panels and trailing matrix.*

On the other hand, the reduction of a matrix A to bidiagonal form is done by applying orthogonal matrices on both the left and right side of A—hence it is called a "two-sided factorization." The two-sided transformations create more data dependencies, which make it impossible to entirely remove matrix-vector products involving the trailing matrix (as in the one-sided factorizations). The panel becomes a block row and block column, as shown in Figure 4(b), but panel operations also involve the entire trailing matrix. Dongarra et al. [42] developed the blocked algorithm for the bidiagonal reduction. The algorithm as implemented in LAPACK is given in Algorithm 1, and can be summarized as follows.

Two orthogonal matrices, U_1 and V_1, are applied on the left and right side, respectively, of an $m \times n$ matrix A to reduce it to bidiagonal form, $B = U_1^T A V_1$. The

Algorithm 1 LAPACK implementation of bidiagonal reduction. In the $\{\cdot\}_{\ldots}$ notation, only the indicated column or row should be computed, not the entire matrix product. y_i and x_i are computed as a series of matrix-vector products by distributing v_i and u_i. In LAPACK, Householder vectors representing V and U overwrite A. Auxiliary function householder(x) (`dlarfg` in LAPACK) returns τ and v that define a Householder reflector H_i, and the updated vector $\hat{x} = H_i x = [\,\pm\|x\|, 0, \ldots, 0\,]^T$.

```
// bidiagonal reduction ($A$ is $m \times n$; assumes $m \geq n$ and $n$ divisible by $n_b$)
function gebrd( $A$ )
    for $i = 1 : n$ by $n_b$
        $(V;\ Y;\ X;\ U) = \text{labrd}(\ A_{i:m,\, i:n}\ )$
        $A_{i+n_b:m,\, i+n_b:n} = A_{i+n_b:m,\, i+n_b:n} - VY^T - XU^T$
    end
end function

// panel of bidiagonal reduction ($A$ is $m \times n$; assumes $m \geq n$)
function labrd( $A$ )
    $V$, $Y$, $X$, $U$ initially empty
    for $i = 1 : n_b$
        // compute column $i$ of $A_{(i-1)}$ using (4),
        // then compute $H_i$ to eliminate below diagonal
        $A_{i:m,\, i} = \left\{A - V_{i-1}Y_{i-1}^T - X_{i-1}U_{i-1}^T\right\}_{i:m,\, i}$
        $(\tau_i;\ v_i;\ A_{i:m,\, i}) = \text{householder}(\ A_{i:m,\, i}\ )$
        $y_i = \tau_i A_{(i-1)}^T v_i = \tau_i (A - V_{i-1}Y_{i-1}^T - X_{i-1}U_{i-1}^T)^T v_i$

        // compute row $i$ of $H_i A_{(i-1)}$ using (2) and (4),
        // then compute $G_i$ to eliminate right of super-diagonal
        $A_{i,\, i+1:n} = \left\{A - V_i Y_i^T - X_{i-1}U_{i-1}^T\right\}_{i,\, i+1:n}$
        $(\pi_i;\ u_i;\ A_{i,\, i+1:n}) = \text{householder}(\ A_{i,\, i+1:n}\ )$
        $x_i = \pi_i (A_{(i-1)} - v_i y_i^T) u_i = \pi_i (A - V_i Y_i^T - X_{i-1}U_{i-1}^T) u_i$
    end
    return $(V_{n_b+1:m,1:n_b}; Y_{n_b+1:n,1:n_b}; X_{n_b+1:m,1:n_b}; U_{n_b+1:n,1:n_b})$
end function
```

matrices U_1 and V_1 are represented as products of elementary Householder reflectors:

$$U_1 = H_1 H_2 \ldots H_n \qquad \text{and} \qquad V_1 = G_1 G_2 \ldots G_{n-1}.$$

Each H_i and G_i has the form

$$H_i = I - \tau_i v_i v_i^T \qquad \text{and} \qquad G_i = I - \pi_i u_i u_i^T,$$

where τ_i and π_i are scalars, and v_i and u_i are vectors. H_i eliminates elements below the diagonal in column i, while G_i eliminates elements right of the superdiagonal in row i. Let $A_{(i-1)}$ be the reduced matrix A after step $i-1$. Applying H_i on the left yields

$$H_i A_{(i-1)} = (I - \tau_i v_i v_i^T) A_{(i-1)} = A_{(i-1)} - v_i y_i^T, \tag{2}$$

while applying both H_i and G_i yields

$$
\begin{aligned}
A_{(i)} = H_i A_{(i-1)} G_i &= (I - \tau_i v_i v_i^T) A_{(i-1)} (I - \pi_i u_i u_i^T) \\
&= A_{(i-1)} - v_i y_i^T - x_i u_i^T,
\end{aligned}
\tag{3}
$$

where $y_i = \tau_i A_{(i-1)}^T v_i$ and $x_i = \pi_i (A_{(i-1)} - v_i y_i^T) u_i$. Blocking together i applications of (3), we obtain

$$
A_i = H_i \cdots H_1 A G_1 \cdots G_i = A - V_i Y_i^T - X_i U_i^T,
\tag{4}
$$

where $U_i = \big[u_1, \dots, u_i \big]$, and similarly with V_i, X_i, and Y_i. Note that it is possible to update just part of A, namely the i-th column and row of A, in order to proceed with the computation of the H_i and G_i. Thus, a delayed update is possible, but at each step we still compute two matrix-vector products involving the entire trailing matrix of A. As a result, if $m = n$, the entire factorization takes approximately $\frac{8}{3}n^3$ flops, with half of the operations in Level 2 BLAS (matrix-vector products), while the other half are in Level 3 BLAS.

5.2. QR Iteration. After the bidiagonal reduction, LAPACK solves the bidiagonal SVD using QR iteration, similar to EISPACK and LINPACK, or using divide and conquer, which is described later in section 7. The original QR iteration algorithm computed singular values to high absolute accuracy, meaning small singular values might be inaccurate. Demmel and Kahan [31] derived the implicit zero-shift QR iteration algorithm and proved that it computes all singular values to high relative accuracy; this is used as needed for accuracy by LAPACK when computing singular vectors. Accuracy is discussed further in section 13.

The `qd` (German: quotienten-differenzen) [100] and *differential* `qd` (`dqd`) [101] algorithms proposed by Rutishauser actually predate QR iteration and are among the first algorithms for computing singular values for modern computers. Subsequent to Demmel and Kahan's work, Fernando and Parlett [48] derived a shifted version called `dqds` that allowed using shifts to maintain fast convergence, while still maintaining high relative accuracy. This is used by LAPACK when computing singular values only (no vectors). Quite a few more variants of `qd` can be derived [97].

5.3. Computation of Singular Vectors. Normally, LAPACK stores orthogonal matrices in an implicit fashion as a sequence of Householder reflectors, each represented by a scalar τ_i and vector u_i. For QR iteration to accumulate the singular vectors, it first generates U_1 and V_1 explicitly (using `dorgbr`); this is essentially applying block Householder reflectors to an identity matrix, as a series of Level 3 BLAS operations.

The QR iteration algorithm then updates U_1 and V_1 by applying the Givens rotations used to reduce the bidiagonal matrix to diagonal. This is implemented in a Level 2 BLAS-like fashion, where an entire sequence of n Givens rotations is applied to update the entire U and V matrices (using `dlasr`). Recently, Van Zee et al. [113] developed a Level 3 BLAS-like implementation of applying Givens rotations, which they found made the SVD using QR iteration competitive with the SVD using divide and conquer (discussed in section 7).

5.4. Initial QR Factorization. If $m \gg n$, it is more efficient to first perform a QR factorization of A, and then compute the SVD of the n-by-n matrix R, since if $A = QR$ and $R = U \Sigma V^T$, then the SVD of A is given by $A = (QU)\Sigma V^T$. Similarly, if $m \ll$

n, it is more efficient to first perform an LQ factorization of A. Chan [23] analyzed this optimization, showing that it reduces the number of floating point operations. The operation counts are given in Figure 5, with the theoretical crossover points based on flops. Figure 6 plots the operation count as the ratio $m:n$ increases, illustrating the large savings as a matrix becomes taller. The results for tall matrices in Figures 8(c) to 8(f) show that LAPACK achieves significant speedups, such as 120× compared with EISPACK. This is a result of the reduced operation count and that much of the computation is done via Level 3 BLAS in QR factorization, followed by a relatively small square SVD problem.

	no vectors	QR iteration, with vectors	D&C, with vectors
Unoptimized	$4mn^2 - \frac{4}{3}n^3$	$12mn^2 + \frac{16}{3}n^3$	$8mn^2 + \frac{4}{3}n^3$
With initial QR	$2mn^2 + 2n^3$	$6mn^2 + 16n^3$	$6mn^2 + 8n^3$
Theoretical crossover	$m \geq \frac{5}{3}n$	$m \geq \frac{16}{9}n$	$m \geq \frac{10}{3}n$

FIG. 5. *Floating point operation counts.*

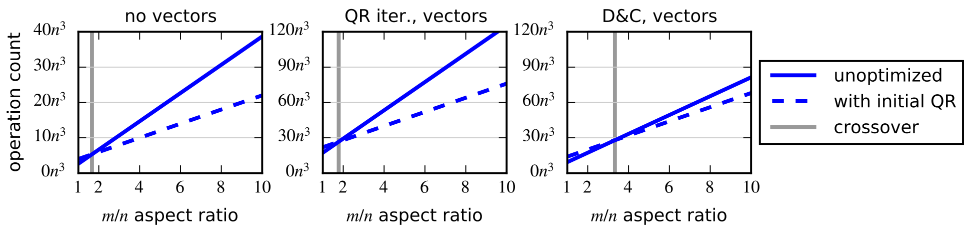


FIG. 6. *Operation counts as ratio $m:n$ increases (i.e., matrix gets taller), showing crossover where doing initial QR factorization is beneficial.*

Since bidiagonal divide and conquer (D&C, discussed in section 7) always operates on a square matrix, doing an initial QR factorization with D&C results in less of an improvement than with QR iteration. Asymptotically, as the ratio $m:n \to \inf$, the initial QR factorization, generating Q, and multiplying by Q are responsible for most of the cost, as shown by the profile of the 1000:1 case in Figure 7. As a result, using QR iteration or divide and conquer yield the same performance for very tall-skinny matrices. The crossover points in Figure 5 are based solely on flop counts. Since doing an initial QR also shifts operations from Level 2 to Level 3 BLAS, the crossovers are ideally tunable parameters, for instance by LAPACK's `ilaenv` tuning function.

5.5. Results. An overview of all the phases in the complete SVD is given in Algorithm 2, with a profile of the time spent in each phase in Figure 7. For the square, no vectors case, we see that the bidiagonal reduction (blue with \\ hatching) takes almost the entire time, while QR iteration (green, no hatching) takes very little time, as expected since QR iteration is $O(n^2)$ while the bidiagonal reduction costs $\frac{8}{3}n^3$ flops. When computing singular vectors, the QR iteration time becomes nearly half of the overall time, due to accumulating $U = U_1U_2$ and $V = V_1V_2$. Generating the explicit U_1 and V_1 matrices (orange with \\\\ hatching) is a small portion of the time, even though together they have nominally the same operation count ($\frac{8}{3}n^3$) as

Algorithm 2 Overview of SVD algorithm using QR iteration (`dgesvd`) for $m \geq n$. Accumulating $U = U_1 U_2$ and $V = V_1 V_2$ occurs during QR iteration. † Marked lines are required only when computing singular vectors.

Description	LAPACK Routine	Cost	Cost for $m \gg n$
if $m \gg n$ **then**			
$QR = A$ (QR factorization)	`dgeqrf`		$2mn^2 - \frac{2}{3}n^3$
$\hat{A} = R$			
else			
$\hat{A} = A$			
end			
$U_1 B V_1^T = \hat{A}$ (bidiagonalization)	`dgebrd`	$4mn^2 - \frac{4}{3}n^3$	$\frac{8}{3}n^3$
generate explicit U_1	`dorgbr` †	$2mn^2 - \frac{2}{3}n^3$	$\frac{4}{3}n^3$
generate explicit V_1	`dorgbr` †	$\frac{4}{3}n^3$	$\frac{4}{3}n^3$
$U_2 \Sigma V_2^T = B$ (QR iteration)	`dbdsqr`	$O(n^2)$	$O(n^2)$
$U = U_1 U_2$	" " †	$6mn^2$	$6n^3$
$V = V_1 V_2$	" " †	$6n^3$	$6n^3$
if $m \gg n$ **then**			
generate explicit Q	`dorgqr` †		$2mn^2 - \frac{2}{3}n^3$
$U = Q\hat{U}$	`dgemm` †		$2mn^2$
end			
Total cost (with vectors †)		$12mn^2 + \frac{16}{3}n^3$	$6mn^2 + 16n^3$
Total cost (no vectors)		$4mn^2 - \frac{4}{3}n^3$	$2mn^2 + 2n^3$

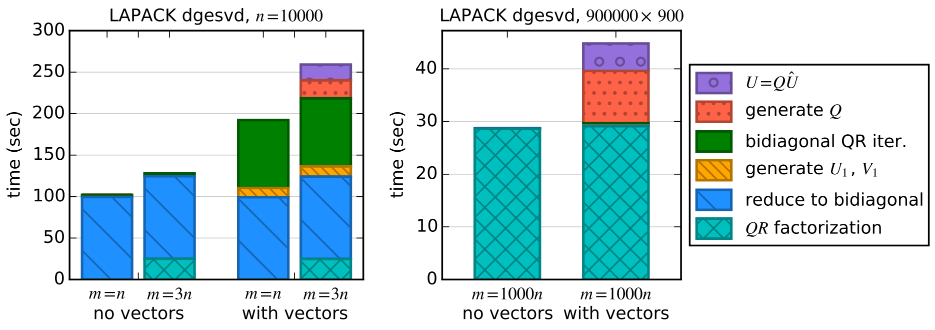


FIG. 7. *Profile of LAPACK SVD. Left is* 10000×10000 *and* 30000×10000 *problem. QR factorization reduces the* 30000×10000 *matrix to a* 10000×10000 *matrix. Right is a* 900000×900 *problem, where reduction to bidiagonal and QR iteration become vanishingly small.*

the bidiagonal reduction. This exemplifies the performance difference between Level 2 BLAS, in the bidiagonal reduction, and Level 3 BLAS, in generating U_1 and V_1.

The tall 3:1 matrix first does an initial QR factorization, resulting in a square R matrix the same size as the square case (10000×10000). Thus the profile for the 3:1 case simply adds the QR factorization, generating Q, and multiplying $U = Q\hat{U}$ steps to the square case. For the tall 1000:1 matrix, the initial QR factorization dominates the overall time, with the subsequent bidiagonal reduction and QR iteration becoming vanishingly small. When vectors are computed, generating Q and multiplying $U = Q\hat{U}$ add significant time, while generating U_1 and V_1 and updating $U = U_1 U_2$ and

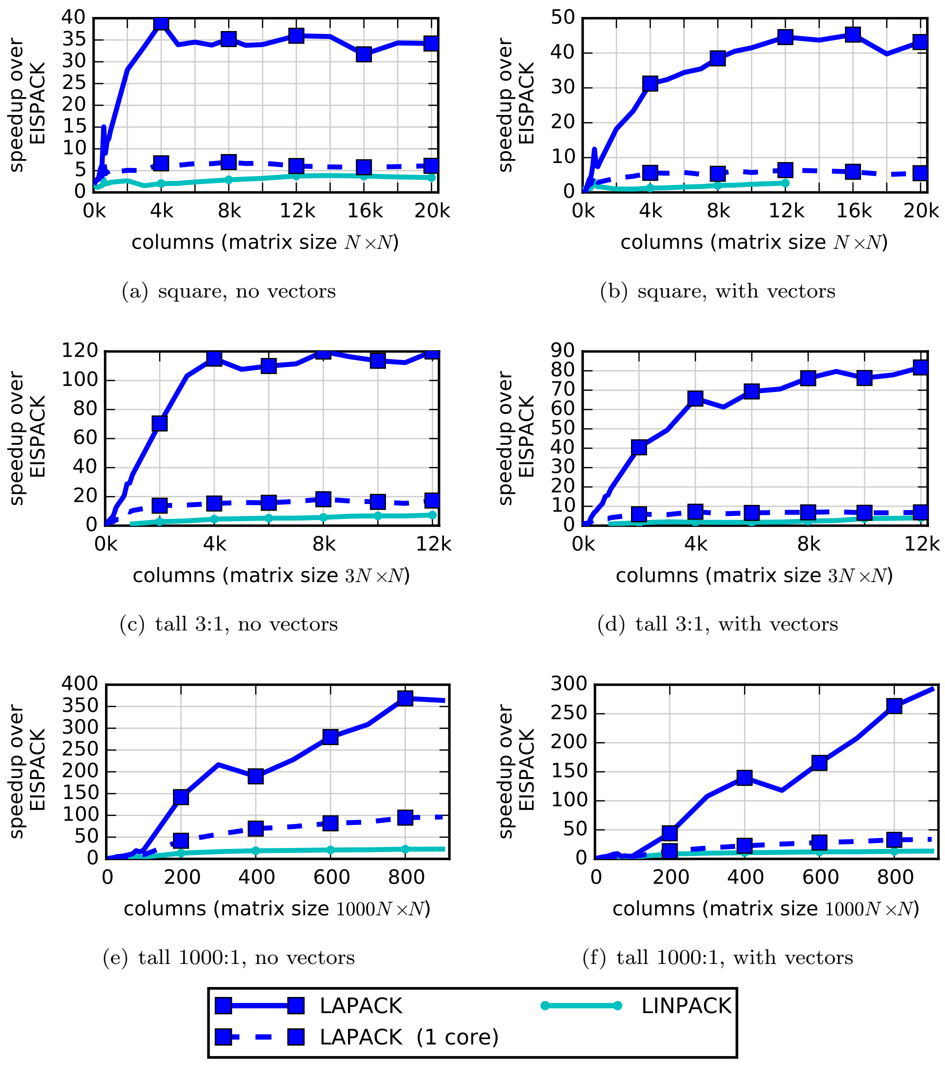


FIG. 8. *Comparison of LAPACK, LINPACK, and EISPACK. Solid lines represent 16-core runs; dashed lines represent single core runs.*

$V = V_1 V_2$ during QR iteration are also vanishingly small. Thus for very tall-skinny matrices, the performance is dominated by operations rich in Level 3 BLAS.

Figure 8 shows the speedup that LAPACK achieves compared with EISPACK. Even a single-core implementation may achieve over 5× speedup. But the real potential is shown when using multiple cores (16 in this case)—a 45× speedup is possible for square matrices and over 350× speedup for tall matrices with a 1000:1 row-to-column ratio. The square, no vectors case in Figure 8(a) is dominated by the bidiagonalization, as the subsequent bidiagonal SVD is $O(n^2)$. With Level 3 BLAS being significantly faster than Level 2 BLAS, and half the operations in Level 2 BLAS, we expect the bidiagonalization to be about 2× the speed of Level 2 BLAS. Modeling the time as

$$t = \frac{4n^3}{3r_2} + \frac{4n^3}{3r_3}$$

with the Level 2 BLAS rate as $r_2 = 13.4$ Gflop/s and the Level 3 BLAS rate as $r_3 = 315$ Gflop/s (from Figure 3), we obtain a theoretical bound of 25.7 Gflop/s. This yields a speedup of 34.7× over EISPACK—exactly what we see for LAPACK in Figure 8(a). When computing singular vectors, LAPACK achieves a higher speedup, up to 45.3×, reflecting that computation of U_1 and V_1 uses Level 3 BLAS. The tall matrix cases achieve even higher speedups because much of the work is done in the initial QR factorization.

5.6. Level 2.5 BLAS Implementation. Since the bidiagonalization performance is limited by the Level 2 BLAS operations, Howell et al. [69] sought to optimize these operations by observing that several Level 2 operations can be done together, thus reducing memory transfers by keeping data in cache. This technique of fusing several Level 2 operations together was called the Level 2.5 BLAS [69, 17]. For instance, to compute

$$\begin{aligned} x &= \beta A^T y + z, \\ w &= \alpha A x, \end{aligned}$$

known as `dgemvt`, A is partitioned into block columns as

$$A = \begin{bmatrix} A_1 \ A_2 \ \cdots \ A_k \end{bmatrix},$$

where each A_i has b columns, and is sized such that it fits into cache. Correspondingly, x and z are partitioned as

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix}, \qquad z = \begin{bmatrix} z_1 \\ \vdots \\ z_k \end{bmatrix}.$$

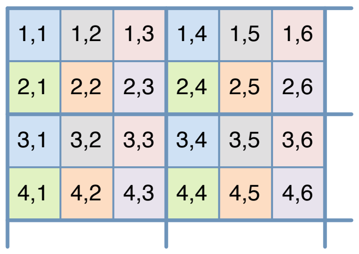
The `dgemvt` loops over the A_i blocks, performing two `dgemv` operations with each block as shown in Algorithm 3. Keeping each A_i in cache for the second `dgemv` cuts main memory traffic roughly in half, thereby increasing the potential performance. With some algebraic manipulation, the two products $y_i = \tau_i A^T v_i$ and $x_i = \pi_i A u_i$ from the bidiagonalization panel can be computed together using this `dgemvt`. Tests that Howell et al. ran showed a 1.2–1.3× speedup over the existing LAPACK implementation for the bidiagonalization. Van Zee et al. [114] further analyzed these operations and fused them at the register level, reusing data in registers to also avoid unnecessary accesses to cache memory, showing potential further speedups. So far, these results have been for single-threaded implementations, and the speedups do not carry over when using multithreaded BLAS. If optimized Level 2.5 BLAS become available for multicore processors, this may become a viable approach, but we don't pursue this further here.

6. ScaLAPACK Implementation.

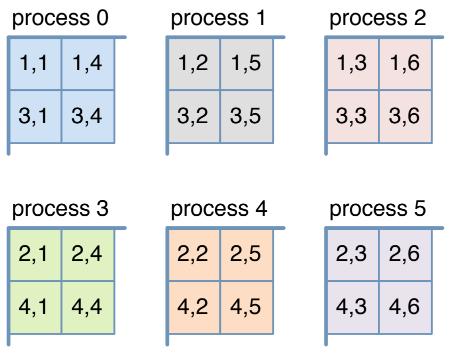
To use a distributed-memory computer, the Scalable Linear Algebra Package (ScaLAPACK) [16] extends LAPACK by distributing the matrices in a 2D block cyclic layout using the prescribed block size n_b and the pair of parameters (p, q) to define a p-by-q process grid, as illustrated in Figure 9. ScaLAPACK parallelizes the LAPACK subroutines using the parallel version of

Algorithm 3 Pseudocode for `dgemvt`

```
$w = 0$
for $i = 1 : k$
    $x_i = \beta A_i^T x_i + z_i$   // dgemv, loads $A_i$ into cache
    $w = \alpha A_i x_i + w$       // dgemv, reuses $A_i$ in cache
end
```



(a) Global view of matrix. Each square is an $n_b \times n_b$ block, colored by process that it is distributed to in (b).

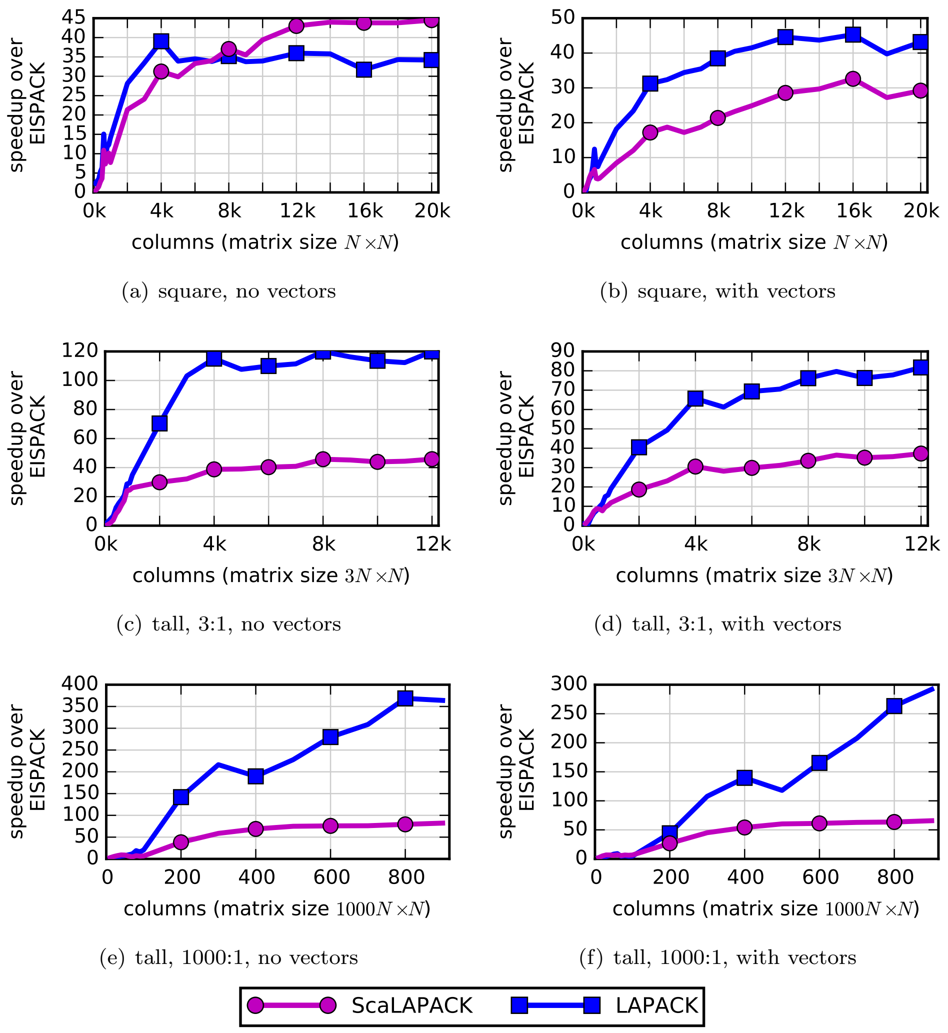


(b) Local view of matrix. Each local submatrix is stored in column-major order.

FIG. 9. *2D block cyclic distribution of the matrix A using 2-by-3 processor grid.*

BLAS (PBLAS) and the Basic Linear Algebra Communication Subprograms (BLACS) for the interprocess communication, implemented on top of the Message Passing Interface (MPI) [93]. For instance, to bidiagonalize the input matrix for computing the SVD [24], `dgebrd` of LAPACK uses the Level 2 BLAS matrix-vector multiply (`dgemv`) to perform about half of its total flops. Now, to perform the matrix-vector multiply on a distributed-memory computer, in `pdgemv` of PBLAS, each process first gathers all the required nonlocal block rows of the input vector from other processes. After the completion of this initial interprocess communication, each process independently computes the matrix-vector multiplication with the local submatrix. Finally, each process computes the local part of the output vector by gathering and accumulating the partial results from the other processes in the same row of the process grid. Hence, ScaLAPACK follows the fork-join parallel programming paradigm and is designed for the weak parallel scalability of the algorithm. Since PBLAS performs most of its local computation using BLAS, ScaLAPACK can exploit a NUMA (non-uniform memory access) architecture using a threaded version of BLAS.

Figure 10 compares the performance of ScaLAPACK's `pdgesvd` with the performance of LAPACK's threaded `dgesvd` for computing the SVD on our 16-core shared-memory computer. While, from ScaLAPACK's perspective, each MPI process has its own memory and explicit messages are passed between MPI processes, on a shared-memory computer the MPI implementation uses an efficient shared-memory communication layer to copy data. See section 11 for ScaLAPACK's performance on a distributed memory computer. The performance of `pdgesvd` was obtained using the tester included in ScaLAPACK version 2.0.2, which was linked with ScaLAPACK and the sequential LAPACK/BLAS of Intel MKL. We tested the performance of `pdgesvd`



(a) square, no vectors

(b) square, with vectors

(c) tall, 3:1, no vectors

(d) tall, 3:1, with vectors

(e) tall, 1000:1, no vectors

(f) tall, 1000:1, with vectors

Fig. 10. *Comparison of ScaLAPACK to LAPACK.*

using 8-by-2, 4-by-4, and 2-by-8 processor grids and block sizes of 32, 64, and 128. The figure shows the optimal performance among these parameter configurations. We see that the performance of `pdgesvd` was often lower than the performance of LAPACK's `dgesvd`. This is mainly because several optimizations have not been implemented in `pdgesvd`. For instance, for a tall-skinny matrix ($m \gg n$), `dgesvd` computes the QR factorization of the input matrix A, followed by SVD of the resulting upper-triangular matrix, as described in subsection 5.4. For a tall-skinny matrix A, this greatly reduces the number of required floating point operations, compared to that of `pdgesvd`, which directly computes the SVD of the input matrix. As a result, for computing the SVD of a tall-skinny matrix, `pdgesvd` was slower than `dgesvd`.

After the bidiagonalization of the input matrix A, `pdgesvd` computes the SVD of the bidiagonal matrix using `dbdsqr` of LAPACK. If only the singular values are requested, `pdgesvd` typically spends an insignificant amount of time in `dbdsqr`. How-

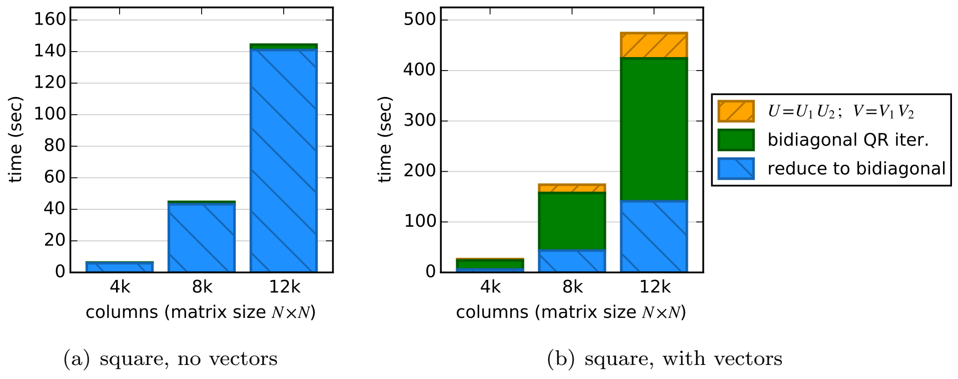


FIG. 11. *Profile of ScaLAPACK reference implementation with* $n_b = 32$ *and* $(p, q) = (4, 4)$.

ever, if the singular vectors are needed, our performance profile in Figure 11 using the reference implementation of `pdgesvd` revealed that the execution time can be dominated by the time to compute the singular vectors of the bidiagonal matrix. The reason is that `pdgesvd` has all the MPI processes in the same column or row of the processor grid redundantly compute the left or right singular vectors, respectively, of the bidiagonal matrix, that are distributed to the process group. Compared with `pdgesvd`, LAPACK's `dgesvd` obtained higher performance by using `dbdsqr` with multithreaded BLAS. The reference implementation of `pdgesvd` obtained about the same performance as that of MKL's `pdgesvd` when linked to MKL BLAS and LAPACK.

Finally, ScaLAPACK supports only the QR iteration algorithm for computing the SVD of the bidiagonal matrix, using LAPACK's `dbdsqr`, while as shown in section 7, the divide and conquer process in LAPACK's `dbdsdc` may be faster than `dbdsqr`.

7. Singular Vectors from the Divide and Conquer Process.

For solving the bidiagonal SVD subproblem, QR iteration and the related qd algorithms may take as much as 80% of the total time when computing singular vectors of a dense matrix [56]. Gu and Eisenstat introduced the bidiagonal divide and conquer (D&C) [57, 59] algorithm, which may be an order of magnitude faster on some machines [56]. The development of D&C was based on prior work focusing on computing eigenvalues and singular values [4, 25, 52, 58, 74].

The divide and conquer process includes a matrix partitioning step that introduces two large submatrices. The splitting can either occur with "the middle" row [4, 56] or column [57]:

$$B = \begin{bmatrix} B_1 & 0 \\ \alpha_k e_k & \beta_k e_1 \\ 0 & B_2 \end{bmatrix} \qquad \text{or} \qquad B = \begin{bmatrix} B_1 & \alpha_k e_k & 0 \\ 0 & \beta_k e_1 & B_2 \end{bmatrix}.$$

Note that after the partitioning, B_1 might not be square, even though B was. The fix is to append a zero row or column [57] to obtain the desired shape.

In either row or column case, the process continues recursively to obtain the SVD of B_1 and B_2, which can be used to decompose B as:

$$B = Q_r M_r W_r \qquad \text{or} \qquad B = Q_c M_c W_c,$$

with orthogonal matrices Q_r, W_r, Q_c, and W_c. M_r and M_c have a special structure:

only the diagonal and either a single row or column is non-zero, respectively:

$$M_r = \begin{bmatrix} z_1 & z_2 & \dots & z_n \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{bmatrix} \quad \text{or} \quad M_c = \begin{bmatrix} z_1 & & & \\ z_2 & d_2 & & \\ \vdots & & \ddots & \\ z_n & & & d_n \end{bmatrix}.$$

Trivially, because matrices Q_r, W_r, Q_c, and W_c are orthogonal, B and M_r or M_c share singular values σ_i. A number of theorems and lemmas [74, 57] lead to a fast and numerically (relatively) stable procedure for computing the SVD of M_r or M_c as $U_m \Sigma_m V_m^T$. The interlacing property sets the bounds and ranges for σ_i:

$$0 \equiv d_1 < \sigma_1 < d_2 < \ldots < d_n < \sigma_n < d_n + \|z\|_2$$

and the secular equation:

$$f(\sigma) = 1 + \sum_{k=1}^{n} \frac{z_k^2}{d_k^2 - \sigma^2} = 0$$

is used for computing the values σ_i with a specifically crafted root finder that accounts for floating-point vagaries of past and modern computing systems [80]. The corresponding formulas for the left singular vectors U_m:

$$u_i = \left[\frac{z_1}{d_1^2 - \sigma_i^2}, \ldots, \frac{z_n}{d_n^2 - \sigma_i^2} \right]^T \Big/ \sqrt{\sum_{k=1}^{n} \frac{z_k^2}{(d_k^2 - \sigma_i^2)^2}}, \tag{5}$$

and the right singular vectors V_m:

$$v_i = \left[-1, \frac{d_2 z_2}{d_2^2 - \sigma_i^2}, \ldots, \frac{d_n z_n}{d_n^2 - \sigma_i^2} \right]^T \Big/ \sqrt{1 + \sum_{k=2}^{n} \frac{(d_k z_k)^2}{(d_k^2 - \sigma_i^2)^2}} \tag{6}$$

indicate that there could be accuracy problems for the components of either set of vectors, even though the computed singular values $\hat{\sigma}$ are a good approximation of the exact singular values σ, because the ratios $z_k/(d_k^2 - \sigma_i^2)$ in (5) and (6) can be inaccurate. The trick is not to use the same M_r or M_c matrices that were used to compute the approximate singular values $\hat{\sigma}_i$, but instead to construct new $\hat{M}_r$ or $\hat{M}_c$ based on $\hat{\sigma}_i$ that improve the accuracy of expressions in Equations (5) and (6) [80, 59]. This can dramatically diminish the departure from orthogonality for both sets of vectors.

After computing U_m and V_m, the SVD of B is computed by multiplying $Q_r U_m$ and $V_m^T W_r$. This is done for each B matrix in the recursion tree, from the leaf nodes to the root. Most of the cost of D&C is in these matrix multiplications, which are Level 3 BLAS. In particular, most of the cost is at the higher levels of the recursion tree, near the root node, as the matrices get larger.

Li et al. [81] recently showed that D&C internally generates matrices with structure that can be exploited. The matrices U_m and V_m, that are the singular vectors of M, have low-rank off-diagonal blocks that can be efficiently compressed with hierarchically semiseparable (HSS) matrices. Using HSS improves the speed of matrix multiplies, reducing the cost of the bidiagonal D&C step from $O(n^3)$ to $O(n^2 r)$, where r depends on the matrix but usually $r \ll n$ for large n. Li et al. showed over 3× improvement compared to Intel MKL for the bidiagonal D&C step on large matrices.

Algorithm 4 Overview of SVD algorithm using divide and conquer (`dgesdd`) for $m \geq n$. Generating explicit U_2 and V_2 occurs during D&C. † Marked lines are required only when computing singular vectors.

Description	LAPACK Routine	Cost	Cost for $m \gg n$
if $m \gg n$ **then**			
$QR = A$ (QR factorization)	`dgeqrf`		$2mn^2 - \frac{2}{3}n^3$
$\hat{A} = R$			
else			
$\hat{A} = A$			
end			
$U_1 B V_1^T = \hat{A}$ (bidiagonalization)	`dgebrd`	$4mn^2 - \frac{4}{3}n^3$	$\frac{8}{3}n^3$
$U_2 \Sigma V_2^T = B$ (D&C)	`dbdsdc`	$O(n^2)$	$O(n^2)$
generate explicit U_2	" " †	$\frac{4}{3}n^3$	$\frac{4}{3}n^3$
generate explicit V_2	" " †	$\frac{4}{3}n^3$	$\frac{4}{3}n^3$
$U = U_1 U_2$	`dormbr` †	$4mn^2 - 2n^3$	$2n^3$
$V = V_1 V_2$	`dormbr` †	$2n^3$	$2n^3$
if $m \gg n$ **then**			
generate explicit Q	`dorgqr` †		$2mn^2 - \frac{2}{3}n^3$
$U = QU$	`dgemm` †		$2mn^2$
end			
Total cost (with vectors †)		$8mn^2 + \frac{4}{3}n^3$	$6mn^2 + 8n^3$
Total cost (no vectors)		$4mn^2 - \frac{4}{3}n^3$	$2mn^2 + 2n^3$

D&C restructures the SVD algorithm somewhat, as shown in Algorithm 4, compared with the QR iteration version in Algorithm 2. D&C directly computes the SVD of the bidiagonal matrix $B = U_2 \Sigma V_2$, and then multiplies $U = U_1 U_2$ and $V = V_1 V_2$ afterwards (using `dormbr`), while with QR iteration, LAPACK first generates U_1 and V_1 (using `dorgbr`), then accumulates U_2 and V_2 onto U_1 and V_1 during QR iteration. The profile in Figure 12 shows this difference in the bidiagonal QR iteration (green, no hatching) vs. D&C steps (green, + hatching); and the generate U_1, V_1 (orange, \\\\ hatching) vs. $U = U_1 U_2$, $V = V_1 V_2$ (orange, // hatching) steps. The main advantage of the divide and conquer approach is that it saves nearly half the flops compared to QR iteration when computing singular vectors. For a square matrix, D&C is $\approx 9n^3$ flops, compared to $\approx 17n^3$ for QR iteration (Figure 5). We can observe this as a reduction in time for the steps mentioned above in Figure 12.

Figure 13 shows the relative speedup over EISPACK when using a modern multicore system, for both the divide and conquer (D&C) and QR iteration algorithms. We see that for square and tall 3:1 matrices, D&C is consistently faster than QR iteration. Because of the initial QR factorization (described in subsection 5.4) the advantage decreases as m grows relative to n, so that for a very tall matrix, both methods are nearly the same time, as seen by the 1000:1 case in Figure 13(c). It may be safely assumed that D&C is superior to the QR iteration algorithm for most scenarios, and the worst case is when both perform at the same speed. When computing only singular values, not singular vectors, LAPACK always uses QR iteration, since in that case both bidiagonal QR iteration and D&C are $O(n^2)$, while the overall time will be dominated by the $O(n^3)$ reduction to bidiagonal.

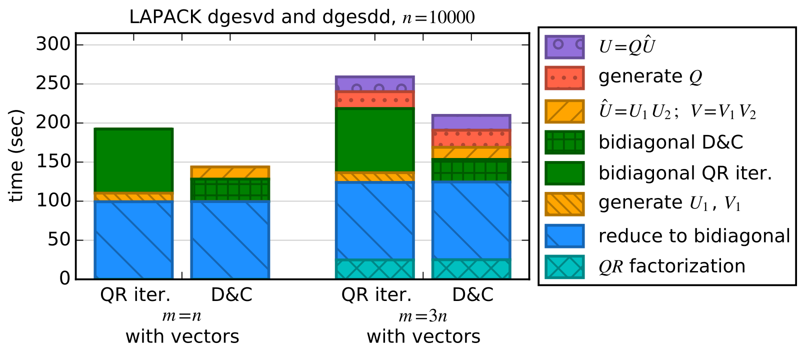


FIG. 12. *Profile comparing LAPACK QR iteration (*`dgesvd`*) and divide and conquer (*`dgesdd`*) algorithms. For QR iteration, it generates U_1 and V_1, then updates those with U_2 and V_2 during QR iteration. Divide and conquer generates U_2 and V_2, then multiplies $\hat{U} = U_1U_2$ and $V = V_1V_2$ afterwards.*

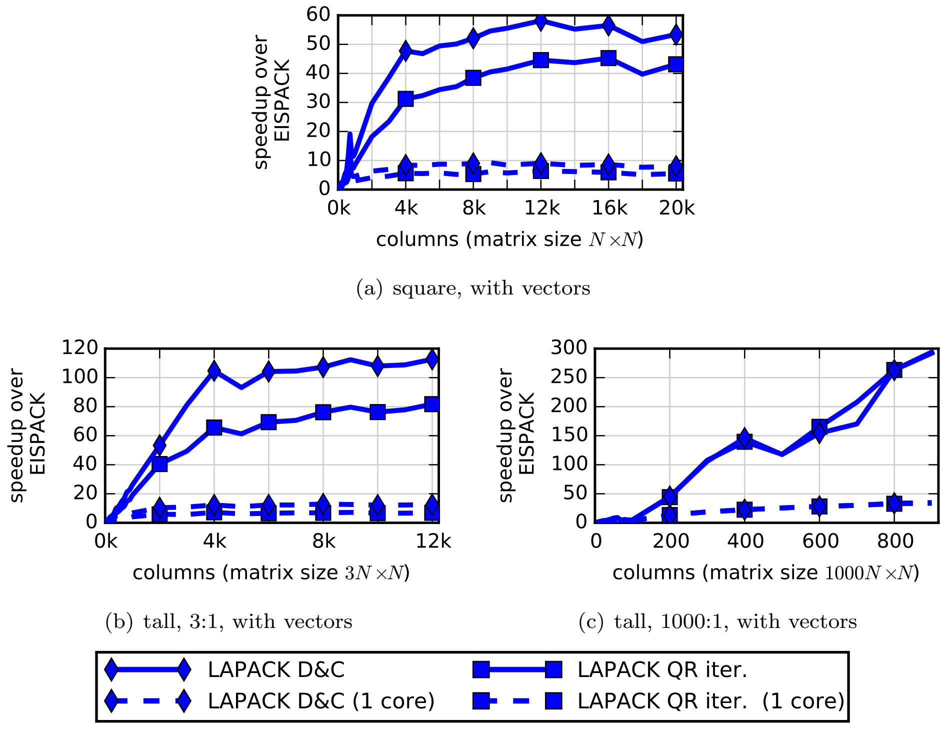


FIG. 13. *Comparison of LAPACK divide and conquer (D&C) to QR iteration. Solid lines represent 16-core runs; dashed lines represent single core runs.*

8. Bisection and Inverse Iteration.

LAPACK 3.6.0 introduced a bisection method (`dgesvdx`) to compute all or a subset of the singular values and vectors [86]. Similar to QR iteration (`dgesvd`) and divide and conquer (`dgesdd`), it first reduces the matrix A to bidiagonal form B. Then it computes the singular values of B based on bisection and the corresponding singular vectors by inverse iteration, using `dbdsvdx`. For computing the SVD of B, it converts the bidiagonal matrix B to the

Golub-Kahan [53] symmetric tridiagonal matrix T of dimension $2n$,

$$T = \text{tridiag}\begin{pmatrix} & b_{1,1} & b_{1,2} & b_{2,2} & b_{2,3} & \dots & b_{n-1,n} & b_{n,n} \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ & b_{1,1} & b_{1,2} & b_{2,2} & b_{2,3} & \dots & b_{n-1,n} & b_{n,n} \end{pmatrix}, \tag{7}$$

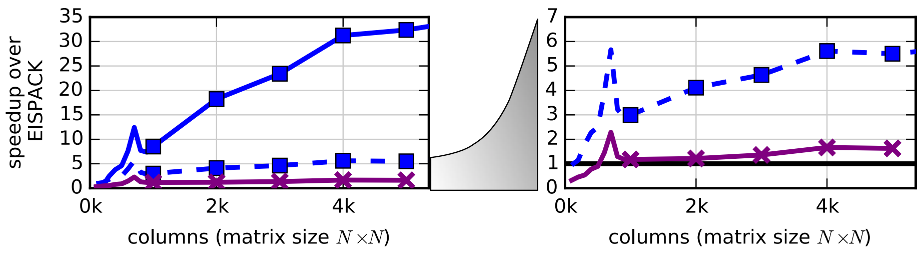
whose eigenpairs are $(\pm\sigma_j, z_j)$, where σ_j is the j-th singular value of B. Elements of u_j and v_j, the corresponding left and right singular vectors of B, are interleaved in the eigenvector as $z_j = [v_{1,j}, -u_{1,j}, v_{2,j}, -u_{2,j}, \dots, v_{n,j}, -u_{n,j}]/\sqrt{2}$. Instead of developing new subroutines, `dbdsvdx` relies on the subroutines `dstebz` and `dstein` that compute the eigenvalues and eigenvectors, respectively, of the symmetric tridiagonal matrix based on bisection and inverse iteration.

The bisection algorithm implemented in `dstebz` uses Sylvester's inertia theorem to compute the number of eigenvalues within a certain interval. In particular, the algorithm relies on the LDL^T factorization of the matrix T, where L is a lower-triangular matrix with unit diagonal and D is a diagonal matrix. For the symmetric tridiagonal matrix T, the diagonal matrix D can be computed with $O(n)$ flops based on the simple recurrence formula,

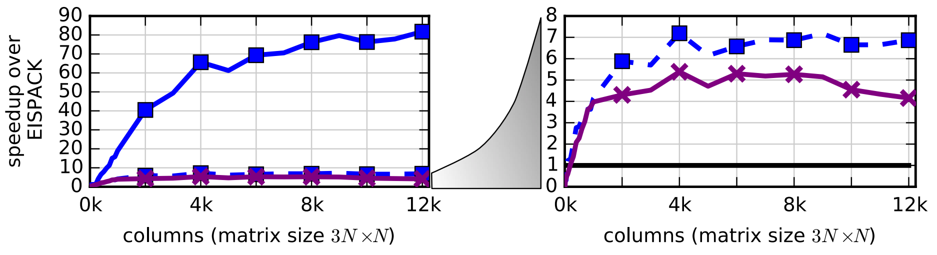
$$d_{i,i} = (t_{i,i} - s) - \frac{t_{i-1,i}^2}{d_{i-1,i-1}}.$$

Given the LDL^T factorization of the matrix $T - sI$ for a certain shift value s, the number of negative elements of D is equal to the number of eigenvalues of T smaller than s. In other words, given the LDL^T factorizations of two shifted matrices, $T - s_1 I$ and $T - s_2 I$, with $s_1 < s_2$, if there are n_1 and n_2 negative entries in their respective diagonal matrices, then there are $n_2 - n_1$ eigenvalues in the interval $(s_1, s_2]$. In addition, for the tridiagonal matrix, it can be shown that the LDL^T factorization without pivoting can be reliably used for counting the number of eigenvalues [34, 75]. Based on these observations, `dstebz` keeps bisecting the initial interval containing all the desired eigenvalues until it finds a small enough interval for each eigenvalue such that the computed eigenvalue has the desired accuracy. Each bisection improves the accuracy of the eigenvalue by one bit, hence the iteration converges linearly. An advantage of bisection is that it can be naturally adapted to compute a subset of eigenvalues, which was one of the motivations for introducing `dgesvdx` [86].

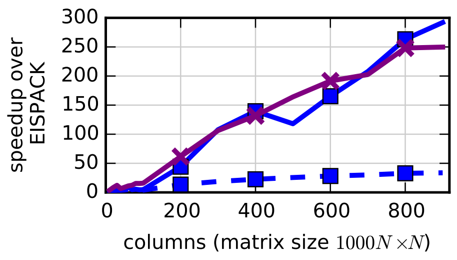
Given the eigenvalues computed by `dstebz`, `dstein` computes the corresponding eigenvectors based on inverse iteration. Namely, for each computed eigenvalue λ, it first computes the LU factorization of the shifted matrix $A - \lambda I$ with partial pivoting. Then the corresponding eigenvector of λ is computed by inverse iteration, with a starting vector whose entries are random numbers uniformly distributed in the interval $(-1, 1)$. Given an accurate eigenvalue approximation, inverse iteration converges quickly [72] (e.g., `dstein` sets the maximum number of iterations to be five). However, when the eigenvalues are close to each other, inverse iteration may fail to generate orthogonal eigenvectors. To recover the orthogonality among such vectors, `dstein` reorthogonalizes the vectors based on the modified Gram-Schmidt procedure. Unfortunately, when the computed eigenvectors are nearly dependent, the eigenvectors may not be accurate after the reorthogonalization [33]. In addition, if many of the eigenvalues are close to each other, this reorthogonalization cost could become significant with $O(k^2 n)$ flops for computing k eigenvectors in the worst case. As a result, in our experiments shown in Figure 14, we saw that when computing all the singular values and vectors, bisection can be significantly slower than other



(a) square, with vectors (right graph is zoom of left graph)



(b) tall, 3:1, with vectors (right graph is zoom of left graph)



(c) tall, 1000:1, with vectors

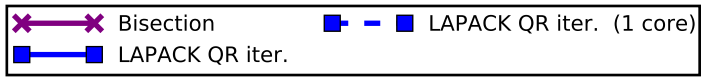


FIG. 14. *Comparison of bisection with QR iteration. Solid lines represent 16-core runs; dashed lines represent single core runs.*

methods. Even with 16 threads available, it is slower than the single-threaded QR iteration (dashed line). Bisection and inverse iteration are embarrassingly parallel—each eigenvalue and eigenvector may be computed independently—however, LAPACK does not currently include such explicit parallelization, instead primarily relying on parallelism within the BLAS, which is not advantageous in this case. On the other hand, as seen in Figure 15, when only a subset of k singular values and vectors are computed, we observed that bisection and inverse iteration (non-hatched bars) can be up to 2.4× faster than divide and conquer (D&C, black bar and dashed line), which must compute all the singular values and vectors. Depending on the matrix type, when computing $k = 400$ or $k = 600$ vectors out of $n = 3000$, it becomes faster to simply compute all the vectors using D&C. The exception here is the cluster, with one singular $\sigma_1 = 1$ and all other $\sigma_i = 1/\kappa$. In that case, computing any $k > 1$ vectors

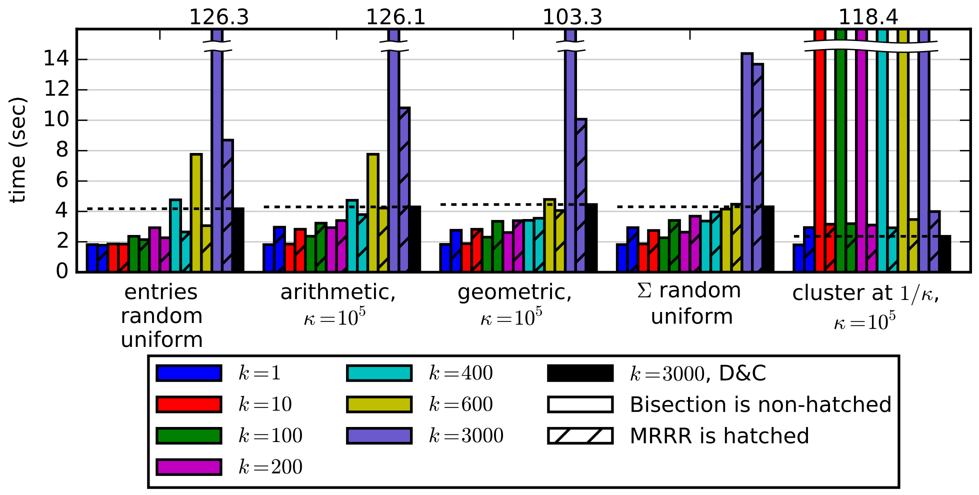


FIG. 15. *Results for computing k singular vectors of $n \times n$ matrix, $n = 3000$. Dashed lines shows D&C performance for computing all vectors. Number on top shows time for bars that exceed graph's height.*

was as slow as computing all vectors with bisection. See section 2 for a description of the matrices.

9. Multiple Relatively Robust Representations (MRRR). MRRR [35, 36] was developed to improve both the performance and accuracy of inverse iteration. Analysis has shown that MRRR can compute the numerically orthogonal eigenvectors of a symmetric tridiagonal matrix in $O(n^2)$ flops. At the time of preparing this paper, there was no publicly available software package that implements MRRR for computing the SVD of a general matrix, but there were at least two software packages that compute the eigenvalues and eigenvectors of a symmetric tridiagonal matrix using MRRR: `dstemr` of LAPACK [38], and `dstexr` due to Willems and Lang [118], which is tailored toward the tridiagonal matrix with zeros on the diagonal, as used in (7) for the SVD. For our experiments, we replaced the symmetric tridiagonal solver (`dstevx`) used in `dgesvdx` with `dstexr`. Performance with `dstemr` was generally similar but somewhat slower.

One of the main drawbacks of inverse iteration is that, for the eigenvalues with small relative gaps, the computed eigenvectors may not be orthogonal to each other. Hence, reorthogonalization is needed. This increases the computational cost and potentially leads to loss of accuracy in the computed eigenvectors. To address these issues, MRRR combines several techniques.

First, though the eigenvectors are invariant under a diagonal shift, we can increase their relative gaps by diagonally shifting the matrix. For instance, let us define the relative gap between two eigenvalues λ_i and λ_j to be $\frac{|\lambda_i-\lambda_j|}{\max(|\lambda_i|,|\lambda_j|)}$. Then, we can increase their relative gap by a factor of $\frac{|\lambda|}{|\lambda-\tau|}$ when we diagonally shift the matrix using a shift τ that is close to λ.

Hence, before applying inverse iteration, MRRR recursively refines the approximation to the eigenvalues and applies appropriate diagonal shifts to a cluster of eigenvalues such that it can guarantee large enough relative gaps between all the eigenvalues of T to maintain the orthogonality among the computed eigenvectors without reorthogonalization. For instance, given two approximate eigenvalues λ_i and λ_j, inverse iteration is used to compute their respective eigenvectors v_i and v_j with small residual norms, i.e.,

$$|Tv_k - \lambda v_k| = O(n\epsilon\,|T|) \text{ for } k = i \text{ and } j.$$

Then, according to [37, 38], a realistic bound on their orthogonality error is given by

$$\left|v_i^T v_j\right| = O\left(\frac{n\epsilon(|\lambda_i| + |\lambda_j|)}{|\lambda_i - \lambda_j|}\right).$$

Therefore, if the gap $|\lambda_i - \lambda_j|$ is of the same order as the eigenvalues, their eigenvectors are numerically orthogonal, i.e., $\left|v_i^T v_j\right| = O(n\epsilon)$.

There are several parameters that can be tuned to improve the performance [38], including the accuracy of the eigenvalue approximation computed at each step and the choice of the algorithm for computing the approximation (e.g., bisection, QR iteration, or Rayleigh quotient correction).

Second, while computing the eigenvalues (e.g., applying the diagonal shift), a small relative roundoff error in the entry of the tridiagonal matrix could result in a large relative error in the computed eigenvalues, especially in those with small magnitudes. To preserve the relatively high accuracy of the computed eigenvalues, MRRR stores the intermediate matrices in particular representations, referred to as the Multiple Relatively Robust Representation (MRRR) of the matrices. For instance, it has been shown that the LDL^T representation of the tridiagonal matrix T, without pivoting, is relatively robust, even with the presence of the element growth [31]. Hence, MRRR stores the sequence of intermediate matrices with different shifts in their LDL^T forms.

Third, for an eigenvalue with a small relative gap, the cost of inverse iteration may be high, requiring a few iterations to obtain the eigenvector with a small relative residual norm. Fortunately, there is at least one starting vector with which inverse iteration converges in one iteration. For example, when the i-th column of $(T - \lambda I)^{-1}$ has the largest column norm, then with the canonical vector e_i as the starting vector, one step of inverse iteration computes the approximate eigenvector x such that

$$|Tx - \lambda x| \le \sqrt{n}\left|\lambda - \bar{\lambda}\right|,$$

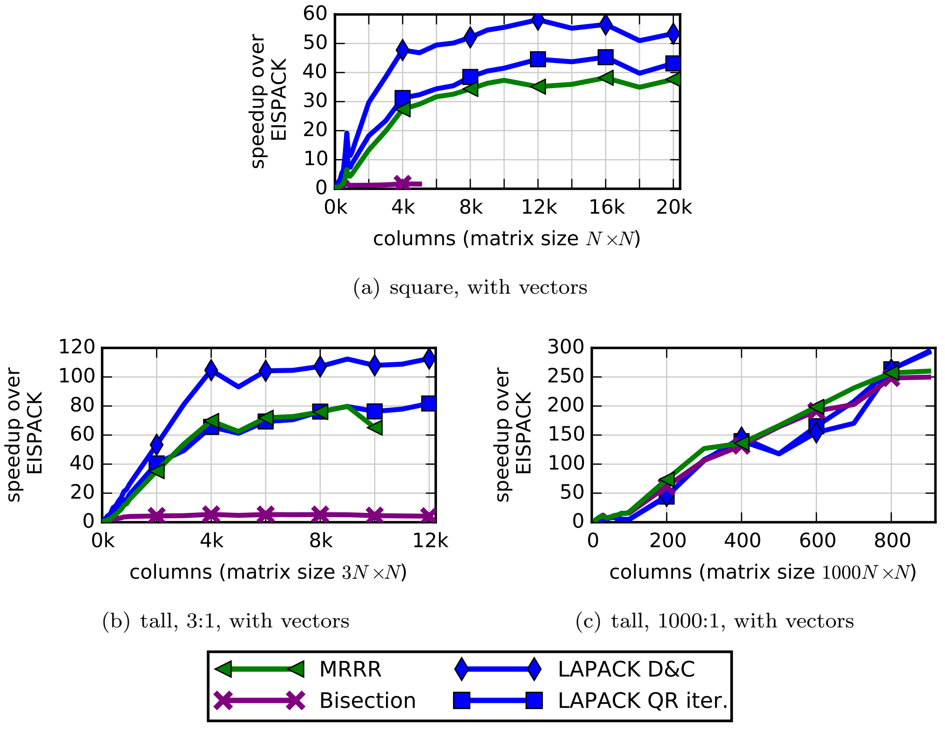
where $\bar{\lambda}$ is the exact eigenvalue [72]. Hence, if the eigenvalue is computed to a high relative accuracy,

$$\left|\lambda - \bar{\lambda}\right| = O(\epsilon\left|\bar{\lambda}\right|),$$

(e.g., using bisection with $O(n)$ flops), then the computed eigenpair obtains a small relative residual norm,

$$|Tx - \lambda x| = O(n\epsilon\left|\bar{\lambda}\right|).$$

There is an algorithm to find the column of $(T - \lambda I)^{-1}$ with largest norm with $O(n)$ flops [98]. In addition, if a twisted factorization is used to find the starting vector, then it can be shown that the computed eigenpairs have small residual norm with respect to the original matrix T [36, 37]. The twisted factorization must be carefully computed for T with a zero diagonal because the leading dimension of an odd dimension is singular. To enhance the numerical stability, `dstexr` computes a block variant of the factorization [118].



(a) square, with vectors

(b) tall, 3:1, with vectors

(c) tall, 1000:1, with vectors

FIG. 16. *Comparison of MRRR with QR iteration, divide and conquer, and bisection.*

As can be seen in Figure 16, by avoiding the reorthogonalization, MRRR can significantly improve the performance of inverse iteration, making MRRR comparable to QR iteration. However, divide and conquer is often faster.

Especially for large matrices, we noticed numerical issues where the backward error $\left\| A - U\Sigma V^T \right\| / (\min(m,n) \left\| A \right\|)$ was large, e.g., 10^{-4} instead of 10^{-16} as expected. Further tests in section 13 show that, even when the above error is acceptable, MRRR has poor relative error for the singular values. Marques and Vasconcelos [86] also observed numerical issues with the existing MRRR implementation.

When only a subset of k singular vectors are computed, we observe in Figure 15 that inverse iteration can be up to 1.6× faster than MRRR for a small number vectors ($k = 1$ or 10). For a larger subset of $k = 600$ vectors, MRRR can be up to 1.8× faster than bisection, but in this case, only for the random entries matrix is MRRR significantly faster (1.3×) than computing all the singular vectors with divide and conquer. The exception is the cluster matrix, where for $k > 1$, MRRR is 30× faster than bisection, but always slower than using divide and conquer.

10. MAGMA Implementation for Accelerator Architectures. Accelerators such as GPUs and the Intel Xeon Phi provide a high degree of parallelism and a larger memory bandwidth than traditional multicore CPUs. The MAGMA library was developed to address this new architecture, and accelerates most phases of the SVD algorithm: reduction to bidiagonal, bidiagonal D&C, and computation of singular vectors. For tall-skinny matrices, it also accelerates the initial QR factorization and generating Q.

The most prominent place to start is an accelerated version of the bidiagonal

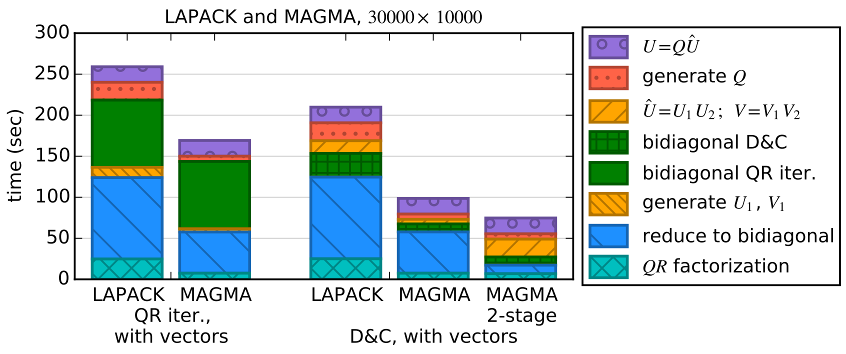
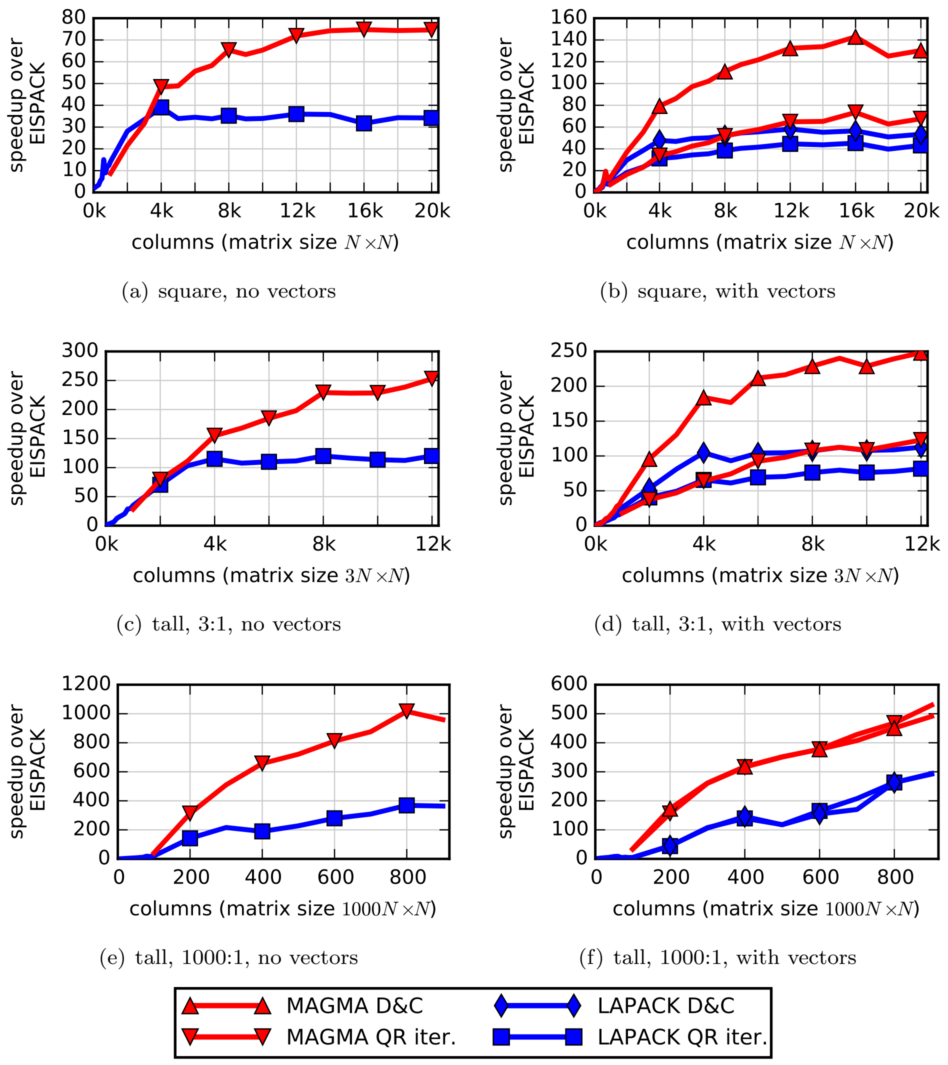


FIG. 17. *Profile comparing LAPACK and MAGMA. Most phases are accelerated using the GPU, except the bidiagonal QR iteration and multiplying $U = Q\hat{U}$. MAGMA 2-stage is described in section 11.*

reduction [109]. We have seen in Figures 7 and 12 that this phase (blue tier with \\ hatching) takes from 50% to 70% of the time for a square matrix when computing singular vectors, and 98% of the time when computing only singular values (no vectors). As described in section 5, the bidiagonal reduction has half its flops in Level 2 BLAS and half in Level 3 BLAS. Accelerators are known for achieving very high performance on compute-intensive, Level 3 BLAS operations. On an NVIDIA K40c GPU, cuBLAS achieves 1245 Gflop/s with `dgemm`, compared with 315 Gflop/s using Intel MKL on the multicore CPU. Due to the accelerator's large memory bandwidth, the memory-bound Level 2 BLAS operations are also significantly faster, achieving 45 Gflop/s with cuBLAS `dgemv`, compared with 14 Gflop/s on the multicore CPU. Therefore, both the trailing matrix-vector product (`dgemv`) and the trailing matrix update (`dgemm`) are performed on the accelerator. The small panel operations—constructing Householder reflectors—are performed on the CPU, which is better at serial operations with more control flow. This incurs CPU-to-GPU communication of a couple of vectors for each `dgemv` operation during the panel. Due to dependencies, the trailing matrix update cannot be overlapped with the next panel, as would occur in a one-sided QR factorization. Using the accelerator improves the speed of the bidiagonal reduction by about a factor of 2, as shown by the profile in Figure 17 (blue tier with \\ hatching) and by the square, no vectors case in Figure 18(a), which is dominated by the bidiagonal reduction.

For the bidiagonal SVD, because D&C is faster than QR iteration, MAGMA will inherently achieve a better overall speedup using D&C. We further implement an accelerated version of D&C [51]. Since most of the operations in D&C are in multiplying $Q_r U_m$ and $V_m^T W_r$ to generate singular vectors, these Level 3 BLAS `dgemm` operations are assigned to the accelerator. The solution of the secular equation to find the singular values of M_c is left on the CPU, since it is a complex iterative algorithm with limited parallelism, as is computing the singular vectors U_m and V_m of M_c. These are parallelized on the CPU using OpenMP. MAGMA achieves about a 3× speedup for the D&C phase compared to LAPACK.

For a tall-skinny ($m \gg n$) matrix, we accelerate the initial QR factorization [110]. This is a one-sided factorization, so it doesn't have the extra dependencies imposed by the two-sided reduction to bidiagonal form. Panel operations are within a simple block column that doesn't involve the trailing matrix. The panel factorization is performed



(a) square, no vectors

(b) square, with vectors

(c) tall, 3:1, no vectors

(d) tall, 3:1, with vectors

(e) tall, 1000:1, no vectors

(f) tall, 1000:1, with vectors

FIG. 18. *Comparison of MAGMA with LAPACK.*

on the CPU, while the trailing matrix update is performed on the accelerator. The accelerator updates the next panel first and sends it back to the CPU so the CPU can start factoring it while the accelerator proceeds with the rest of the trailing matrix update. This overlap allows the factorization to achieve a substantial portion of the peak `dgemm` speed, up to 970 Gflop/s with an NVIDIA K40c. The QR factorization phase was up to 3.6× faster than on the multicore CPU, as seen in Figure 17 (cyan tier with × hatching).

There are three routines that are solely applying block Householder reflectors, which are implemented as a series of Level 3 BLAS matrix multiplies entirely on the accelerator: (1) for QR iteration, generating explicit U_1 and V_1 matrices (`dorgbr`) (2) for D&C, multiplying $U_1 U_2$ and $V_1 V_2$ (`dormbr`), and (3) for a tall-skinny matrix, generating an explicit Q matrix (`dorgqr`). These were up all up to 3.3× faster when

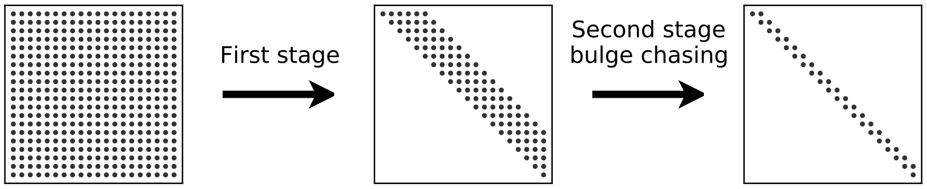


FIG. 19. *Two-stage technique for the reduction phase.*

using the accelerator than when using the multicore CPU. We see in Figure 17 that the time for all three of these phases is substantially reduced.

Overall, MAGMA achieves significant improvements using an accelerator for the SVD problem. Figure 18 shows that it is about 2× faster than LAPACK in most cases. For the square, vectors case in Figure 18(b), MAGMA's SVD using D&C is 2.5× LAPACK's D&C version, and 2× MAGMA's SVD using QR iteration, while MAGMA's SVD using QR iteration is only 1.6× LAPACK's QR iteration version, due to both D&C being inherently faster and having an accelerated version of the D&C phase. In the tall 1000:1 case in Figure 18(e), MAGMA is 2.6× faster, and for some sizes as much as 3.5× faster, than LAPACK, and up to 1000× faster than EISPACK, due to the accelerated QR factorization.

11. Two-stage Reduction.

While all the preceding algorithmic and architectural improvements have greatly increased the speed of the SVD, all these one-stage methods remain limited by the memory-bound, Level 2 BLAS operations. To overcome the limitations of the one-stage approach, Großer and Lang [78, 55] introduced the two-stage bidiagonal reduction, which increases the use of compute-intensive Level 3 BLAS operations. The idea behind the two-stage algorithm is to split the original one-stage bidiagonal reduction into a compute-intensive phase (first stage) and a memory-bound phase (second or *bulge-chasing* stage), as represented in Figure 19. The first stage reduces the original general dense matrix to a band form (either upper or lower), and the second stage reduces the band form to bidiagonal form (again, either upper or lower). The algorithm maps computational tasks to the strengths of the available hardware components, taking care of the data reuse. It also uses techniques to mix between dynamic and static scheduling to extract efficiency and performance. We implemented two-stage algorithms in the PLASMA library for multicore environments [82, 83, 62, 60], the DPLASMA library for distributed environments [19, 18], and the MAGMA library for accelerator architectures [51]. Similar two-stage reduction [61] and multi-stage successive band reduction (SBR) [13, 6] to tridiagonal have been used for the symmetric eigenvalue problem. A multi-stage approach would also work for the bidiagonal reduction, and could be advantageous to achieve optimal communication costs at each stage. However, when computing singular vectors, each stage adds cost to the back transformation, making a multi-stage approach less favorable.

11.1. First Stage: Compute-Intensive and Efficient Kernels.

The first stage applies a sequence of blocked Householder transformations to reduce the general dense matrix to an upper (for $m \geq n$) band matrix. This stage uses compute-intensive matrix-multiply kernels that eliminate the memory-bound matrix-vector products from the one-stage panel factorization.

The first stage proceeds by computing a QR factorization of a block column to annihilate entries below the diagonal, and updating the trailing matrix, as shown in Figure 20. It then computes an LQ factorization of a block row to annihilate entries right of the upper bandwidth, and updates the trailing matrix. It repeats factoring block columns and block rows, until the entire matrix is brought to band form. The width of the block columns and rows is the resulting matrix bandwidth, n_b.

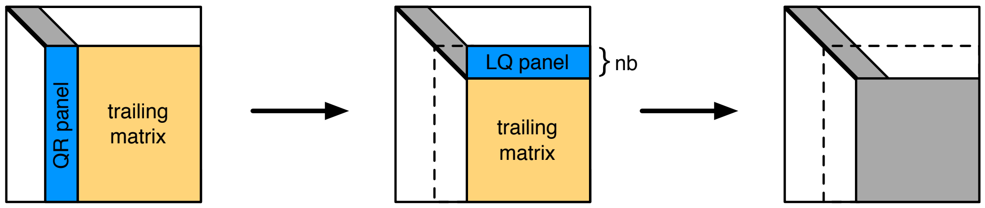


Fig. 20. *One panel of the first stage reduction to band form.*

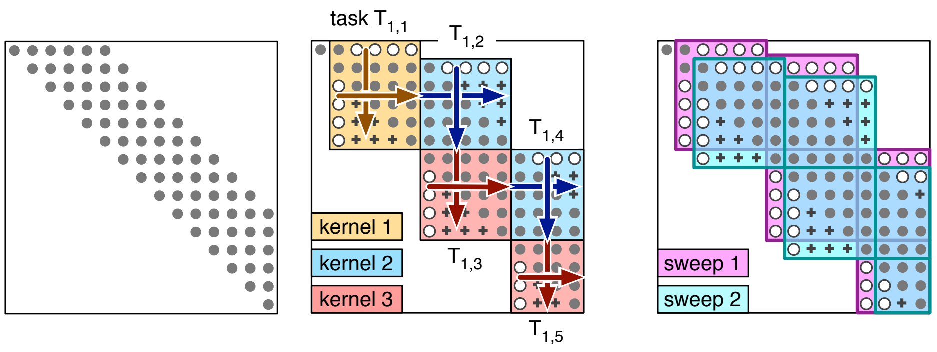
The PLASMA and DPLASMA implementations use a tile algorithm [1] that makes it highly parallel. The matrix is split into tiles of size $n_b \times n_b$, where n_b is the matrix bandwidth. Data within each tile is stored contiguously in memory. A panel factorization is a series of QR or LQ factorizations done between pairs of tiles; once a pair of tiles has been factored, updates on the corresponding portions of the trailing matrix can start immediately, before the rest of the panel has finished factoring. This unlocks a large amount of parallelism very quickly. The algorithm then proceeds as a collection of interdependent tasks that operate on the tile data layout and are scheduled in an out-of-order fashion using either the OpenMP runtime for PLASMA or the powerful PaRSEC distributed runtime system for DPLASMA.

The MAGMA implementation uses a standard column-wise layout. It does the QR and LQ factorizations on the CPU, copies the block Householder reflectors to the accelerator, and updates the trailing matrix on the accelerator. Unlike in the one-sided factorizations, it cannot start the next panel until the trailing matrix update is finished due to data dependencies.

The first stage's cost is $\frac{8}{3}n^3$ operations in Level 3 BLAS. As shown in [60], the performance of this stage is comparable to the performance of the QR factorization and can reach a high percentage of the machine's peak.

11.2. Second Stage: Cache-Friendly Computational Kernels. The second stage reduces the band form to the final bidiagonal form using a bulge chasing technique. It involves $6n_bn^2$ operations, so it takes a small percentage of the total operations, which decreases with n. The operations are memory bound, but are fused together as Level 2.5 BLAS [69] for cache efficiency. We designed the algorithm to use fine-grained, memory-aware tasks in an out-of-order, data-flow task-scheduling technique that enhances data locality [60, 61].

The second stage proceeds in a series of sweeps, each sweep bringing one row to bidiagonal and chasing the created fill-in elements down to the bottom right of the matrix using successive orthogonal transformations. It uses three kernels. Kernel 1 (yellow task $T_{1,1}$ in Figure 21(b)) applies a Householder reflector from the right (indicated by the down arrow) to eliminate a row right of the superdiagonal, which also creates a bulge of fill-in beneath the diagonal. It then applies a Householder reflector from the left (indicated by the right arrow) to eliminate the first column of the bulge below the diagonal, and applies the update to the first block column only. The



(a) initial band matrix (b) tasks in sweep 1 (c) overlap of sweeps

FIG. 21. *Bulge-chasing algorithm. "o" indicates eliminated elements; "+" indicates fill. Arrows show application of Householder reflector on left (→), which update a block row, and on right (↓), which update a block column.*

remainder of the bulge is not eliminated, but is instead left for subsequent sweeps to eliminate, as they would reintroduce the same nonzeros.

Kernel 2 (blue task $T_{1,2}$) continues to apply the left Householder reflector from kernel 1 (or kernel 3) to the next block column, creating a bulge above the upper bandwidth. It then applies a right Householder reflector to eliminate the first row of the bulge right of the upper bandwidth, updating only the first block row.

Kernel 3 (red task $T_{1,3}$) continues to apply the right Householder reflector from kernel 2, creating a bulge below the main diagonal. As in kernel 1, it then applies a left Householder reflector to eliminate the first column of the bulge below the diagonal and updates just the current block column. After kernel 3, kernel 2 is called again (blue task $T_{1,4}$) to continue application of the left Householder reflector in the next block column. A sweep consists of calling kernel 1 to bring a row to bidiagonal, followed by repeated calls to kernels 2 and 3 to eliminate the first column or row of the resulting bulges, until the bulges are chased off the bottom-right of the matrix.

For parallelism, once a sweep has finished the first kernel 3, a new sweep can start in parallel. This new sweep is shifted over one column and down one row, as shown in Figure 21(c). Before task i in sweep s, denoted as $T_{s,i}$, can start, it depends on task $T_{s-1,\,i+3}$ in the previous sweep being finished, to ensure that kernels do not update the same entries simultaneously. To maximize cache reuse, tasks are assigned to cores based on their data location. Ideally, the band matrix fits into the cores' combined caches, and each sweep cycles through the cores as it progresses down the band.

11.3. Singular Vectors Computation. The singular vectors of A are computed from the orthogonal transformations used in the reduction to bidiagonal form and from the singular vectors of the bidiagonal form. Recall that for the classical one-stage approach, $A = U_1 B V_1^T$ and $B = U_2 \Sigma V_2^T$. After using D&C to obtain U_2 and V_2, we multiply $U = U_1 U_2$ and $V = V_1 V_2$, costing $2n^3$ each for U and V (if $m = n$).

In the case of the two-stage approach, the first stage reduces the original matrix A to a band matrix by applying a two-sided transformation to A such that $A = U_a A_{\text{band}} V_a^T$. Similarly, the second, bulge-chasing stage reduces the band matrix A_{band} to bidiagonal form by applying a two-sided transformation such that $A_{\text{band}} = U_b B V_b^T$.

As a consequence, the singular vectors must be multiplied according to:

$$U = U_a U_b U_2 \qquad \text{and} \qquad V = V_a V_b V_2.$$

Hence the two-stage approach introduces a nontrivial amount of extra computation—the application of U_b and V_b—when the singular vectors are needed. The total cost of updating the singular vectors when using the two-stage technique is $2(1+\frac{i_b}{n_b})n^3+2n^3$ each for U and V, where n_b is the bandwidth and i_b is an internal blocking; usually $i_b \leq n_b/4$. This extra cost compared with the one-stage approach reduces the potential speedup, but as it is in Level 3 BLAS, it does not completely negate the large speedup that we gain by the two-stage bidiagonal reduction.

11.4. PLASMA Implementation for Multicore. The experiments shown in Figure 22 illustrate the superior efficiency of our two-stage SVD solver compared with the optimized LAPACK version from Intel MKL. Figure 22(a) shows that the bidiagonal reduction itself is 6× faster than LAPACK, both using 16 cores, and 2.5× faster than the MAGMA one-stage version. The reason is that LAPACK and MAGMA are bound by the Level 2 BLAS performance, while our two-stage algorithm relies on Level 3 BLAS for most of its computation. When computing singular vectors in Figure 22(b), it is still about 1.8× faster than LAPACK, even though it requires an extra $2 \times 2(1 + \frac{i_b}{n_b})n^3$ operations to multiply by U_b and V_b. Here, the accelerated MAGMA one-stage version is still faster.

For the tall 3:1 case in Figure 22(c), both LAPACK and MAGMA fare better, since part of the computation is in the initial QR factorization, which is primarily efficient Level 3 BLAS operations for all three implementations (LAPACK, MAGMA, and PLASMA). For the very tall 1000:1 matrices in Figures 22(e) and 22(f), PLASMA and MAGMA rely on their efficient QR factorization. In PLASMA, this is an implementation of the tall-skinny QR [1, 29], which even beats the accelerated MAGMA implementation.

Overall, we expected such an improvement using the two-stage technique, due to its heavy reliance on Level 3 BLAS. Even when performing more operations, it can still have an advantage.

11.5. Energy Consumption. As we move toward exascale computing, power and energy consumption play increasingly critical roles. Figure 23 shows the power consumption over time during the SVD computation. We observe that PLASMA has the lowest energy consumption, due to its fast execution, despite having the highest power rate, indicative of its high compute intensity using Level 3 BLAS. Its energy consumption is about half that of LAPACK, and 23× less than EISPACK, as shown in Figure 24(b). When computing singular values only, no vectors, the difference is even more remarkable, with PLASMA being 5.6× more energy efficient than LAPACK, and 40× more energy efficient than EISPACK, as shown in Figure 24(a).

Interestingly, we can correlate the various phases of the computation with the power consumption. For LAPACK, the long plateau in Figure 23 up to the 105 seconds mark is the reduction to bidiagonal, followed by divide and conquer, where the power varies significantly, and ending with the two back transformations by U_1 and V_1 from the 130–150 seconds mark. In PLASMA, the reduction to bidiagonal is significantly shorter, up to the 20 seconds mark, followed by divide and conquer, and the back transformations by U_a, U_b, V_a, and V_b, which are twice as long as they are in LAPACK. EISPACK, in contrast, has a very long and steady computation. It uses only one core, and thus has low power consumption; but the computation itself is 48× longer than LAPACK.

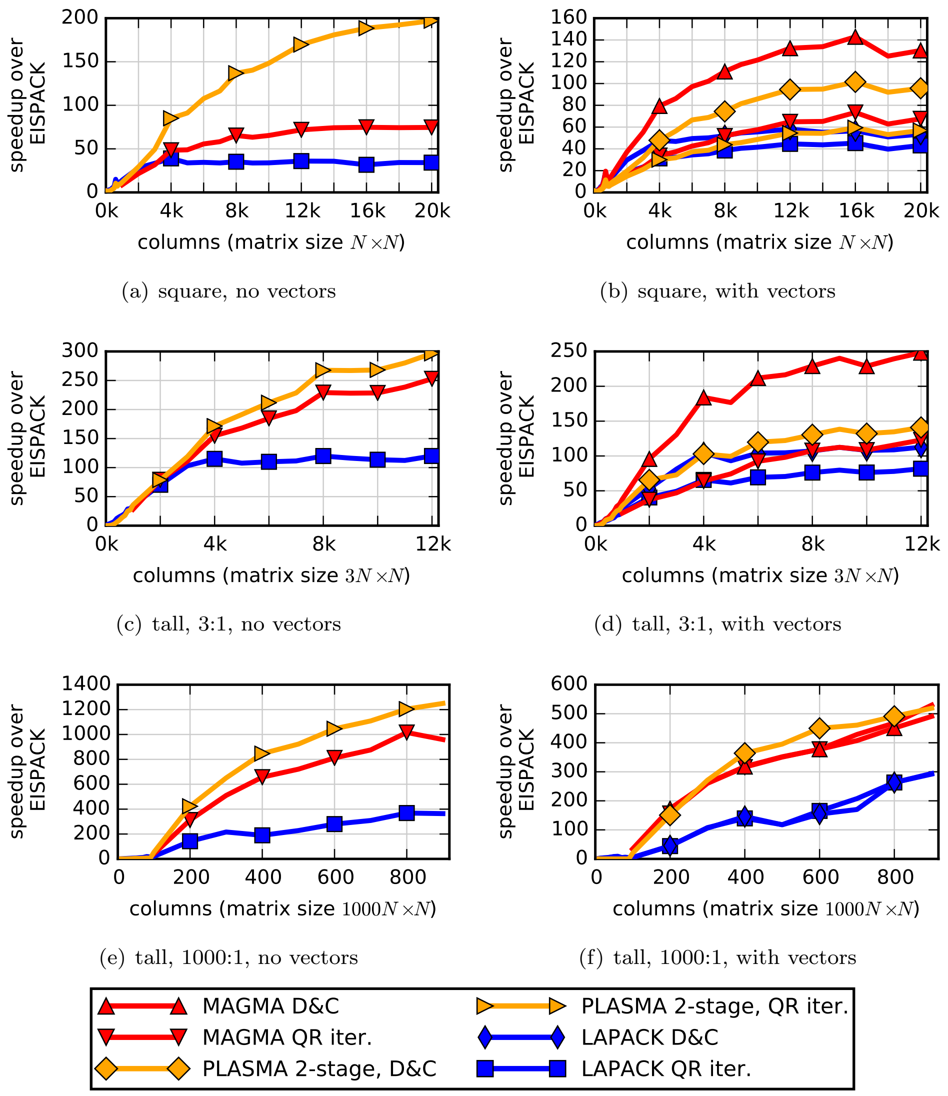


FIG. 22. *Comparison of MAGMA, PLASMA, and LAPACK.*

11.6. MAGMA Accelerated Two-stage Reduction. A two-stage algorithm can also be implemented very effectively using an accelerator. MAGMA accelerates the first-stage reduction to band form, as described above, and uses PLASMA for the second-stage reduction from band to bidiagonal. MAGMA also accelerates computation of singular vectors, both applying the transformations from the second stage (e.g., $U_b U_2$) and applying the transformations from the first stage (e.g., $U_a(U_b U_2)$). Other steps are as in the accelerated one-stage MAGMA version. The profile in Figure 17 shows the difference with the one-stage version: the reduction to bidiagonal (blue with \\ hatching) is significantly reduced, but multiplying $U = U_1 U_2 = U_a U_b U_2$ and $V = V_1 V_2 = V_a V_b V_2$ (orange with // hatching) is increased.

Figure 25 shows the performance of the MAGMA two-stage implementation

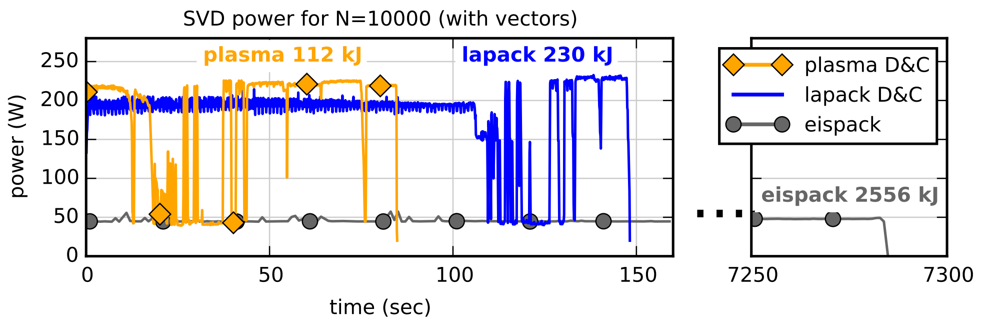
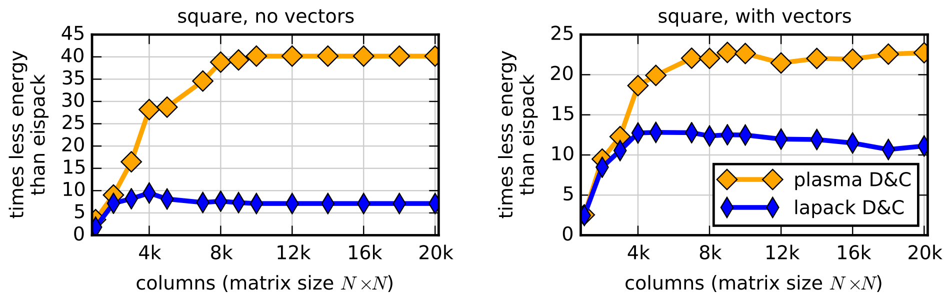


FIG. 23. *Comparison of power during SVD computation for PLASMA, LAPACK, and EISPACK, for square matrix of size $n = 10000$. The total energy consumed during the computation is annotated for each.*



(a) square, no vectors (b) square, with vectors

FIG. 24. *Reduction in total energy consumption compared to EISPACK.*

(dashed line), compared with the PLASMA two-stage and MAGMA one-stage implementations. The square, no vectors case in Figure 25(a) shows that for the bidiagonal reduction itself, the two-stage MAGMA is up to 2.4× faster than the two-stage PLASMA and 6.4× faster than the one-stage MAGMA, and nearly 500× faster than EISPACK. When computing singular vectors, in Figure 25(b), it is again up to 2.4× faster than PLASMA, but only 1.7× faster than the one-stage MAGMA, due to the extra cost in multiplying by U_b and V_b. It also performs well in the tall 3:1 case, while for the tall 1000:1 case, its time is dominated by the initial QR factorization, so it performs similarly to the one-stage MAGMA.

11.7. DPLASMA Implementation for Distributed Memory. To cover the distributed memory environment, we also performed a study on a modern, large distributed system. It is representative of a vast class of supercomputers commonly used for computationally intensive workloads. The DPLASMA algorithm is the two-stage algorithm described above for multicore, but implemented using the PaRSEC runtime engine [19, 18] to exploit the data flow representation, handle all the communication, and provide asynchronous task execution based on dependency analysis. PaRSEC employs the dataflow programming and execution model to provide a dynamic platform that can address the challenges posed by distributed hardware resources. The PaRSEC runtime combines the source program's task and dataflow information with supplementary information provided by the user—such as data distribution or hints

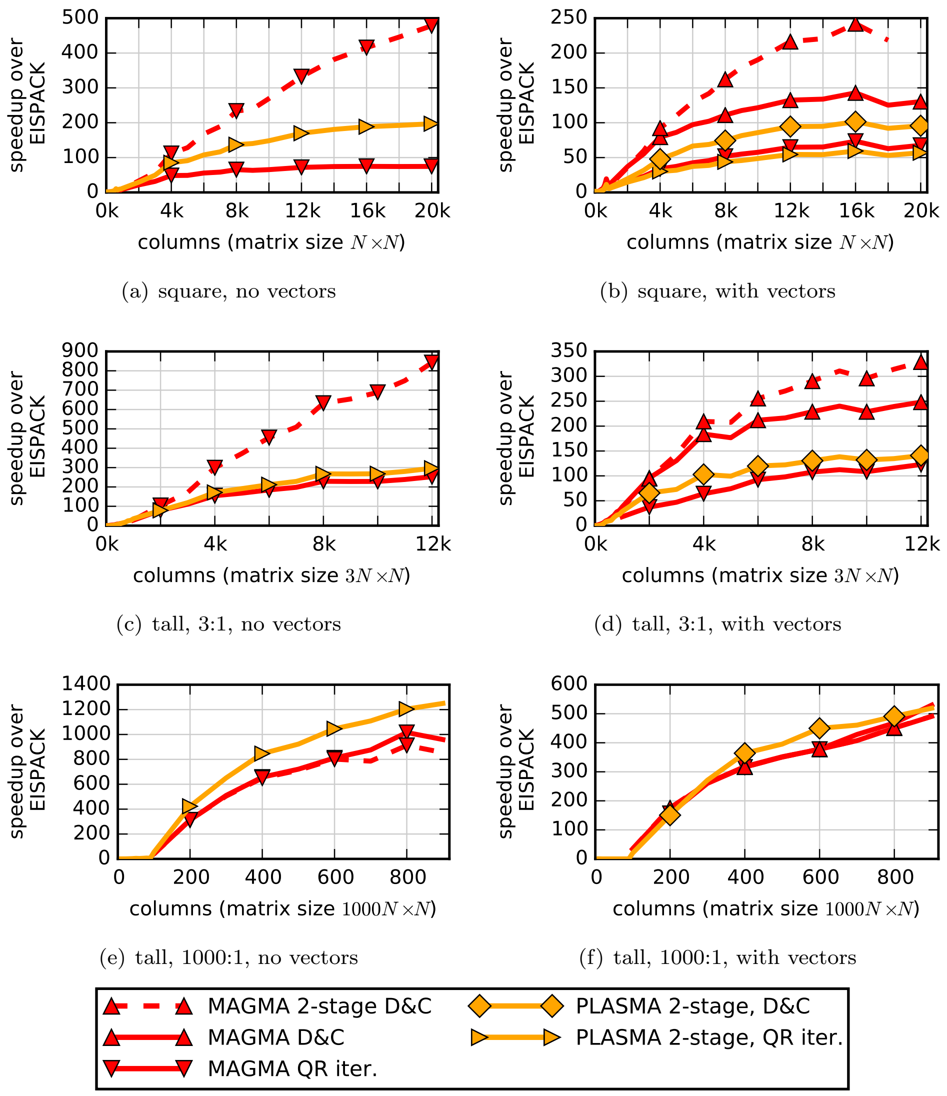
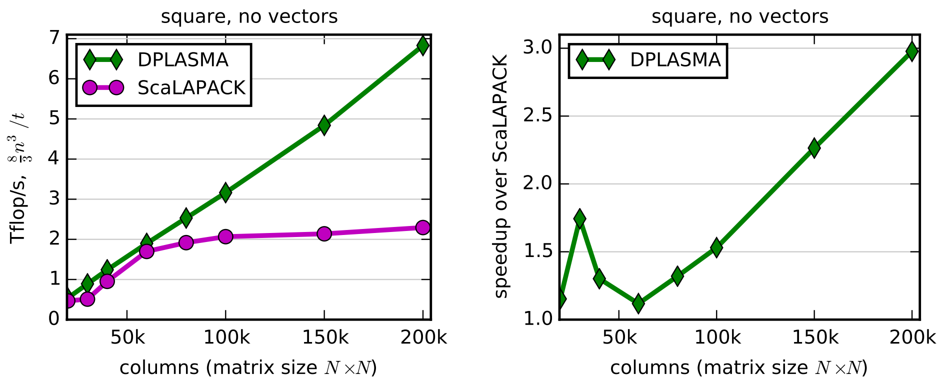


FIG. 25. *Comparison of MAGMA 2-stage with MAGMA 1-stage and PLASMA 2-stage.*

about the importance of different tasks—and orchestrates task execution on the available hardware. From a technical perspective, PaRSEC is an event-driven system. When an event occurs, such as task completion, the runtime reacts by examining the dataflow to discover what future tasks can be executed based on the data generated by the completed task. The runtime handles the data exchange between distributed nodes, and thus it reacts to the events triggered by the completion of data transfers as well. Thus, communications become implicit and are handled automatically as efficiently as possible by the runtime. When no events are triggered because the hardware is busy executing application code, the runtime gets out of the way, allowing all hardware resources to be devoted to the application code's execution.

We benchmarked our two-stage implementation from the DPLASMA library, and



(a) square, no vectors, performance (b) square, no vectors, speedup

FIG. 26. *Comparison of DPLASMA and ScaLAPACK computing singular values only for square matrices on 49 nodes (1764 cores).*

the ScaLAPACK SVD routine from Intel MKL. Because only the singular values computation of our two-stage approach is currently implemented in the distributed DPLASMA library, we limited our tests to the case where only the singular values are computed. We performed our experiment on a recent hardware system consisting of 49 distributed nodes, where every node has two sockets of 18-core Intel Xeon E5-2697 (Broadwell) processors, running at 2.6 GHz, providing a total of 1764 cores. Each socket has 35 MiB of shared L3 cache, and each core has a private 3.5 MiB L2 and 448 KiB L1 cache. The system is equipped with 52 GiB of memory per node. When only singular values are to be computed, the SVD solution consists of the reduction to bidiagonal and the computation of the singular values using QR iteration. Note that QR iteration on the bidiagonal matrix is a sequential process and thus it does not exploit any parallelism for either DPLASMA or ScaLAPACK. Its computational time is the same on either 1 or 49 nodes, and this time increases quadratically with the matrix size. Thus, the percentage of time spent in this portion varies with the matrix size. QR iteration consists of less than 5% of the time for a matrix of size 20k, while it reaches about 15% for ScaLAPACK and 26% for DPLASMA for a matrix of size 200k. As a result, the speedup will be affected by this constant:

$$\text{speedup} = \frac{\text{time}_{\text{DPLASMA-BRD}} + t_x}{\text{time}_{\text{SCALAPACK-BRD}} + t_x},$$

where t_x is the time required to perform the bidiagonal singular value computation. Figure 26 shows the comparison between our implementation versus the ScaLAPACK `pdgesvd`. Asymptotically, our code achieves up to a 3× speedup for the largest matrices tested. This is the result of the efficient implementation of the first stage (reduction to band) using the PaRSEC engine, which enables us to exploit the compute-intensive nature of this stage, thereby minimizing the communication cost, and also from the careful design and implementation of the second stage that maps both the algorithm and the data to the hardware using cache-friendly kernels and data-locality-aware scheduling. Note that for small matrix sizes (e.g., a matrix of size 20k), there is not enough parallelism to exploit the 1764 available cores to make our two-stage algorithm 3× faster; the tile size is about 160, so there are only about 125 tiles in each direction.

12. Jacobi methods. In contrast to bidiagonalization methods, Jacobi methods operate on the entire matrix A, without ever reducing to bidiagonal. This allows Jacobi methods to attain high relative accuracy, which will be discussed in section 13. Jacobi first proposed his method in 1848 for solving the symmetric eigenvalue problem [73] by diagonalizing the matrix A using a sequence of plane rotations:

$$A_{(0)} = A, \qquad A_{(k+1)} = J_{(k)}^T A_{(k)} J_{(k)}, \qquad A_{(k)} \to \Lambda \text{ as } k \to \infty.$$

Each plane rotation, $J_{(k)} = J_{(k)}(i, j, \theta)$, now called a Jacobi or Givens rotation, is an orthogonal matrix that differs from the identity only in rows and columns i and j:

$$J(i,j,\theta) = \begin{bmatrix} I & & & & \\ & c & & s & \\ & & I & & \\ & -s & & c & \\ & & & & I \end{bmatrix},$$

where $c = \cos\theta$ and $s = \sin\theta$. The angle θ is chosen to eliminate the pair a_{ij}, a_{ji} by applying $J(i,j,\theta)$ on the left and right of A, which can be viewed as the 2×2 eigenvalue problem,

$$\hat{J}_{(k)}^T \hat{A}_{(k)} \hat{J}_{(k)} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} d_{ii} & 0 \\ 0 & d_{jj} \end{bmatrix} = \hat{A}_{(k+1)},$$

where the notation $\hat{A}$ is the 2×2 submatrix $\begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}$ of matrix A. Subsequent eliminations will fill in the eliminated entry, but at each step the norm of off-diagonal elements,

$$\operatorname{off}(A) = \|A - \operatorname{diag}(A)\|_F = \Big(\sum_{i \neq j} a_{ij}^2\Big)^{1/2},$$

is reduced until the matrix converges to diagonal form, Λ, revealing the eigenvalues. Accumulating the plane rotations, $V = J_{(0)} J_{(1)} \ldots$, yields the eigenvectors. Originally, Jacobi chose to eliminate the off-diagonal pair a_{ij}, a_{ji} of largest magnitude at each step, giving the largest possible reduction in $\operatorname{off}(A)$. This is inefficient as it introduces an $O(n^2)$ search for each rotation of $O(n)$ work. Instead, in modern times the method was reformulated so that one *sweep* goes over all $n(n-1)/2$ combinations of (i, j) with $i < j$ in a predetermined order, typically cyclic by rows, i.e.,

$$(1,2), (1,3), \ldots, (1,n); (2,3), \ldots, (2,n); \ldots; (n-1,n),$$

or cyclic by columns. It converges after a small number of sweeps, typically 5–10. Wilkinson [116] showed that convergence is ultimately quadratic. Rutishauser [102] gave a robust implementation in the Wilkinson-Reinsch Handbook.

12.1. Two-sided Jacobi SVD. Jacobi's eigenvalue method was generalized to the SVD of a general, nonsymmetric matrix in two different ways. The first way is the two-sided method due to Kogbetliantz [76], which applies two different plane rotations, $J(i,j,\theta)$ on the left of A and $K(i,j,\phi)$ on the right of A, to eliminate the a_{ij} and a_{ji} entries. As before, sweeps are done over the off-diagonal entries until

Algorithm 5 Two-sided Jacobi SVD method for $n \times n$ matrix A.

```
function two_sided_jacobi_svd( $A$ )
    $U = I$; $V = I$
    repeat    // loop over sweeps
        for each pair $(i, j)$, $i < j$, in prescribed order
            solve 2 × 2 SVD $\hat{J}^T \hat{A}_{(k)} \hat{K} = \hat{A}_{(k+1)}$
            $A = J^T A$    // update rows $i$ and $j$
            $A = AK$    // update cols $i$ and $j$
            $U = UJ$
            $V = VK$
        end
    until off$(A) <$ tol $\|A_0\|_F$
    for $i = 1, \ldots, n$
        $\sigma_i = |a_{ii}|$
        if $a_{ii} < 0$ then $u_i = -u_i$
    end
    sort $\Sigma$ and apply same permutation to columns of $U$ and $V$
    return $(U, \Sigma, V)$
end function
```

the norm of off-diagonal entries is below a specified tolerance, revealing the singular values, Σ, via the iteration:

$$A_{(0)} = A, \qquad A_{(k+1)} = J_{(k)}^T A_{(k)} K_{(k)}, \qquad A_{(k)} \to \Sigma \text{ as } k \to \infty.$$

Accumulating the left rotations, $U = J_{(0)} J_{(1)} \ldots$, gives the left singular vectors, while accumulating the right rotations, $V = K_{(0)} K_{(1)} \ldots$, gives the right singular vectors.

Determining $J(i, j, \theta)$ and $K(i, j, \phi)$ can be viewed as solving a 2×2 SVD problem,

$$(8) \qquad \hat{J}_{(k)}^T \hat{A}_{(k)} \hat{K}_{(k)} = \begin{bmatrix} c_J & s_J \\ -s_J & c_J \end{bmatrix}^T \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix} \begin{bmatrix} c_K & s_K \\ -s_K & c_K \end{bmatrix} = \begin{bmatrix} d_{ii} & \\ & d_{jj} \end{bmatrix} = \hat{A}_{(k+1)}.$$

The angles for J and K are not uniquely determined, so various methods have been derived [22, 49, 76]. Brent et al. [22] proposed the algorithm USVD, which uses one rotation to symmetrize the 2×2 subproblem, then a second rotation to eliminate the off-diagonal entries. This produces an unnormalized SVD, where the diagonal entries are unsorted and may be negative. Post-processing to sort and adjust the signs of the singular values and singular vectors yields a standard SVD. They also formulated the normalized rotation/reflection algorithm NSVD that corrects the signs during the iteration. Algorithm 5 outlines the two-sided Jacobi method.

Rectangular matrices can be handled by first doing a QR factorization, optionally with pivoting, and then doing the SVD of the R matrix, as previously described for bidiagonalization methods (subsection 5.4). For Jacobi, this QR factorization has the added benefit of preconditioning the system to converge faster, as discussed further in subsection 12.5.

Heath et al. [67] developed a variant for computing the SVD of a product of matrices, $A = B^T C$, without explicitly forming A. Applying rotations $B_{(k+1)} = B_{(k)} J$ and $C_{(k+1)} = C_{(k)} K$ implicitly applies J and K on both sides of A. When $B = C$, it simplifies to the one-sided Jacobi method, discussed next.

Algorithm 6 One-sided Jacobi SVD method for $m \times n$ matrix A, $m \geq n$

```
function one_sided_jacobi_svd( $A$ )
    $V = I$
    repeat    // loop over sweeps
        done = true
        for each pair $(i, j)$, $i < j$, in prescribed order
            $b_{ii} = A_i^T A_i = \|A_i\|^2$
            $b_{jj} = A_j^T A_j = \|A_j\|^2$
            $b_{ij} = A_i^T A_j$
            if $|b_{ij}| \geq \epsilon\sqrt{b_{ii} b_{jj}}$ then
                solve $2 \times 2$ symmetric eigenvalue problem $\hat{J}^T \hat{B} \hat{J} = \hat{D}$
                $A = AJ$        // update cols $i$ and $j$
                $V = VJ$
                done = false
            end
        end
    until done
    for $i = 1, \ldots, n$
        $\sigma_i = \|a_i\|_2$
        $u_i = a_i / \sigma_i$
    end
    sort $\Sigma$ and apply same permutation to columns of $U$ and $V$
    return $(U, \Sigma, V)$
end function
```

12.2. One-sided Jacobi. The second way to generalize the Jacobi method to the SVD is a one-sided method due to Hestenes [68]. Earlier we noted that the SVD can be solved by computing the eigenvalues of the Gram matrix, $A^T A$, but that explicitly forming $A^T A$ is undesirable for numerical reasons. Instead, Hestenes applied plane rotations on only the right side of A to orthogonalize the columns of A, which implicitly performs the two-sided Jacobi eigenvalue method on $A^T A$. The columns of A converge to $U\Sigma$, that is, the left singular vectors scaled by the singular values:

$$A_{(0)} = A, \qquad A_{(k+1)} = A_{(k)} J_{(k)}, \qquad A_{(k)} \to U\Sigma \text{ as } k \to \infty.$$

This means that, implicitly, ${A_{(k)}}^T A_{(k)} \to \Sigma^2$. Accumulating the rotations, $V = J_{(0)} J_{(1)} \ldots$, gives the right singular vectors. Alternatively, V can be solved for after the iteration, as described below in subsection 12.5.

The rotations are determined similarly to the Jacobi eigenvalue method, by solving the 2×2 eigenvalue problem

$$\hat{J}_{(k)}^T \begin{bmatrix} b_{ii} & b_{ij} \\ b_{ij} & b_{jj} \end{bmatrix} \hat{J}_{(k)} = \begin{bmatrix} d_{ii} & \\ & d_{jj} \end{bmatrix}, \tag{9}$$

where $b_{ij} = a_i^T a_j$ and a_i is the i-th column of $A_{(k)}$. Over the coarse of a sweep, it computes the matrix $B = A^T A$, however, J is not applied directly to $A^T A$, but to A itself, avoiding the numerical instabilities associated with $A^T A$. Algorithm 6 outlines the one-sided Jacobi method.

It skips rotations if $|b_{ij}| < \epsilon\sqrt{b_{ii}b_{jj}}$, indicating that columns a_i and a_j are already numerically orthogonal. It converges when all rotations in a sweep are skipped. Using this formula to check for convergence is required for attaining high relative accuracy [32] (see section 13). The b_{ii} column norms can be cached rather than recomputed for each pair, which reduces operations when rotations are skipped. Note that the last sweep takes about n^3 flops computing b_{ij} terms to check for convergence, without doing any useful work.

A left-handed version can be defined analogously, by applying rotations on the left to orthogonalize the rows of A [84]. This might be preferred if A is a wide matrix stored row-wise, rather than a tall matrix stored column-wise.

One-sided Jacobi can be applied to a rectangular matrix, but again, preprocessing using a QR factorization, and apply Jacobi on the square R matrix, reduces the operation count and preconditions the system for faster convergence; see subsection 12.5.

12.3. Convergence. For the row and column cyclic orderings, Forsythe and Henrici [49] proved that all the Jacobi methods (two-sided eigenvalue, one-sided SVD, and two-sided SVD) converge, provided the rotation angles are bounded below $\pi/2$ by some b,

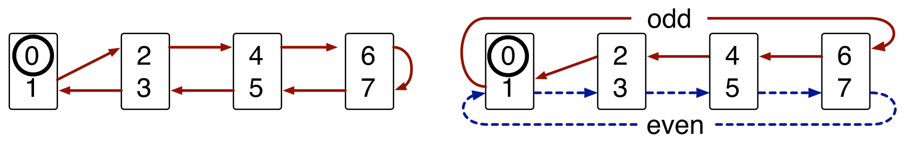
$$|\theta| \le b < \pi/2. \tag{10}$$

For the two-sided eigenvalue and one-sided SVD methods, θ can always be chosen to satisfy (10); see [102]. For the two-sided SVD method, however, this condition may fail to hold. In Forsythe and Henrici's method, the bound is $b = \frac{\pi}{2}$, which may introduce a cycle interchanging two singular values without converging. For the methods of Brent et al. [22], NSVD has a bound $b = 3\pi/4$ and USVD has a bound $b = 5\pi/4$. Proofs for other orderings, particularly for parallel orderings, have been elusive. Despite failing to satisfy the convergence proof's prerequisites, in practice Jacobi methods reliably converge. Using a threshold to skip updates to small entries is a common tactic, especially in the first several sweeps, to accelerate and guarantee convergence [102, 27, 8].

When applied to triangular matrices, Heath et al. [67] and Hari and Veselić [66] observed that applying one sweep of the two-sided SVD method with the row-cyclic ordering (without thresholding) converts an upper triangular matrix to lower triangular, and vice-versa. Hari and Veselić derived rotation angle formulas in the triangular case, and prove that the angles are bounded below $\pi/2$, guaranteeing convergence. Hari and Matejaš [65] later derived more accurate formulas.

Applying column pivoting during the Jacobi iterations can improve convergence. In the one-sided method, de Rijk [27] follows the row-cyclic ordering, but at the start of row i, searches columns $i, \dots, n$ for the column of largest norm and pivots it to column i. Unfortunately, using the row-cyclic ordering makes parallelism difficult. Zhou and Brent [120] likewise show that sorting column norms improves convergence, and give a parallel ordering for sorting.

12.4. Parallel orderings. In two-sided Jacobi, a pair of rotations applied on the left and right affect only two rows and two columns. In one-sided Jacobi, each rotation applied on the right affects only two columns. Therefore, in both cases, $\lfloor n/2 \rfloor$ rotations can be performed in parallel. However, the row and column cyclic orderings are not amenable to parallel computation, as they introduce dependencies between consecutive pairs of elements. Since there are $n(n-1)/2$ pairs to eliminate, an optimal parallel ordering would have $n-1$ steps, with each step eliminating $n/2$



(a) round-robin (b) odd-even

FIG. 27. *Parallel orderings. Rectangles indicating processors are labeled with their assigned columns. Arrows depict movement of columns between Jacobi sweeps. Circled pivot column is stationary.*

pairs in parallel (for n even). Many different parallel Jacobi orderings have been devised. While parallel orderings typically lack a proof of convergence, in practice they work reliably.

Commonly, for parallel implementations of both one-sided and two-sided Jacobi, the matrix is distributed by columns. Early systolic implementations placed two columns [21] or a 2×2 submatrix [22] per processor. Later block implementations placed two block columns [15, 11] or a 2×2 block submatrix [12] per processor. When each processor stores two columns, one-sided Jacobi has the advantage that no communication is required during an update, whereas in two-sided Jacobi, the left transformations (J's) must be broadcast in an all-to-all fashion.

Brent and Luk [21] introduced the round-robin ordering, shown in Figure 27(a), which had previously been known for chess tournaments. After each Jacobi rotation, each node sends and receives two columns, except the pivot node that sends and receives one column. Eberlein [46] gave the odd-even ordering in Figure 27(b). After each odd sweep, the odd permutation (solid red lines) is used; after even sweeps, the even permutation (dashed blue lines) is used. Each node sends and receives one column.

Luk and Park [85] studied the equivalence of orderings, demonstrating that many orderings are equivalent in the sense that relabeling the columns gives identical orderings. For example, choosing a different pivot column in round-robin will give an equivalent ordering. Luk and Park showed that the two main classes of Jacobi orderings are the round-robin and odd-even types. Bečka and Vajteršic [12, 11] compared implementations of the round-robin, odd-even, and a butterfly-like ordering inspired by the Fast Fourier Transform (FFT), on ring, hypercube, and mesh networks for block Jacobi methods.

12.5. Preconditioning. Another means to improving the speed of Jacobi methods is to precondition the matrix to reduce the number of sweeps required for convergence. Drmač and Veselić [44] introduced several forms of preconditioning for the one-sided Jacobi method. The major ideas are outlined below, with a simplified version in Algorithm 7.

First, for a square matrix A, heuristically choose to factor either $X = A$ or $X = A^T$. They give the example of $A = DQ$, where D is diagonal and Q is orthogonal. One-sided Jacobi applied to A implicitly diagonalizes $Q^T D^2 D$, while applied to A^T, it implicitly diagonalizes D^2, which is already diagonal. One heuristic they suggest is to choose the X that maximizes $\left\|\operatorname{diag}(X^T X)\right\|_2$, hence minimizing $\operatorname{off}(X^T X)$. Their second heuristic is to choose the X that minimizes the diagonal entropy of $X^T X$,

Algorithm 7 Preconditioned one-sided Jacobi SVD method (simplified)

```
function preconditioned_one_sided_jacobi( $A$ )
    input: $m \times n$ matrix $A$, $m \geq n$
    output: $U$, $\Sigma$, $V$
    transpose = ($m == n$ and $\eta_d(AA^T) < \eta_d(A^TA)$)    // see (11)
    if transpose then
        $A = A^T$
    end
    $Q_r R P_r^T = A$    // QR factorization with column pivoting
    $L Q_l = R$    // LQ factorization
    $(U_l, \Sigma)$ = one_sided_jacobi_svd($L$)    // Algorithm 6; skip $V$
    $U = Q_r U_l$
    $V = P_r Q_l^T L^{-1} (U_l \Sigma)$  or  $V = P_r R^{-1} (U_l \Sigma)$
    if transpose then
        swap $U \Leftrightarrow V$
    end
end function
```

defined by

$$\eta_d(X^TX) = \eta\left(\operatorname{diag}(X^TX)/\operatorname{trace}(X^TX)\right), \tag{11}$$

where the entropy of a vector p with $p_i \geq 0$, $\sum_i p_i = 1$, is defined as

$$\eta(p) = -\frac{1}{\log n}\sum_{i=1}^{n} p_i \log p_i, \qquad \text{with } 0 \log 0 \equiv 0. \tag{12}$$

Both heuristics are $O(n^2)$.

The second preconditioning technique is to use a QR factorization with column pivoting (QRP) of A, then factor R. This concentrates the mass of the matrix along the diagonal of R^TR, reducing the number of Jacobi sweeps. For a rectangular $m \times n$ problem, $m > n$, this also shrinks it to an $n \times n$ problem, as in subsection 5.4.

Third, use either an LQ factorization of R, or simply let $L = R^T$, then factor L. An LQ factorization further concentrates the mass along the diagonal of L^TL. Using LQ is particularly advantageous in the rank deficient case. For a matrix of rank r, QRP generates $R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$ with the $(n-r) \times (n-r)$ block R_{22} being negligible. Doing an LQ factorization of $\begin{bmatrix} R_{11} & R_{12} \end{bmatrix}$ yields a smaller, $r \times r$, full-rank matrix L. Alternatively, simply using $L = R^T$ is an implicit step of Rutishauser's LR diagonalization applied to R^TR, again concentrating mass along the diagonal of L^TL as compared to R^TR.

Additionally, Drmač and Veselić's error analysis based on using QRP and optionally LQ factorization shows that computing V by solving with either of the triangular matrices L or R is numerically stable and generates an orthogonal matrix; see Algorithm 7 for specifics. This allows us to skip accumulating V during the one-sided Jacobi iteration, removing some Level 1 BLAS operations, and adding Level 3 BLAS operations after the iteration, so we can expect a good performance increase. Their paper gives detailed algorithms that make choices about which preconditioning to use based on condition estimates. Hari [64] and Bečka et al. [9] also applied QRP and LQ preconditioning in the context of parallel one-sided block Jacobi.

In addition to preconditioning, Drmač and Veselić [45] introduced optimizations in the one-sided Jacobi iteration, based on the structure of the preconditioned matrix. In the first sweep, the zero structure of the triangular matrix can be exploited to reduce computation. Second, based on work by Mascarenhas [87], they use a modified row-cyclic strategy to more frequently visit diagonal blocks, since those blocks converge at a slower rate. Heuristically, based on the expectation that $L^T L$ is diagonally dominant, during the first few sweeps, if two rotations in a row are skipped due to thresholding, they skip the rest of the row. This avoids computing dot products when the rotation will likely be skipped. Finally, they use a tiled row-cyclic strategy to improve cache efficiency. All of these improvements combine for a more efficient algorithm.

Okša and Vajteršic [95] showed that the same preconditioning techniques, QRP factorization optionally followed by LQ factorization, also improve convergence for the parallel two-sided block Jacobi method. In their tests, preconditioning concentrated more than 99% of the weight of $\|A\|_F$ into the diagonal blocks. Depending on the singular value distribution, this gave up to an order-of-magnitude reduction in time. This preconditioning was later extended to multiple QR iterations [10].

As noted earlier, two-sided Jacobi preserves the triangular structure when used with an appropriate cyclic ordering. Hari and Matejaš [65, 88, 89] use the QRP and LQ preprocessing to generate triangular matrices. They prove high relative accuracy results for the two-sided Jacobi method on such triangular matrices, and utilize a parallel ordering due to Sameh [103] that preserves the triangular structure.

12.6. Block Jacobi. In section 5, we saw that blocking was a major improvement for SVD methods. Blocking can also be favorably applied to Jacobi methods. Van Loan [112] and Bischof [15] were among the first to describe two-sided block Jacobi SVD methods. The method is very similar to the non-block implementation, with plane rotations J and K operating on two rows or columns now becoming orthogonal block rotations operating on two block rows or block columns. For a block size n_b, let $N = \lceil n/n_b \rceil$ be the number of blocks. The indices i, j now loop over the blocks, $1, \ldots, N$. We reinterpret the notation $\hat{A}$ to be the 2×2 block matrix

$$\hat{A} = \begin{bmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{bmatrix},$$

where each A_{ij} is an $n_b \times n_b$ block. Instead of the 2×2 SVD (8), it computes a 2×2 block SVD,

$$\hat{J}^T \hat{A} \hat{K} = \hat{J}^T \begin{bmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{bmatrix} \hat{K} = \begin{bmatrix} D_{ii} & 0 \\ 0 & D_{jj} \end{bmatrix},$$

either recursively using a serial Jacobi method, or using some other SVD method such as QR iteration. Each processor now holds two block columns. Block row and column updates by the orthogonal matrices J and K are applied as Level 3 BLAS matrix multiplies, greatly enhancing the efficiency of the algorithm.

Bischof [15] investigated two methods to solve the SVD subproblem: using QR iteration or using a single sweep of two-sided Jacobi. In the later case, using only one sweep the block method does not fully annihilate the off-diagonal blocks of the 2×2 block subproblem, and is in fact simply a reorganization of the non-block method, but with updates applied using Level 3 BLAS. He found that using Jacobi to solve the subproblem was faster than using QR iteration; however, this was prior to the fast blocked versions of QR iteration available in LAPACK.

Arbenz and Slapničar [5] gave an early implementation for the one-sided block Jacobi SVD method. Again, the block method is very similar to the non-block method, with the 2×2 eigenvalue problem (9) being replaced with a 2×2 block eigenvalue problem,

$$\hat{J}^T \hat{A} \hat{J} = \hat{J}^T \begin{bmatrix} B_{ii} & B_{ij} \\ B_{ij}^T & B_{jj} \end{bmatrix} \hat{J} = \begin{bmatrix} D_{ii} & 0 \\ 0 & D_{jj} \end{bmatrix},$$

with $B_{ij} = A_i^T A_j$, where A_i as the i-th block column of A. Arbenz and Slapničar used the two-sided Jacobi eigenvalue method to solve the subproblem, which is important for preserving Jacobi's high relative accuracy. Hari [64] derived an optimization using the cosine-sine decomposition as a kind of "fast scaled block-rotation", reducing the flop count up to 40%. Boukaram et al. [20] developed batched one-sided Jacobi and block Jacobi methods for GPUs, to compute SVD factorizations of a batch of small matrices.

Bečka et al. introduced dynamic orderings for the two-sided [8] and one-sided [9] Jacobi methods. Instead of using a cyclic ordering such as row-cyclic, round-robin, or odd-even, the idea is to find the off-diagonal blocks of maximum norm to eliminate. This is Jacobi's original idea, applied on the block level. Using a greedy solution to the *maximum-weight perfect matching problem* takes $O(p^2 \log p)$ time for p processors and yields a set of $N/2$ subproblems of maximum weight to solve in parallel. Their results show significantly improved convergence and time to solution.

12.7. Performance analysis. While Jacobi methods have a long history, even predating bidiagonalization methods, many implementations have been either research codes or for unique systems like the ILLIAC IV [84]. Therefore, we do not have as rich a collection of historical implementations to compare as for bidiagonalization methods. We tested four current implementations of Jacobi methods:

- One-sided Jacobi, available in LAPACK as `dgesvj`, due to Drmač [44].
- Preconditioned one-sided Jacobi, available in LAPACK as `dgejsv`, due to Drmač [44].
- Two-sided Jacobi, available in Eigen 3.3.3 [47].
- Preconditioned one-sided block Jacobi, due to Bečka et al. [9].

Jacobi has traditionally trailed bidiagonalization methods in performance for two reasons. First, a comparison of flops in Figure 28 shows that for computing singular values only (no vectors), Jacobi cannot finish even one sweep in the same flops as bidiagonalization ($\frac{8}{3}n^3$). When computing vectors, Jacobi would need to complete in two sweeps to have fewer flops than QR iteration, and one sweep to have fewer flops than divide and conquer. However, with optimizations to skip rotations and take advantage of matrix structure [45, 89], these Jacobi flop counts are significant overestimates.

However, as we have repeatedly seen, flops are now a poor metric for performance. It matters whether flops are in compute-intensive Level 3 BLAS or not. For Jacobi, dot products and plane rotations are Level 1 BLAS, so are memory bandwidth limited. For preconditioned Jacobi, QR with column pivoting (QRP) has a mixture of Level 2 and Level 3 BLAS operations, similar to the traditional one-stage bidiagonalization discussed in subsection 5.1, so its performance is also limited by memory bandwidth. The triangular solve for V and multiplying QU will both be Level 3 BLAS operations. The level of parallelism also matters. The two LAPACK implementations, one-sided

	no vectors	with vectors
QR iteration	$\frac{8}{3}n^3$	$\frac{52}{3}n^3 \approx 17n^3$
divide and conquer	$\frac{8}{3}n^3$	$\frac{28}{3}n^3 \approx 9n^3$
one-sided Jacobi	$5Sn^3$	$7Sn^3$
two-sided Jacobi	$4Sn^3$	$8Sn^3$
preconditioned one-sided Jacobi	$5Sn^3 + \frac{8}{3}n^3$	$5Sn^3 + \frac{17}{3}n^3$
preconditioned two-sided Jacobi	$4Sn^3 + \frac{8}{3}n^3$	$6Sn^3 + \frac{17}{3}n^3$

FIG. 28. *Floating point operation counts for square $n \times n$ matrix and S Jacobi sweeps. For Jacobi, fast Givens rotations [63] are assumed. For preconditioned Jacobi, initial QRP and LQ factorizations and triangular solve for V are also assumed.*

Jacobi and preconditioned one-sided Jacobi, do not use explicit parallelism. Therefore, the only parallelism is within the BLAS, which is very limited for Level 1 BLAS. In contrast, the block Jacobi method uses Level 3 BLAS operations and explicit parallelism via MPI, so we can expect much better performance.

In Figure 29(a), for square matrices without vectors, both one-sided Jacobi methods were about half EISPACK's speed, while with vectors in Figure 29(b), preconditioned Jacobi is 2× faster than plain Jacobi, and close to EISPACK's speed. For tall, 3:1 matrices in Figure 29(c), the plain one-sided Jacobi does not do an initial QR factorization, so it remains about half of EISPACK's speed, while the preconditioned Jacobi improves to about 2× EISPACK's speed. When computing vectors in Figure 29(d), the preconditioned Jacobi version gains even more, being over 3× faster than EISPACK.

For the very tall-skinny 1000:1 case in Figures 29(e) and 29(f), the time with preconditioned Jacobi is dominated by QRP, which uses more Level 2 and 3 BLAS operations, so the performance improves to over 100× EISPACK. LAPACK's QR iteration uses a regular QR factorization (no pivoting), which is predominantly Level 3 BLAS, so its performance is significantly faster than Jacobi. However, QRP will generate a more accurate factorization than regular QR, especially if A is ill-conditioned.

In most cases, the Jacobi single-core performance was identical to its multi-core performance, indicating that the Level 1 BLAS routines do not have appreciable parallelism. For tall matrices, preconditioning gained an advantage when using multiple cores, shown by the difference between the solid and dashed green lines in Figures 29(c) to 29(f), due to parallelism within QRP, solving for V, and computing QU.

In all of these results, the two-sided Jacobi implementation available in Eigen was considerably slower. This can partly be explained because it has to update the matrix both row-wise and column-wise, making for poor cache performance. For square matrices, it does not do any preconditioning. For tall matrices, it uses QRP, which improves its relative performance somewhat. (Note that Eigen can be configured to instead call LAPACK's QR iteration method.)

Figure 30 shows results for the precondition one-sided block Jacobi method. We tested two variants of the preconditioning, one using QR factorization with column pivoting (QRP), the other using regular QR factorization (no pivoting). In both cases, this was followed by an LQ factorization. This implementation has explicit parallelism via MPI. It uses ScaLAPACK for the QRP, QR, and LQ factorizations. We see that

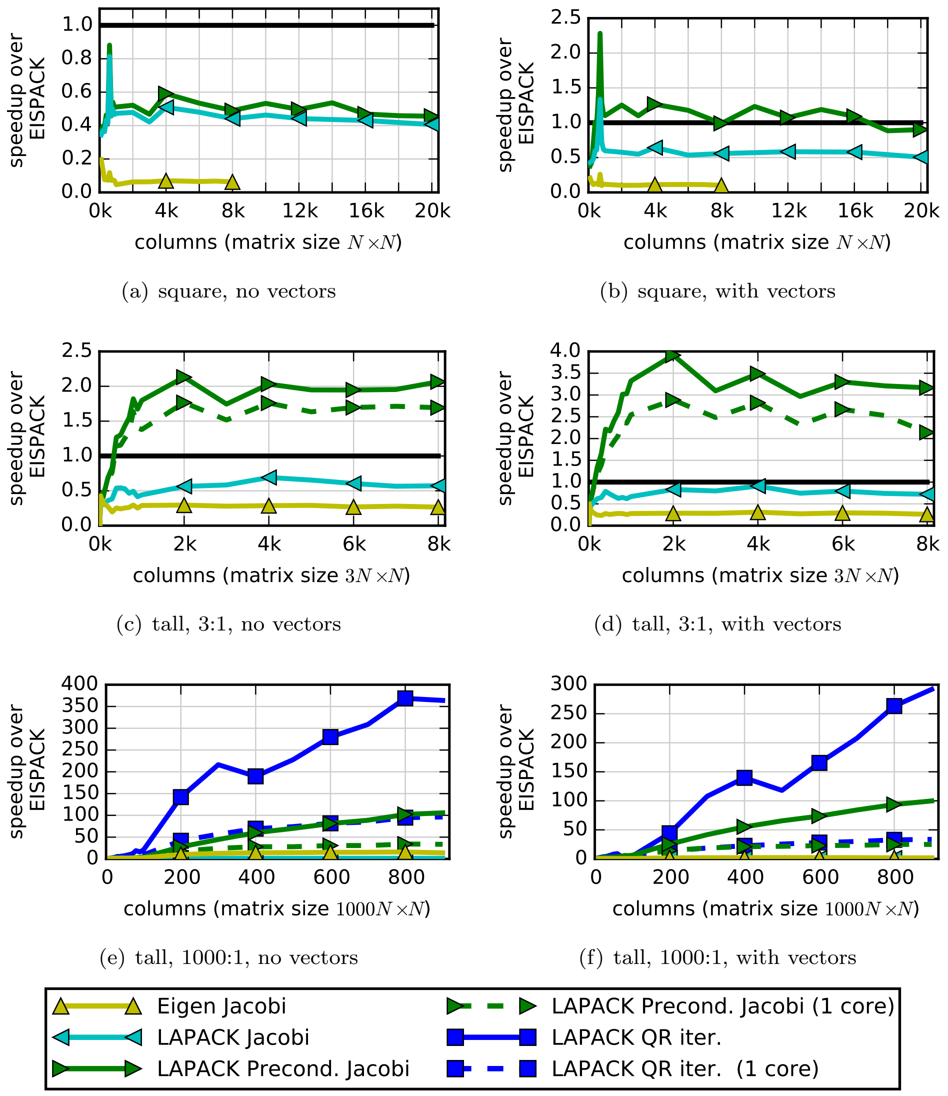


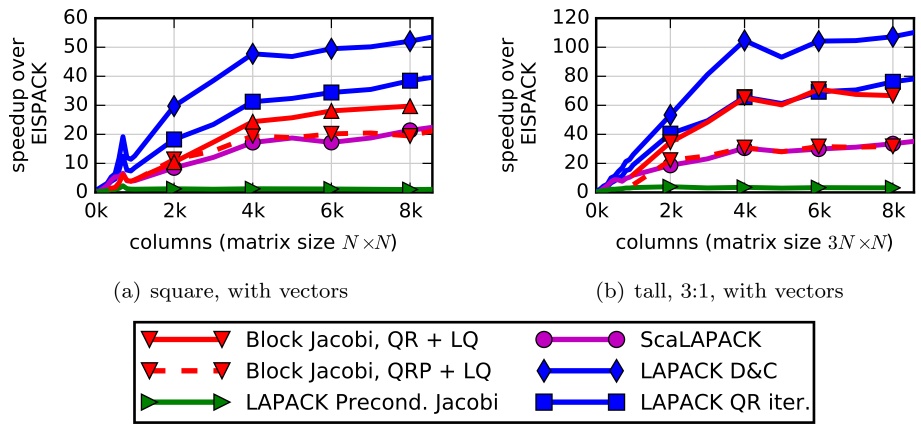
FIG. 29. *Comparison of LAPACK's one-sided Jacobi, preconditioned one-sided Jacobi, and Eigen's two-sided Jacobi.*

with QRP + LQ, it performed similarly to ScaLAPACK QR iteration, while with QR + LQ, it was a bit faster, matching LAPACK's QR iteration in performance for the tall, 3:1 case.

13. Accuracy.

While Jacobi methods have struggled to compete with the performance of bidiagonalization methods, for some classes of matrices they have a distinct advantage in accuracy, which is now their main motivation. In this section, we briefly explore the accuracy differences between methods. The traditional perturbation theory [32] for both bidiagonalization and Jacobi methods shows that

$$\frac{|\sigma_i - \hat{\sigma}_i|}{\sigma_i} \leq O(\epsilon)\kappa(A),$$



(a) square, with vectors (b) tall, 3:1, with vectors

FIG. 30. *Comparison of preconditioned one-side block Jacobi, LAPACK's preconditioned one-sided Jacobi, QR iteration, and divide and conquer.*

where σ_i and $\hat{\sigma}_i$ are the singular values of A and $A + \delta A$, respectively, with a small perturbation δA such that $\|\delta A\|_2 \leq O(\epsilon)\,\|A\|_2$, and $\kappa(A)$ is the condition number of A. This implies that large singular values are computed accurately, but small singular values may be totally inaccurate if $\kappa(A)$ is large. For the one-sided Jacobi SVD method, this bound can be improved. Specifically, on matrices of the form $A = CD$, where C has columns with unit two-norm and D is diagonal, Demmel and Veselić [32] proved the bound

$$\frac{|\sigma_i - \hat{\sigma}_i|}{\sigma_i} \leq O(\epsilon)\kappa(C). \tag{13}$$

Crucially, it can be that $\kappa(C) \ll \kappa(A)$, particularly in the instance of a *strongly scaled* matrix where D is ill-conditioned. If ill-conditioning is artificial, due to poor scaling, then one-sided Jacobi will be unaffected by it and will compute even small singular values to high relative accuracy. Demmel et al. [30] extended methods of computing the SVD with high relative accuracy to a wider class of matrices of the form $A = XDY^T$, where D is diagonal, and X and Y are well-conditioned.

Similar results apply for the two-sided Jacobi eigenvalue method with a positive definite matrix $A = D^T BD$ [32]. For eigenvalues of an indefinite matrix, though, QR iteration may be more accurate than Jacobi [108].

When applied to triangular matrices, Matejaš and Hari [88, 89] proved that the two-sided Jacobi SVD method also attains high relative accuracy. One can preprocess a general matrix using QRP to obtain such a triangular matrix.

Applied to a bidiagonal matrix, the implicit zero-shift variant of QR iteration and the bisection method have been shown to achieve high relative accuracy for all singular values [31]. However, the classical reduction from dense to bidiagonal perturbs the singular values so the exact singular values of the bidiagonal matrix no longer have high relative accuracy for the original matrix A. Hence, *any method* based on an initial reduction to bidiagonal will lose relative accuracy for small singular values of an ill-conditioned matrix. To address this deficiency, Barlow [7] developed a more accurate bidiagonalization, using QRP followed by a Givens rotation based bidiagonalization. Recently, Drmač [43] demonstrated that preprocessing a matrix with QRP (LAPACK's `dgeqp3` routine) is sufficient to make a subsequent QR iteration or

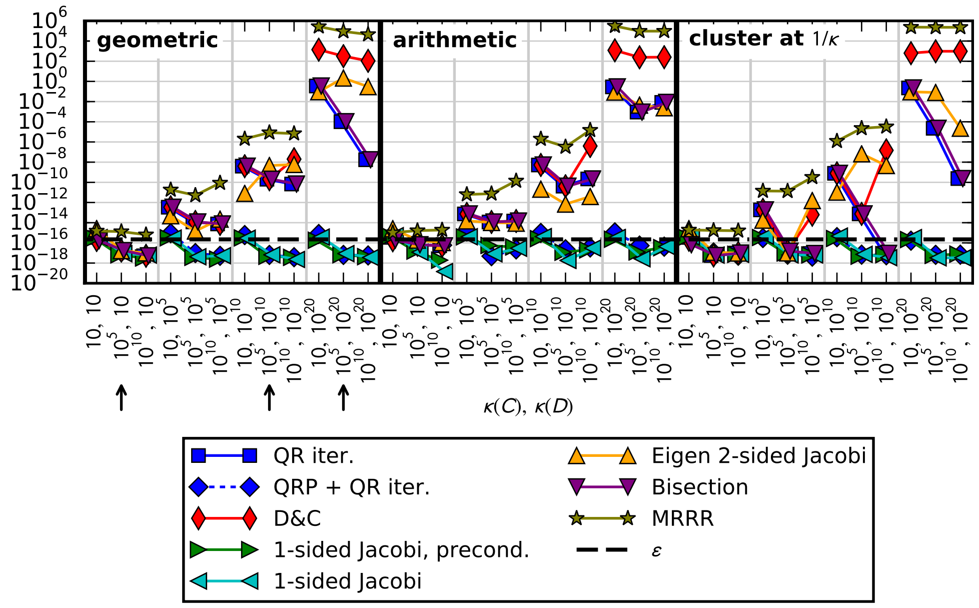


FIG. 31. *Maximum relative error in singular values,* $\max|\hat{\sigma}_i - \sigma_i| / (\kappa(C)\sigma_i)$*, for* $i = 1, \ldots, 100$*, with various test matrices. Figure 32 shows details for three instances indicated by arrows: geometric distribution with* $(\kappa(C),\ \kappa(D)) = (10^5, 10)$*;* $(10^5, 10^{10})$*;* $(10^5, 10^{20})$.

bisection have high relative accuracy. (But not divide and conquer, which is not as accurate as QR iteration.)

Here we test the accuracy of various methods on matrices with three different distributions of singular values: arithmetic, geometric, and a cluster at $1/\kappa(C)$, as described in section 2. For each distribution, we generate singular values Σ with condition number $\kappa(C)$, scale them so that $\sum \sigma_i^2 = n$, and set $\tilde{C} = U\Sigma V^T$ where U and V are random orthogonal matrices from the Haar distribution [106]. To satisfy the conditions of (13), we use the method by Davies and Higham [26] to make $C = \tilde{C}W$ with columns of unit two-norm, where W is orthogonal. Finally, we set $A = CD$, where D is diagonal with entries whose logarithms are random uniform on $(\log(1/\kappa(D)), \log(1))$. For each distribution, we set $n = 100$ and vary $\kappa(C) \in \{10, 10^5, 10^{10}\}$ and $\kappa(D) \in \{10, 10^5, 10^{10}, 10^{20}\}$. For a reference solution, we used MATLAB's [90] variable-precision arithmetic (vpa) with 64 digits.

Figure 31 demonstrates the significant difference between one-sided Jacobi methods and bidiagonalization methods (QR iteration, D&C, bisection, MRRR). Both one-sided Jacobi methods achieve high relative accuracy for all singular values, at or below the dashed line representing machine ϵ. For small scaling, with $\kappa(D) = 10$, all methods achieve high accuracy on all the matrices. Most of the bisection methods show increased relative errors as the scaling $\kappa(D)$ grows. For $\kappa(D) = 10^{20}$, the maximum errors were sometimes larger than 1, i.e., no correct digits in the smallest singular values. Among bisection methods, the exception was preprocessing using QR with column pivoting, then using QR iteration (QRP + QR iter., blue diamonds), which also achieved high relative accuracy, as predicted by Drmač.

QR iteration (blue squares) and bisection (purple down triangles) produce extremely similar errors, demonstrating that they both accurately compute singular values of the bidiagonal matrix, and the error occurs in the reduction to bidiagonal. Once the condition number $\kappa(A)$ exceeds $1/\epsilon$, divide and conquer (red diamonds) has much worse error than QR iteration. Even with modest scaling, MRRR (stars) has the worst error. Eigen's two-sided Jacobi (orange up triangles) also exhibits significant error as the scaling increases. Preprocessing with QRP before Eigen's two-sided Jacobi (not shown) improved the accuracy, but not to the high relative accuracy of one-sided Jacobi. Based on [89], other two-sided Jacobi implementations are expected to achieve high relative accuracy.

To explain these results in more detail, we look at three specific cases for the geometric distribution with $\kappa(C) = 10^5$ and $\kappa(D) \in \{10, 10^{10}, 10^{20}\}$. In Figure 32, the left column shows the actual singular values, in both log and linear scale, while the right column shows the relative error in each singular value, σ_i from $i = 1, \dots, 100$. In the top row, with minimal scaling ($\kappa(D) = 10$), all the methods achieve high accuracy, below ϵ in almost all cases. Eigen has a little higher error for large singular values, and MRRR is a little higher for small singular values.

In the middle row, with modest scaling ($\kappa(D) = 10^{10}$), the one-sided Jacobi methods and QRP + QR iteration maintain high relative accuracy for all singular values. The bidiagonalization methods have high accuracy for the large singular values (near σ_1), but the relative error increases for small singular values, losing digits of accuracy. Eigen's error also increases.

In the bottom row, with large scaling ($\kappa(D) = 10^{20}$), the error of bidiagonalization methods for small singular values grows even more. As seen in the bottom-left graph, several methods compute singular values that noticeably diverge from the reference solution. For this matrix with $\sigma_{\max} \approx 10^{20}$, MRRR declares all $\sigma_i < 10^7$ to be 3.27×10^7, i.e., it cannot resolve smaller singular values. Similarly for D&C, all $\sigma_i < 10^3$ are computed as 6.91×10^3. Eigen also has issues for $\sigma < 10^4$, though it does not flatline as MRRR and D&C do. QR iteration and bisection follow the true singular values much more closely, but still exhibit significant error for small singular values.

14. Additional test cases.

So far, we have mostly considered the performance of random uniform matrices. In this section, we look briefly at additional test cases using various distributions of singular values. Our purpose here is to give the reader an idea of the variability in performance and how representative the random uniform tests are. The distribution of singular values affects the performance of various SVD algorithms differently. For QR iteration and divide and conquer, whenever a singular value is determined with sufficient accuracy, it can be removed to shrink the problem size, a process known as *deflation*, improving the performance. For MRRR, having singular values close to one another will cause it to recurse further in the representation tree, decreasing its performance [119]. For one-sided Jacobi, matrices that are close to orthogonal—i.e., most of the weight of A^TA is near the diagonal—converge faster.

Figure 33 shows results for six methods on various matrices. These all use the LAPACK implementations, except MRRR which uses a modification of the bisection `dgesvdx` code, as described in section 9. Note that the y-axis scale is different for QR iteration, divide and conquer, and MRRR than for Jacobi and bisection. See section 2 for a description of the matrix types. For each algorithm, the first, blue bar is for a random uniform matrix, matching most of the results elsewhere in this paper. The geometric and log-random distributions themselves are similar, so in most cases their performance trends are similar, except when using bisection. We see that for

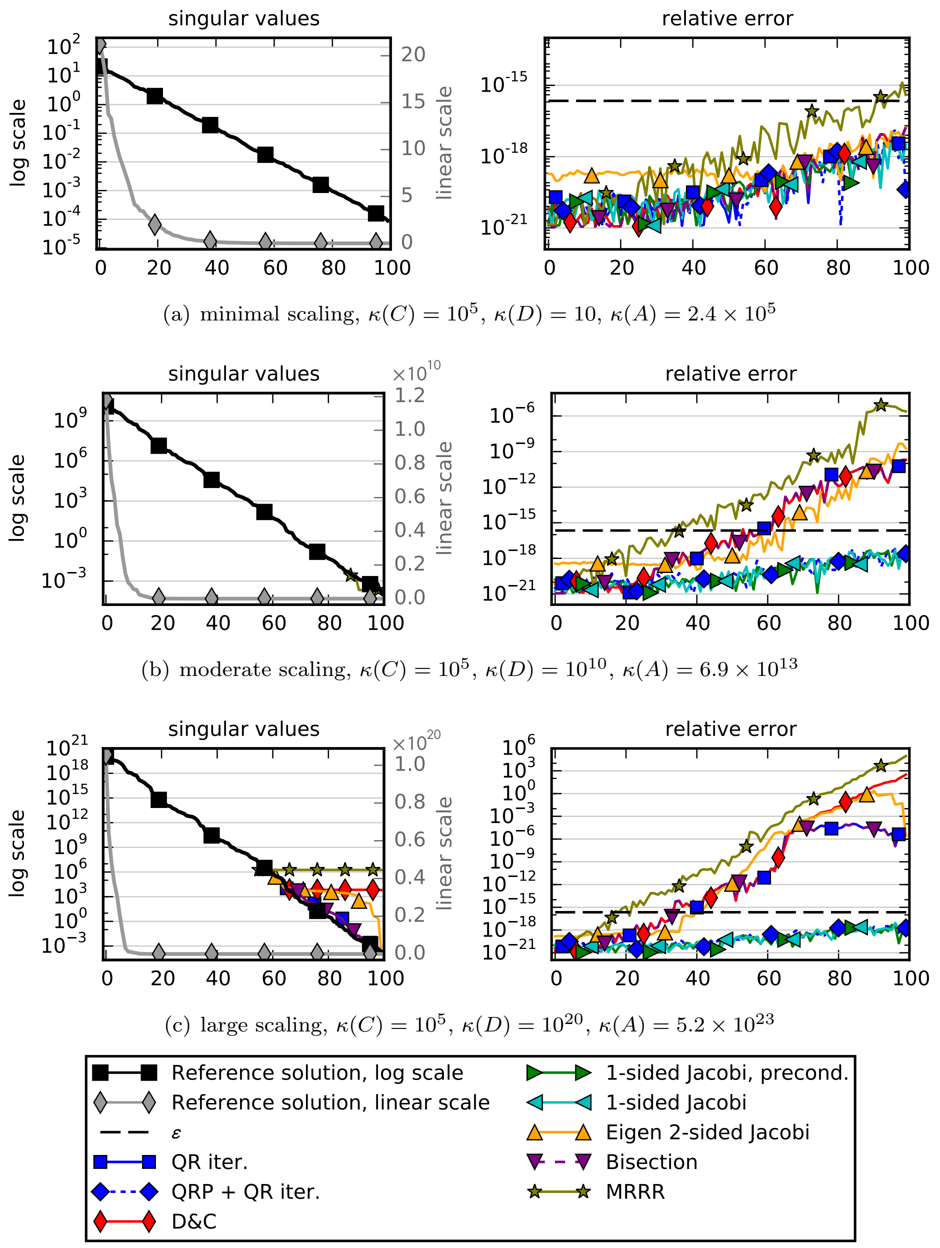


FIG. 32. *Singular values of $A = CD$ are plotted twice in left column, once in log scale (black squares), once in linear scale (gray diamonds). In most cases, computed singular values are visually coincident with reference solution (log scale). Right column shows relative error in each singular value, $|\hat{\sigma}_i - \sigma_i| / (\kappa(C)\sigma_i)$. x axis indexes the singular values from largest to smallest, $i = 1, \ldots, 100$.*

QR iteration, the performance for most matrices is similar to that of random uniform, with a few being up to 18% slower. For divide and conquer, the arithmetic distribution (cyan) was up to 24% slower, while log-random (green) was up to 23% faster than random uniform. The two clusters (red, orange) were 77% and 60% faster, due to significant deflation. MRRR is more variable, with geometric (purple) and log-random

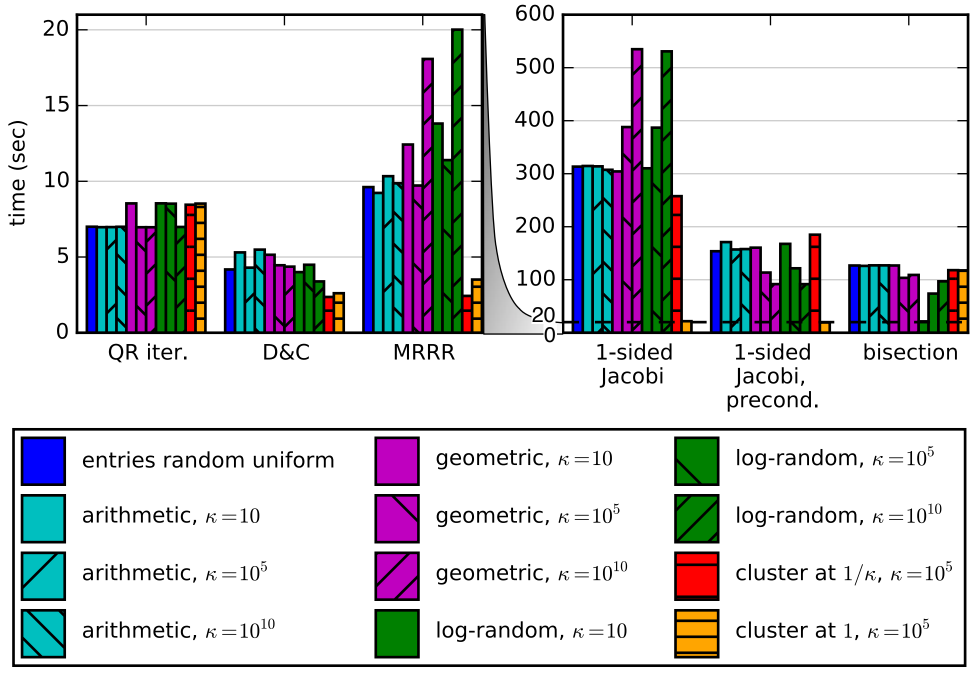


FIG. 33. *Time to compute full SVD for $n = 3000$. Note change in y axis; dashed line at $t = 20$ corresponds to y axis in left plot.*

(green) being up to 47% and 52% slower on ill-conditioned matrices ($\kappa = 10^{10}$), while both clusters of repeated singular values were up to 3× faster than random uniform. Arithmetic (cyan) was not significantly affected by conditioning.

Because one-sided Jacobi and bisection were significantly slower, they are plotted with a different y axis. In all cases, one-sided Jacobi and bisection were slower than QR iteration, divide and conquer, and MRRR. The geometric (purple) and log-random (green) matrices exhibited opposite behavior for the two Jacobi methods: for plain Jacobi, both matrices became slower as the conditioning worsened, while for preconditioned Jacobi, both became faster. A cluster at $1/\kappa$ took similar time to random. A cluster at 1 was much faster, because $A^T A$ is already nearly diagonal, but preconditioning did not further improve it. Bisection was surprisingly 4.9× faster for a well-conditioned ($\kappa = 10$) log-random matrix, but the speedup decreased for poorer conditioning. As we saw earlier in section 8 when computing a subset of vectors, clusters were not advantageous.

While the performance does vary for different classes of matrices—sometimes substantially—at a high level, our performance conclusions remain valid: divide and conquer is the fastest (being tied with MRRR in one case), then QR iteration, then MRRR. One-sided Jacobi is the slowest method, with preconditioning generally improving its speed, often by a factor of 2× or more. For computing all vectors, bisection is also slow; its main advantage is in computing a subset of vectors, as previously shown in section 8.

15. Conclusions.

As we have seen, algorithms to compute the SVD have continually evolved to address changes in computer hardware design, as well as advancements in mathematics. Early implementations such as EISPACK demonstrated that computing the SVD stably was feasible. Later implementations focused on improving the performance, first by using Level 1 BLAS for vector computers, then by reformulating the algorithm for Level 3 BLAS to address the emergence of cache-based memory hierarchies. More recently, a two-stage algorithm shifted even more operations from Level 2 to Level 3 BLAS. These changes have addressed the growing gap between memory bandwidth and computational speed, as well as enabled greater use of parallel hardware. Implementations have also taken advantage of different architectures such as distributed memory computers and accelerators. Mathematical advancements have been important in reducing the number of operations performed. For tall-skinny problems, using an initial QR factorization can eliminate a quarter to half of the operations. For square matrices, the divide and conquer algorithm reduces operations by nearly half. For Jacobi methods, preconditioning has been vital to improve convergence, while at the same time making computation of singular vectors more efficient. Block Jacobi methods with dynamic selection of subproblems have become competitive with some bidiagonalization methods. Improvements in algorithms used to preprocess a matrix, such as using a CAQR factorization [29] for tall-skinny matrices, or future improvements to QRP methods, are immediately applicable to benefit SVD performance.

As we build the next generation of linear algebra software targeting exascale computers [77], the goal is to integrate these techniques—such as the two-stage reduction to bidiagonal, accelerators, and distributed computing—into a scalable SVD solver. While the techniques have been demonstrated to work, the challenge is being able to hide communication latencies in large distributed machines. Bottlenecks due to Amdahl's law, such as solving the bidiagonal SVD, will also be crucial to resolve. Improving algorithms to remove communication and memory bandwidth limitations becomes critically important.

For certain classes of matrices that are strong scaled, classical methods based on reduction to bidiagonal will not accurately compute small singular values. In these cases, one should turn to Jacobi methods or preprocessing the matrix using QR factorization with column pivoting (QRP) to attain high relative accuracy.

We have focused on solving dense systems. There are of course different techniques for solving SVD problems with sparse linear systems. Also, if one is concerned with only an approximate, low-rank solution, then using a randomized SVD algorithm [99] may be another avenue to pursue. This is often the case for huge systems arising from big data problems.

Here we have compared implementations on a common, modern architecture. To give some historical perspective, in 1977, EISPACK took 0.79 seconds (1.7 Mflop/s) to compute singular values for $n = 80$ on an IBM 370/195 [105]. Today, the same EISPACK code achieves 0.74 Gflop/s on large problems, yielding over two orders-of-magnitude advancement in single core hardware speed. On top of this, we have shown an additional two orders-of-magnitude improvement going from EISPACK to PLASMA (146 Gflop/s) on a multicore architecture, and four orders of magnitude to DPLASMA (6.8 Tflop/s) on a distributed-memory machine—while moving from solving systems of dimension 100 to over 100,000—yielding over six orders-of-magnitude performance improvement in 40 years.

Acknowledgments. We thank Martin Bečka, Gabriel Okša, and Marián Vajteršic for use of their block Jacobi code; Osni Marques for assistance with the MRRR code; and the anonymous reviewers for feedback to improve the quality and scope of this work.

References.

[1] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra, *Comparative study of one-sided factorizations with multiple software packages on multi-core hardware*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09), ACM, 2009, p. 20, https://doi.org/10.1145/1654059.1654080.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK users' guide*, SIAM, Philadelphia, third ed., 1999, https://doi.org/10.1137/1.9780898719604.

[3] H. Andrews and C. Patterson, *Singular value decomposition (SVD) image coding*, IEEE Transactions on Communications, 24 (1976), pp. 425–432, https://doi.org/10.1109/TCOM.1976.1093309.

[4] P. Arbenz and G. Golub, *On the spectral decomposition of Hermitian matrices modified by low rank perturbations with applications*, SIAM Journal on Matrix Analysis and Applications, 9 (1988), pp. 40–58, https://doi.org/10.1137/0609004.

[5] P. Arbenz and I. Slapničar, *An analysis of parallel implementations of the block-Jacobi algorithm for computing the SVD*, in Proceedings of the 17th International Conference on Information Technology Interfaces ITI, vol. 95, 1995, pp. 13–16, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.4595.

[6] G. Ballard, J. Demmel, and N. Knight, *Avoiding communication in successive band reduction*, ACM Transactions on Parallel Computing, 1 (2015), p. 11, https://doi.org/10.1145/2686877.

[7] J. L. Barlow, *More accurate bidiagonal reduction for computing the singular value decomposition*, SIAM Journal on Matrix Analysis and Applications, 23 (2002), pp. 761–798, https://doi.org/10.1137/S0895479898343541.

[8] M. Bečka, G. Okša, and M. Vajteršic, *Dynamic ordering for a parallel block-Jacobi SVD algorithm*, Parallel Computing, 28 (2002), pp. 243–262, https://doi.org/10.1016/S0167-8191(01)00138-7.

[9] M. Bečka, G. Okša, and M. Vajteršic, *New dynamic orderings for the parallel one–sided block-Jacobi SVD algorithm*, Parallel Processing Letters, 25 (2015), p. 1550003, https://doi.org/10.1142/S0129626415500036.

[10] M. Bečka, G. Okša, M. Vajteršic, and L. Grigori, *On iterative QR preprocessing in the parallel block-Jacobi SVD algorithm*, Parallel Computing, 36 (2010), pp. 297–307, https://doi.org/10.1016/j.parco.2009.12.013.

[11] M. Bečka and M. Vajteršic, *Block-Jacobi SVD algorithms for distributed memory systems I: hypercubes and rings*, Parallel Algorithms and Application, 13 (1999), pp. 265–287, https://doi.org/10.1080/10637199808947377.

[12] M. Bečka and M. Vajteršic, *Block-Jacobi SVD algorithms for distributed memory systems II: meshes*, Parallel Algorithms and Application, 14 (1999), pp. 37–56, https://doi.org/10.1080/10637199808947370.

[13] C. Bischof, B. Lang, and X. Sun, *Algorithm 807: The SBR Toolbox—software for successive band reduction*, ACM Transactions on Mathematical Software (TOMS), 26 (2000), pp. 602–616, https://doi.org/10.1145/365723.365736.

[14] C. Bischof and C. Van Loan, *The WY representation for products of Householder matrices*, SIAM Journal on Scientific and Statistical Computing, 8 (1987), pp. 2–13, https://doi.org/10.1137/0908009.

[15] C. H. Bischof, *Computing the singular value decomposition on a distributed*

system of vector processors, Parallel Computing, 11 (1989), pp. 171–186, https://doi.org/10.1016/0167-8191(89)90027-6.

[16] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, et al., *ScaLAPACK users' guide*, SIAM, Philadelphia, 1997, https://doi.org/10.1137/1.9780898719642.

[17] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al., *An updated set of basic linear algebra subprograms (BLAS)*, ACM Transactions on Mathematical Software (TOMS), 28 (2002), pp. 135–151, https://doi.org/10.1145/567806.567807.

[18] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, et al., *Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA*, in 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), IEEE, 2011, pp. 1432–1441, https://doi.org/10.1109/IPDPS.2011.299.

[19] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, *DAGuE: A generic distributed DAG engine for high performance computing*, Parallel Computing, 38 (2012), pp. 37–51, https://doi.org/10.1016/j.parco.2011.10.003.

[20] W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes, *Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression*, Parallel Computing, (2017), https://doi.org/10.1016/j.parco.2017.09.001.

[21] R. P. Brent and F. T. Luk, *The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays*, SIAM Journal on Scientific and Statistical Computing, 6 (1985), pp. 69–84, https://doi.org/10.1137/0906007.

[22] R. P. Brent, F. T. Luk, and C. van Loan, *Computation of the singular value decomposition using mesh-connected processors*, Journal of VLSI and computer systems, 1 (1985), pp. 242–270, http://maths-people.anu.edu.au/~brent/pd/rpb080i.pdf.

[23] T. F. Chan, *An improved algorithm for computing the singular value decomposition*, ACM Transactions on Mathematical Software (TOMS), 8 (1982), pp. 72–83, https://doi.org/10.1145/355984.355990.

[24] J. Choi, J. Dongarra, and D. W. Walker, *The design of a parallel dense linear algebra software library: reduction to Hessenberg, tridiagonal, and bidiagonal form*, Numerical Algorithms, 10 (1995), pp. 379–399, https://doi.org/10.1007/BF02140776.

[25] J. J. M. Cuppen, *A divide and conquer method for the symmetric tridiagonal eigenproblem*, Numerische Mathematik, 36 (1980), pp. 177–195, https://doi.org/10.1007/BF01396757.

[26] P. I. Davies and N. J. Higham, *Numerically stable generation of correlation matrices and their factors*, BIT Numerical Mathematics, 40 (2000), pp. 640–651, https://doi.org/10.1023/A:102238421.

[27] P. P. M. de Rijk, *A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer*, SIAM Journal on Scientific and Statistical Computing, 10 (1989), pp. 359–371, https://doi.org/10.1137/0910023.

[28] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, *Indexing by latent semantic analysis*, Journal of the Ameri-

can society for information science, 41 (1990), p. 391, https://doi.org/10.1002/(SICI)1097-4571(199009)41:6⟨391::AID-ASI1⟩3.0.CO;2-9.

[29] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM Journal of Scientific Computing, 34 (2012), pp. A206–A239, https://doi.org/10.1137/080731992.

[30] J. Demmel, M. Gu, S. Eisenstat, I. Slapničar, K. Veselić, and Z. Drmač, *Computing the singular value decomposition with high relative accuracy*, Linear Algebra and its Applications, 299 (1999), pp. 21–80, https://doi.org/10.1016/S0024-3795(99)00134-2.

[31] J. Demmel and W. Kahan, *Accurate singular values of bidiagonal matrices*, SIAM Journal on Scientific and Statistical Computing, 11 (1990), pp. 873–912, https://doi.org/10.1137/0911052.

[32] J. Demmel and K. Veseli'c, *Jacobi's method is more accurate than QR*, SIAM Journal on Matrix Analysis and Applications, 13 (1992), pp. 1204–1245, https://doi.org/10.1137/0613074.

[33] J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM, Philadelphia, 1997, https://doi.org/10.1137/1.9781611971446.

[34] J. W. Demmel, I. Dhillon, and H. Ren, *On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic*, Electronic Transactions on Numerical Analysis, 3 (1995), pp. 116–149, http://emis.ams.org/journals/ETNA/vol.3.1995/pp116-149.dir/pp116-149.pdf.

[35] I. S. Dhillon, *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*, PhD thesis, EECS Department, University of California, Berkeley, 1997, http://www.dtic.mil/docs/citations/ADA637073.

[36] I. S. Dhillon and B. N. Parlett, *Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices*, Linear Algebra and its Applications, 387 (2004), pp. 1–28, https://doi.org/10.1016/j.laa.2003.12.028.

[37] I. S. Dhillon and B. N. Parlett, *Orthogonal eigenvectors and relative gaps*, SIAM Journal on Matrix Analysis and Applications, 25 (2004), pp. 858–899, https://doi.org/10.1137/S0895479800370111.

[38] I. S. Dhillon, B. N. Parlett, and C. Vömel, *The design and implementation of the MRRR algorithm*, ACM Transactions on Mathematical Software (TOMS), 32 (2006), pp. 533–560, https://doi.org/10.1145/1186785.1186788.

[39] J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK users' guide*, SIAM, Philadelphia, 1979, https://doi.org/10.1137/1.9781611971811.

[40] J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Transactions on Mathematical Software (TOMS), 16 (1990), pp. 1–17, https://doi.org/10.1145/77626.79170.

[41] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, *An extended set of FORTRAN basic linear algebra subprograms*, ACM Transactions on Mathematical Software (TOMS), 14 (1988), pp. 1–17, https://doi.org/10.1145/42288.42291.

[42] J. Dongarra, D. C. Sorensen, and S. J. Hammarling, *Block reduction of matrices to condensed forms for eigenvalue computations*, Journal of Computational and Applied Mathematics, 27 (1989), pp. 215–227, https://doi.org/10.1016/0377-0427(89)90367-1. Special Issue on Parallel Algorithms for Numerical Linear Algebra.

[43] Z. Drmač, *Algorithm 977: A QR-preconditioned QR SVD method for comput-*

ing the SVD with high accuracy, ACM Transactions on Mathematical Software (TOMS), 44 (2017), p. 11, https://doi.org/10.1145/3061709.

[44] Z. Drmač and K. Veselić, *New fast and accurate Jacobi SVD algorithm, I*, SIAM Journal on Matrix Analysis and Applications, 29 (2008), pp. 1322–1342, https://doi.org/10.1137/050639193.

[45] Z. Drmač and K. Veseli'c, *New fast and accurate Jacobi SVD algorithm, II*, SIAM Journal on Matrix Analysis and Applications, 29 (2008), pp. 1343–1362, https://doi.org/10.1137/05063920X.

[46] P. Eberlein, *On one-sided Jacobi methods for parallel computation*, SIAM Journal on Algebraic Discrete Methods, 8 (1987), pp. 790–796, https://doi.org/10.1137/0608064.

[47] Eigen, *Eigen 3.3.3*, 2017, http://eigen.tuxfamily.org/.

[48] K. V. Fernando and B. N. Parlett, *Accurate singular values and differential qd algorithms*, Numerische Mathematik, 67 (1994), pp. 191–229, https://doi.org/10.1007/s002110050024.

[49] G. E. Forsythe and P. Henrici, *The cyclic Jacobi method for computing the principal values of a complex matrix*, Transactions of the American Mathematical Society, 94 (1960), pp. 1–23, https://doi.org/10.2307/1993275.

[50] B. S. Garbow, J. M. Boyle, C. B. Moler, and J. Dongarra, *Matrix eigensystem routines – EISPACK guide extension*, vol. 51 of Lecture Notes in Computer Science, Springer, Berlin, 1977, https://doi.org/10.1007/3-540-08254-9.

[51] M. Gates, S. Tomov, and J. Dongarraa, *Accelerating the SVD two stage bidiagonal reduction and divide and conquer using GPUs*, Parallel Computing, 74 (2018), pp. 3–18, https://doi.org/10.1016/j.parco.2017.10.004.

[52] G. Golub, *Some modified matrix eigenvalue problems*, SIAM Review, 15 (1973), pp. 318–334, https://doi.org/10.1137/1015032.

[53] G. Golub and W. Kahan, *Calculating the singular values and pseudo-inverse of a matrix*, SIAM Journal on Numerical Analysis (Series B), 2 (1965), pp. 205–224, https://doi.org/10.1137/0702016.

[54] G. Golub and C. Reinsch, *Singular value decomposition and least squares solutions*, Numerische Mathematik, 14 (1970), pp. 403–420, https://doi.org/10.1007/BF02163027.

[55] B. Grosser and B. Lang, *Efficient parallel reduction to bidiagonal form*, Parallel Computing, 25 (1999), pp. 969–986, https://doi.org/10.1016/S0167-8191(99)00041-1.

[56] M. Gu, J. Demmel, and I. Dhillon, *Efficient computation of the singular value decomposition with applications to least squares problems*, Tech. Report LBL-36201, Lawrence Berkeley Laboratory, September 29 1994, http://www.cs.utexas.edu/users/inderjit/public_papers/least_squares.pdf.

[57] M. Gu and S. C. Eisenstat, *A divide and conquer algorithm for the bidiagonal SVD*, Tech. Report YALEU/DCS/TR-933, Department of Computer Science, Yale University, November 1992, http://cpsc.yale.edu/research/technical-reports/1992-technical-reports.

[58] M. Gu and S. C. Eisenstat, *A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem*, SIAM Journal on Matrix Analysis and Applications, 15 (1994), pp. 1266–1276, https://doi.org/10.1137/S089547989223924X.

[59] M. Gu and S. C. Eisenstat, *A divide-and-conquer algorithm for the bidiagonal SVD*, SIAM Journal on Matrix Analysis and Applications, 16 (1995),

pp. 79–92, https://doi.org/10.1137/S0895479892242232.

[60] A. Haidar, J. Kurzak, and P. Luszczek, *An improved parallel singular value algorithm and its implementation for multicore hardware*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13), ACM, 2013, p. 90, https://doi.org/10.1145/2503210.2503292.

[61] A. Haidar, H. Ltaief, and J. Dongarra, *Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels*, in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), ACM, 2011, pp. 8:1–8:11, https://doi.org/10.1145/2063384.2063394.

[62] A. Haidar, H. Ltaief, P. Luszczek, and J. Dongarra, *A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction*, in 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2012, pp. 25–35, https://doi.org/10.1109/IPDPS.2012.13.

[63] S. Hammarling, *A note on modifications to the Givens plane rotation*, IMA Journal of Applied Mathematics, 13 (1974), pp. 215–218, https://doi.org/10.1093/imamat/13.2.215.

[64] V. Hari, *Accelerating the SVD block-Jacobi method*, Computing, 75 (2005), pp. 27–53, https://doi.org/10.1007/s00607-004-0113-z.

[65] V. Hari and J. Matejaš, *Accuracy of two SVD algorithms for 2× 2 triangular matrices*, Applied Mathematics and Computation, 210 (2009), pp. 232–257, https://doi.org/10.1016/j.amc.2008.12.086.

[66] V. Hari and K. Veselić, *On Jacobi methods for singular value decompositions*, SIAM Journal on Scientific and Statistical Computing, 8 (1987), pp. 741–754, https://doi.org/10.1137/0908064.

[67] M. Heath, A. Laub, C. Paige, and R. Ward, *Computing the singular value decomposition of a product of two matrices*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 1147–1159, https://doi.org/10.1137/0907078.

[68] M. R. Hestenes, *Inversion of matrices by biorthogonalization and related results*, Journal of the Society for Industrial and Applied Mathematics, 6 (1958), pp. 51–90, https://doi.org/10.1137/0106005.

[69] G. W. Howell, J. W. Demmel, C. T. Fulton, S. Hammarling, and K. Marmol, *Cache efficient bidiagonalization using BLAS 2.5 operators*, ACM Transactions on Mathematical Software (TOMS), 34 (2008), p. 14, https://doi.org/10.1145/1356052.1356055.

[70] IBM Corporation, *ESSL Guide and Reference*, 2016, http://publib.boulder.ibm.com/epubs/pdf/a2322688.pdf.

[71] Intel Corporation, *User's Guide for Intel Math Kernel Library for Linux OS*, 2015, http://software.intel.com/en-us/mkl-for-linux-userguide.

[72] I. C. F. Ipsen, *Computing an eigenvector with inverse iteration*, SIAM Review, (2006), pp. 254–291, https://doi.org/10.1137/S0036144596300773.

[73] C. G. J. Jacobi, *Über ein leichtes verfahren die in der theorie der säcularstörungen vorkommenden gleichungen numerisch aufzulösen.*, Journal für die reine und angewandte Mathematik, 30 (1846), pp. 51–94, http://eudml.org/doc/147275.

[74] E. Jessup and D. Sorensen, *A divide and conquer algorithm for computing the singular value decomposition*, in Proceedings of the Third SIAM Conference

on Parallel Processing for Scientific Computing, Philadelphia, PA, 1989, SIAM, pp. 61–66.

[75] W. Kahan, *Accurate eigenvalues of a symmetric tri-diagonal matrix*, tech. report, Stanford University, Stanford, CA, USA, 1966, http://www.dtic.mil/docs/citations/AD0638796.

[76] E. Kogbetliantz, *Solution of linear equations by diagonalization of coefficients matrix*, Quarterly of Applied Mathematics, 13 (1955), pp. 123–132, http://www.ams.org/journals/qam/1955-13-02/S0033-569X-1955-88795-9/S0033-569X-1955-88795-9.pdf.

[77] J. Kurzak, P. Wu, M. Gates, I. Yamazaki, P. Luszczek, G. Ragghianti, and J. Dongarra, *Designing SLATE: Software for linear algebra targeting exascale*, SLATE Working Note 3, Innovative Computing Laboratory, University of Tennessee, Sep 2017, http://www.icl.utk.edu/publications/swan-003.

[78] B. Lang, *Parallel reduction of banded matrices to bidiagonal form*, Parallel Computing, 22 (1996), pp. 1–18, https://doi.org/10.1016/0167-8191(95)00064-X.

[79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for FORTRAN usage*, ACM Transactions on Mathematical Software (TOMS), 5 (1979), pp. 308–323, https://doi.org/10.1145/355841.355847.

[80] R.-C. Li, *Solving secular equations stably and efficiently*, Tech. Report UCB//CSD-94-851, University of California Berkeley, Computer Science Division, 1994, http://www.netlib.org/lapack/lawns/. Also: LAPACK Working Note 89.

[81] S. Li, M. Gu, L. Cheng, X. Chi, and M. Sun, *An accelerated divide-and-conquer algorithm for the bidiagonal SVD problem*, SIAM Journal on Matrix Analysis and Applications, 35 (2014), pp. 1038–1057, https://doi.org/10.1137/130945995.

[82] H. Ltaief, J. Kurzak, and J. Dongarra, *Parallel two-sided matrix reduction to band bidiagonal form on multicore architectures*, IEEE Transactions on Parallel and Distributed Systems, 21 (2010), pp. 417–423, https://doi.org/10.1109/TPDS.2009.79.

[83] H. Ltaief, P. Luszczek, and J. Dongarra, *High-performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures*, ACM Transactions on Mathematical Software (TOMS), 39 (2013), pp. 16:1–16:22, https://doi.org/10.1145/2450153.2450154.

[84] F. T. Luk, *Computing the singular-value decomposition on the ILLIAC IV*, ACM Transactions on Mathematical Software (TOMS), 6 (1980), pp. 524–539, https://doi.org/10.1145/355921.355925.

[85] F. T. Luk and H. Park, *On parallel Jacobi orderings*, SIAM Journal on Scientific and Statistical Computing, 10 (1989), pp. 18–26, https://doi.org/10.1137/0910002.

[86] O. Marques and P. B. Vasconcelos, *Computing the bidiagonal SVD through an associated tridiagonal eigenproblem*, in International Conference on Vector and Parallel Processing (VECPAR), Springer, 2016, pp. 64–74, https://doi.org/10.1007/978-3-319-61982-8_8.

[87] W. F. Mascarenhas, *On the convergence of the Jacobi method for arbitrary orderings*, SIAM Journal on Matrix Analysis and Applications, 16 (1995), pp. 1197–1209, https://doi.org/10.1137/S0895479890179631.

[88] J. Matejaš and V. Hari, *Accuracy of the Kogbetliantz method for scaled diag-*

onally dominant triangular matrices, Applied Mathematics and Computation, 217 (2010), pp. 3726–3746, https://doi.org/10.1016/j.amc.2010.09.020.

[89] J. Matejaš and V. Hari, *On high relative accuracy of the Kogbetliantz method*, Linear Algebra and its Applications, 464 (2015), pp. 100–129, https://doi.org/10.1016/j.laa.2014.02.024.

[90] MathWorks, *MATLAB*, 2017, http://www.mathworks.com/products/matlab.html.

[91] J. D. McCalpin, *A survey of memory bandwidth and machine balance in current high performance computers*, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, (1995), pp. 19–25, http://tab.computer.org/tcca/NEWS/DEC95/dec95_mccalpin.ps.

[92] B. Moore, *Principal component analysis in linear systems: Controllability, observability, and model reduction*, IEEE Transactions on Automatic Control, 26 (1981), pp. 17–32, https://doi.org/10.1109/TAC.1981.1102568.

[93] MPI Forum, *MPI: A message-passing interface standard: Version 3.1*, June 2015, http://www.mpi-forum.org/.

[94] NVIDIA Corporation, *CUDA Toolkit 7.0*, March 2015, http://developer.nvidia.com/cuda-zone.

[95] G. Okša and M. Vajteršic, *Efficient pre-processing in the parallel block-Jacobi SVD algorithm*, Parallel Computing, 32 (2006), pp. 166–176, https://doi.org/10.1016/j.parco.2005.06.006.

[96] OpenBLAS, *OpenBLAS User Manual*, 2016, http://www.openblas.net/.

[97] B. N. Parlett, *The new qd algorithms*, Acta Numerica, 4 (1995), pp. 459–491, https://doi.org/10.1017/S0962492900002580.

[98] B. N. Parlett and I. S.Dhillon, *Fernando's solution to Wilkinson's problem: An application of double factorization*, Linear Algebra and its Applications, 267 (1997), pp. 247–279, https://doi.org/10.1016/S0024-3795(97)80053-5.

[99] V. Rokhlin, A. Szlam, and M. Tygert, *A randomized algorithm for principal component analysis*, SIAM Journal on Matrix Analysis and Applications, 31 (2009), pp. 1100–1124, https://doi.org/10.1137/080736417.

[100] H. Rutishauser, *Der quotienten-differenzen-algorithmus*, Zeitschrift für angewandte Mathematik und Physik ZAMP, 5 (1954), pp. 233–251, https://doi.org/10.1007/BF01600331.

[101] H. Rutishauser, *Solution of eigenvalue problems with the LR-transformation*, National Bureau of Standards Applied Mathematics Series, 49 (1958), pp. 47–81.

[102] H. Rutishauser, *The Jacobi method for real symmetric matrices*, in Handbook for Automatic Computation: Volume II: Linear Algebra, vol. 186 of Grundlehren Der Mathematischen Wissenschaften, Springer-Verlag, New York, NY, 1971, pp. 202–211, https://doi.org/10.1007/978-3-642-86940-2.

[103] A. H. Sameh, *On Jacobi and Jacobi-like algorithms for a parallel computer*, Mathematics of Computation, 25 (1971), pp. 579–590, https://doi.org/10.1090/S0025-5718-1971-0297131-6.

[104] R. Schreiber and C. Van Loan, *A storage-efficient WY representation for products of Householder transformations*, SIAM Journal on Scientific and Statistical Computing, 10 (1989), pp. 53–57, https://doi.org/10.1137/0910005.

[105] B. T. Smith, J. M. Boyle, J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide, Second Edition*, vol. 6 of Lecture Notes in Computer Science, Springer, Berlin, 1976, https://doi.org/10.1007/3-540-07546-1.

[106] G. W. Stewart, *The efficient generation of random orthogonal matrices with an application to condition estimators*, SIAM Journal on Numerical Analysis, 17 (1980), pp. 403–409, https://doi.org/10.1137/0717034.

[107] G. W. Stewart, *On the early history of the singular value decomposition*, SIAM Review, 35 (1993), pp. 551–566, https://doi.org/10.1137/1035134.

[108] G. W. Stewart, *QR sometimes beats Jacobi*, Tech. Report CS-TR-3434, University of Maryland, 1995, http://drum.lib.umd.edu/handle/1903/709.

[109] S. Tomov, R. Nath, and J. Dongarra, *Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing*, Parallel Computing, 36 (2010), pp. 645–654, https://doi.org/10.1016/j.parco.2010.06.001.

[110] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, *Dense linear algebra solvers for multicore with GPU accelerators*, in 2010 IEEE International on Parallel and Distributed Processing Symposium, Workshops and Phd Forum (IPDPSW), IEEE, 2010, pp. 1–8, https://doi.org/10.1109/IPDPSW.2010.5470941.

[111] M. A. Turk and A. P. Pentland, *Face recognition using eigenfaces*, in Computer Vision and Pattern Recognition, 1991. Proceedings CVPR'91., IEEE Computer Society Conference on, IEEE, 1991, pp. 586–591, https://doi.org/10.1109/CVPR.1991.139758.

[112] C. Van Loan, *The block Jacobi method for computing the singular value decomposition*, Tech. Report TR 85-680, Cornell University, 1985, https://ecommons.cornell.edu/handle/1813/6520.

[113] F. G. Van Zee, R. A. Van de Geijn, and G. Quintana-Ortí, *Restructuring the tridiagonal and bidiagonal QR algorithms for performance*, ACM Transactions on Mathematical Software (TOMS), 40 (2014), p. 18, https://doi.org/10.1145/2535371.

[114] F. G. Van Zee, R. A. Van De Geijn, G. Quintana-Ortí, and G. J. Elizondo, *Families of algorithms for reducing a matrix to condensed form*, ACM Transactions on Mathematical Software (TOMS), 39 (2012), p. 2, https://doi.org/10.1145/2382585.2382587.

[115] R. C. Whaley and J. Dongarra, *Automatically tuned linear algebra software*, in Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, 1998, pp. 1–27, https://doi.org/10.1109/SC.1998.10004.

[116] J. H. Wilkinson, *Note on the quadratic convergence of the cyclic Jacobi process*, Numerische Mathematik, 4 (1962), pp. 296–300, https://doi.org/10.1007/BF01386321.

[117] J. H. Wilkinson and C. Reinsch, *Handbook for Automatic Computation: Volume II: Linear Algebra*, vol. 186 of Grundlehren Der Mathematischen Wissenschaften, Springer-Verlag, New York, NY, 1971, https://doi.org/10.1007/978-3-642-86940-2.

[118] P. R. Willems and B. Lang, *A framework for the MR^3 algorithm: theory and implementation*, SIAM Journal on Scientific Computing, 35 (2013), pp. A740–A766, https://doi.org/10.1137/110834020.

[119] P. R. Willems, B. Lang, and C. Vömel, *Computing the bidiagonal SVD using multiple relatively robust representations*, SIAM Journal on Matrix Analysis and Applications, 28 (2006), pp. 907–926, https://doi.org/10.1137/050628301.

[120] B. B. Zhou and R. P. Brent, *A parallel ring ordering algorithm for efficient one-sided Jacobi SVD computations*, Journal of Parallel and Distributed

Computing, 42 (1997), pp. 1–10, https://doi.org/10.1006/jpdc.1997.1304.