NUMERICAL LINEAR ALGEBRA WITH APPLICATIONS
Numer. Linear Algebra Appl. (2016)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/nla.2046

New fast divide-and-conquer algorithms for the symmetric tridiagonal eigenvalue problem

Shengguo Li[1,*,†], Xiangke Liao[1,2], Jie Liu[1,3] and Hao Jiang[1]

[1]*College of Computer, National University of Defense Technology (NUDT), Changsha 410073, China*
[2]*Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology (NUDT), Changsha 410073, China*
[3]*State Key Laboratory of High Performance Computing, National University of Defense Technology (NUDT), Changsha 410073, China*

SUMMARY

In this paper, two accelerated divide-and-conquer (ADC) algorithms are proposed for the symmetric tridiagonal eigenvalue problem, which cost $O(N^2r)$ flops in the worst case, where N is the dimension of the matrix and r is a modest number depending on the distribution of eigenvalues. Both of these algorithms use hierarchically semiseparable (HSS) matrices to approximate some intermediate eigenvector matrices, which are Cauchy-like matrices and are off-diagonally low-rank. The difference of these two versions lies in using different HSS construction algorithms, one (denoted by ADC1) uses a structured low-rank approximation method and the other (ADC2) uses a randomized HSS construction algorithm. For the ADC2 algorithm, a method is proposed to estimate the off-diagonal rank. Numerous experiments have been carried out to show their stability and efficiency. These algorithms are implemented in parallel in a shared memory environment, and some parallel implementation details are included. Comparing the ADCs with highly optimized multithreaded libraries such as Intel MKL, we find that ADCs could be more than six times faster for some large matrices with few deflations. Copyright © 2016 John Wiley & Sons, Ltd.

Received 28 June 2014; Revised 30 December 2015; Accepted 19 February 2016

KEY WORDS: HSS matrices; cauchy-like matrices; eigenvalue problems; Schur complements; divide-and-conquer algorithm

1. INTRODUCTION

Computing the eigendecomposition of a symmetric tridiagonal matrix is a classic linear algebra problem and is ubiquitous in computational science. Some well-known algorithms include the QR algorithm [1, 2], the multiple relatively robust representations [3] and the divide-and-conquer (DC) algorithm [4, 5]. According to the comparisons in [6], DC and multiple relatively robust representations are generally faster than QR for large matrices, especially when the eigenvectors are required. In this work, we focus on the DC algorithm, and the goal is to develop an improved version.

Although DC is very fast in practice, which takes $O(N^{2.3})$ flops on average [7, 8], for some matrices with few deflations, its complexity [4, 5, 9] can be $O(N^3)$. By using hierarchically semiseparable (HSS) matrices [10, 11], we show that its worst case complexity can be reduced to $O(N^2r)$, where r is a modest number and is usually much smaller than a big N. The main observation of this work is from the following theorem.

*Correspondence to: Shengguo Li, College of Computer, National University of Defense Technology (NUDT), Changsha 410073, China.
†E-mail: nudtlsg@nudt.edu.cn

Copyright © 2016 John Wiley & Sons, Ltd.

Theorem 1 (Bunch, Nielsen, and Sorensen [12]*)*
Assume that $D = \mathrm{diag}(d_1, \cdots, d_N)$, such that $d_1 < d_2 < \cdots < d_N$, and $u \in \mathbb{R}^N$ is a vector and a scalar $\rho > 0$. Then, the eigenvector corresponding to λ_i, an eigenvalue of $M = D + \rho u u^T$, is

$$q_i = \left(\frac{u_1}{d_1 - \lambda_i}, \cdots, \frac{u_N}{d_N - \lambda_i} \right)^T \Big/ \sqrt{\sum_{j=1}^{N} \frac{u_j^2}{(d_j - \lambda_i)^2}}, \tag{1}$$

and the eigenvalues of M satisfy

$$d_1 < \lambda_1 < d_2 < \lambda_2 < \cdots < d_N < \lambda_N. \qquad \square \tag{2}$$

It shows that $Q = \left(\frac{u_i v_j}{d_i - \lambda_j} \right)_{i,j}$ with $v_j = 1/\sqrt{\sum_{k=1}^{N} \frac{u_k^2}{(d_k - \lambda_j)^2}}$ is a Cauchy-like matrix. Recall that C is called Cauchy-like if it satisfies

$$D \cdot C - C \cdot \Lambda = u \cdot v^T, \tag{3}$$

where $D = \mathrm{diag}(d_1, \cdots, d_N), \Lambda = \mathrm{diag}(\lambda_1, \cdots, \lambda_N)$ and $u, v \in \mathbb{R}^N$, which are called the *generators* of Cauchy-like matrix Q. It is easy to check that Q is also off-diagonally low-rank. To take advantage of these two properties, we can use an HSS matrix to approximate Q and then use the fast HSS matrix multiplication algorithm to update the eigenvectors, like the bidiagonal SVD case [13]. A structured low-rank approximation method is designed for a Cauchy-like matrix in [13, 14], called structured rank-revealing Schur-complement factorization (SRRSC), which can be used to construct HSS matrices efficiently. By incorporating SRRSC into DC, an accelerated DC (ADC) algorithm is proposed for the singular value problem in [13], where ADC is three times faster than DC in Intel MKL for some large matrices. We show that this technique also works for the symmetric tridiagonal eigenvalue problem, which is presented in Example 3 in Section 5. In this paper, we implement the ADC algorithm in parallel, and it obtains even better speedups.

In this paper, we further show that the randomized HSS construction algorithm [15] can also be used to compute the eigendecomposition reliably, and it can obtain similar speedups as using SRRSC when compared with Intel MKL. Recent work has suggested the efficiency of the randomized algorithms [16, 17] for computing a low-rank matrix approximation. Martinsson [15] has developed a novel HSS construction algorithm by using the randomized sampling technique, and for simplicity we refer to this algorithm as `RSHSS`. This method is extremely suitable for matrices with fast matrix-vector multiplication algorithms. For example, if the matrix-vector product costs $O(N)$ flops, RSHSS has linear complexity $O(Nr^2)$, where N is the dimension of the matrix and r is its maximum numerical rank of off-diagonal blocks. For matrix Q in (1), the fast multipole method (FMM) [18, 19] can be used for the matrix-vector product in $O(N \log N)$ flops. Therefore, an HSS matrix approximation to Q can be constructed in $O(Nr^2)$ flops if combining RSHSS with FMM. To use RSHSS, we need to know the maximum rank of off-diagonal blocks, which is difficult for general matrices. Fortunately, the off-diagonal rank of the matrix Q defined in (1) can be estimated by using the approximation theory of function $1/x$. We show in Section 2 that the estimated rank based on the exponential expansion [20] is quite acceptable.

By adding the HSS matrix techniques to the symmetric DC algorithm [4, 5, 9], two new ADC algorithms are obtained. One is denoted by ADC1 based on SRRSC, and the other is denoted by ADC2 based on RSHSS. Similar to the analysis in [7], the complexity of ADCs can be shown to be $O(N^2 r)$, where N is the dimension of a symmetric tridiagonal matrix T and r is a modest integer, which is related to the distribution of eigenvalues. Because the HSS matrix construction [10, 15, 21] and multiplication [22] algorithms are naturally parallelizable, we further implement these two algorithms in parallel by using OpenMP in a shared memory multicore environment. We also simply parallelize the classical process of DC algorithm by using OpenMP such as solving the subproblems at the bottom level of the DC tree and all the secular equations. Numerous experiments have been carried out to test these two ADC algorithms. It turns out that our ADCs can be about six times faster than the DC implementation in MKL for some large matrices with few deflations. The accuracy comparisons are also included in Section 5.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

2. PRELIMINARY

Assume that T is a symmetric tridiagonal matrix,

$$T = \text{tridiag}\begin{pmatrix} & b_1 & b_2 & \cdot & b_{N-2} & & b_{N-1} & \\ a_1 & & a_2 & \cdot & \cdot & a_{N-1} & & a_N \\ & b_1 & b_2 & \cdot & b_{N-2} & & b_{N-1} & \end{pmatrix}. \tag{4}$$

We briefly introduce some formulae of Cuppen's divide-and-conquer algorithm [4, 9]. T is decomposed into the sum of two matrices,

$$T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + b_k v v^T, \tag{5}$$

where $T_1 \in R^{k\times k}$ and $v = [0, \ldots, 0, 1, 1, 0, \ldots, 0]^T$ with ones at the k-th and $(k+1)$-th entries. If $T_1 = Q_1 D_1 Q_1^T$ and $T_2 = Q_2 D_2 Q_2^T$, then T can be written as

$$T = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix}\left(\begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix} + b_k u u^T\right)\begin{bmatrix} Q_1^T & \\ & Q_2^T \end{bmatrix}, \tag{6}$$

where $u = \begin{bmatrix} Q_1^T & \\ & Q_2^T \end{bmatrix} v = \begin{bmatrix} \text{last column of } Q_1^T \\ \text{first column of } Q_2^T \end{bmatrix}$. By Theorem 1, we can obtain the eigenvectors Q of the middle matrix at the right-hand side of (6). The eigenvectors of T would be $\begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} Q$.

The matrices $T_i, i = 1, 2$ can also be divided recursively. More numerical details can be found in [9] and [7]. We only point out that the eigenvectors cannot be computed directly from (1). In practice, the computed $\hat{\lambda}_i$ is only an approximation to λ_i. To compute the eigenvectors orthogonally [5, 23], we need to use Löwner's theorem [24] to recompute the vector u as

$$\hat{u}_i = \sqrt{\prod_{j=1}^{i-1}\left(\frac{\hat{\lambda}_j - d_i}{d_j - d_i}\right)\cdot \prod_{j=i+1}^{N}\left(\frac{\hat{\lambda}_j - d_i}{d_j - d_i}\right)\cdot(\hat{\lambda}_i - d_i)}, \tag{7}$$

and then use (1) to compute the eigenvectors.

2.1. The low-rank structure of Q

To be more specific, the matrix Q is defined as

$$Q = \begin{bmatrix} \frac{u_1 v_1}{d_1-\lambda_1} & \frac{u_1 v_2}{d_1-\lambda_2} & \cdots & \frac{u_1 v_N}{d_1-\lambda_N} \\ \frac{u_2 v_1}{d_2-\lambda_1} & \frac{u_2 v_2}{d_2-\lambda_2} & \cdots & \frac{u_2 v_N}{d_2-\lambda_N} \\ \vdots & \vdots & & \vdots \\ \frac{u_N v_1}{d_N-\lambda_1} & \frac{u_N v_2}{d_N-\lambda_2} & \cdots & \frac{u_N v_N}{d_N-\lambda_N} \end{bmatrix}. \tag{8}$$

Because $\{d_i\}$ and $\{\lambda_j\}$ are interlacing (2), Q is usually off-diagonally low rank, and the ranks of off-diagonal blocks depend on the distribution of $\{d_i\}$ and $\{\lambda_i\}$. We use the following example to show that.

Example 1
Assume that a matrix Q satisfies the eigendecomposition $M = D + uu^T = Q\Lambda Q^T$, where $D = \text{diag}(d_1, d_2, \ldots, d_N)$, $d_i = i \cdot \frac{b-a}{N}$, $a = 1.0$, $b = 9.0$, for $i = 1, \ldots, N$ and $u \in \mathbb{R}^N$ is a random normalized vector. The off-diagonal low-rank property of Q is shown in Table I, which includes the numerical ranks of the submatrices $Q(1 : m, m+1 : N)$ for different m with $N = 2000$. The ranks are computed by truncating the singular values less than $1.0e^{-13}$.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

Table I. The ranks of different off-diagonal blocks of Q.

m	100	200	300	400	500	600	700	800	900	1000
rank	18	20	21	22	23	23	23	24	24	24

The ranks of the off-diagonal blocks can be estimated by using the approximation theory of function $f(x) = 1/x$. The element $\frac{1}{d_i - \lambda_j}$ can be approximated by the sums of exponentials,

$$\frac{1}{d_i - \lambda_j} \approx \sum_{k=1}^{r} \omega_k e^{-\alpha_k (d_i - \lambda_j)} := s_r(d_i - \lambda_j). \tag{9}$$

Assume that d_i and λ_j belong to two different subintervals of $[\hat{a}, \hat{b}]$, $d_i \in I_1, \lambda_j \in I_2, I_1 \cap I_2 = \emptyset$ and $d_i < \lambda_j$. Denote $\text{dist}(I_1, I_2) = \min_{d_i \in I_1, \lambda_j \in I_2} |d_i - \lambda_j|$, (In our case, d_i and λ_j are the eigenvalues of D and M, respectively, $\hat{a} = d_1$ and $\hat{b} = \lambda_N$, the largest eigenvalue of M), then

$$1 \leqslant \frac{d_i - \lambda_j}{\text{dist}(I_1, I_2)} \leqslant \frac{\hat{b} - \hat{a}}{\text{dist}(I_1, I_2)} := R. \tag{10}$$

An approximation error bound is given in [20] for the sums of exponentials,

$$\left| \frac{1}{x} - s_r(x) \right| \leqslant 16 e^{-\frac{r\pi^2}{\log(8R)}}, \tag{11}$$

where $s_r(x)$ is defined in (9) and $x \in [1, R]$. As long as the number of approximation terms satisfies

$$r \geqslant \left\lceil \frac{\log(16/\epsilon) \log(8R)}{\pi^2} \right\rceil, \tag{12}$$

the approximation error of the sum of exponentials is less than ϵ, a small constant. For example, when $m = 300$ and $\epsilon = 1e\text{-}13$, $\text{dist}(I_1, I_2) = 4.3e\text{-}3$, $R = 2.0e3$ and the off-diagonal rank estimated by (11) is 32, which is close to the result in Table I. Equation (11) would be used to estimate the rank in Algorithm 2.

Remark 1
To keep the orthogonality of Q , $d_i - \lambda_j$ must be computed by

$$d_i - \lambda_j = \begin{cases} (d_i - d_j) - \gamma_j & \text{if } i \leqslant j \\ (d_i - d_{j+1}) + \mu_j & \text{if } i > j \end{cases}, \tag{13}$$

where $\gamma_i = \lambda_i - d_i$ (the distance between λ_i and d_i) and $\mu_i = d_{i+1} - \lambda_i$ (the distance between λ_i and d_{i+1}), which can be returned by calling the LAPACK routine `dlaed4`. If using FMM to compute the eigenvectors, Equation (13) shows that some modifications of classic FMM are needed because some d_i and λ_j may equal in double precision but γ_i is not zero.

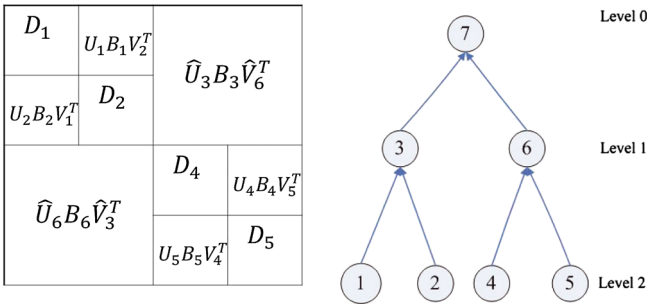
2.2. Introduction to hierarchically semiseparable matrices

The HSS matrices are a very important type of rank-structured matrices that share the same property that the off-diagonal blocks are low-rank. Other rank-structured matrices include $\mathcal{H}$-matrix [25, 26], $\mathcal{H}^2$-matrix [27, 28], quasiseparable matrices [29, 30] and sequentially semiseparable [31, 32] matrices. The HSS matrix was first discussed in [10, 33], which can be seen as an algebraic counterpart of FMM in [34]. In this paper we use the HSS matrix to accelerate the computation of eigenvectors, and other rank-structured matrices can be similarly used too. We follow the notation in [11, 35] and briefly introduce some key concepts of the HSS matrix.

Let $\mathcal{I} = \{1, 2, \ldots, N\}$ and $\mathcal{T}$ be a *postordered* binary tree, which means the ordering of a nonleaf node i satisfies $i_1 < i_2 < i$, where i_1 is its left child and i_2 is its right child. Each node i is associated with a contiguous subset of $\mathcal{I}$, t_i, satisfying the following conditions:

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla



(a) Matrix A (b) HSS tree

Figure 1. A 4×4 hierarchically semiseparable (HSS) matrix and its corresponding HSS tree. (a) Matrix A and (b) HSS tree.

- $t_{i_1} \cup t_{i_2} = t_i$ and $t_{i_1} \cap t_{i_2} = \emptyset$, for a parent node i with left child i_1 and right child i_2;
- $\cup_{i \in LN} t_i = \mathcal{I}$, where LN denotes the set of all leaf nodes;
- $t_{\text{root}(\mathcal{T})} = \mathcal{I}$, root$(\mathcal{T})$ denotes the root of $\mathcal{T}$.

A block row or column excluding the diagonal block is called an *HSS block row or column*, denoted by

$$H_i^{row} = A_{t_i \times (\mathcal{I} \setminus t_i)}, \quad H_i^{col} = A_{(\mathcal{I} \setminus t_i) \times t_i},$$

associated with node i. We also simply call them *HSS blocks*. As in [35], the maximum (numerical) rank of all the HSS blocks is called *HSS rank*.

For each node i in $\mathcal{T}$, there are matrices $\widehat{D}_i$, $\widehat{U}_i$, $\widehat{V}_i$ and B_i associated with it, called *generators*, such that

$$\begin{aligned} \widehat{D}_i &= A|_{t_i \times t_i} = \begin{bmatrix} \widehat{D}_{i_1} & \widehat{U}_{i_1} B_{i_1} \widehat{V}_{i_2}^T \\ \widehat{U}_{i_2} B_{i_2} \widehat{V}_{i_1}^T & \widehat{D}_{i_2} \end{bmatrix}, \\ \widehat{U}_i &= \begin{bmatrix} \widehat{U}_{i_1} & \\ & \widehat{U}_{i_2} \end{bmatrix} U_i, \quad \widehat{V}_i = \begin{bmatrix} \widehat{V}_{i_1} & \\ & \widehat{V}_{i_2} \end{bmatrix} V_i. \end{aligned} \tag{14}$$

For a leaf node i, $\widehat{D}_i = D_i$, $\widehat{U}_i = U_i$, $\widehat{V}_i = V_i$. Figure 1(a) shows a 4×4 HSS matrix A, and it can be written as

$$A = \begin{bmatrix} \begin{bmatrix} D_1 & U_1 B_1 V_2^T \\ U_2 B_2 V_1^T & D_2 \end{bmatrix} & \widehat{U}_3 B_3 \widehat{V}_6^T \\ \widehat{U}_6 B_6 \widehat{V}_3^T & \begin{bmatrix} D_4 & U_4 B_4 V_5^T \\ U_5 B_5 V_4^T & D_5 \end{bmatrix} \end{bmatrix}, \tag{15}$$

and Figure 1(b) shows its corresponding postordering HSS tree.

Remark 2

(1) The generators of a Cauchy-like matrix (3) can be represented by four vectors, while the generators of an HSS matrix are *matrices*. For an HSS matrix, we only need to store the generators D_i, U_i, V_i and B_i, and $\widehat{D}_i$, $\widehat{U}_i$ and $\widehat{V}_i$ can be constructed hierarchically when needed.

(2) The HSS representation (14) is equivalent to the previous representations in [10, 33, 35] but is simpler (generators R and W are not introduced). For a parent node i, if let $U_i = \begin{bmatrix} R_{i_1} \\ R_{i_2} \end{bmatrix}$,

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

$V_i = \begin{bmatrix} W_{i_1} \\ W_{i_2} \end{bmatrix}$, then (15) reduces to the form in [35],

$$A = \begin{bmatrix} D_1 & U_1 B_1 V_2^T & U_1 R_1 B_3 W_4^T V_4^T & U_1 R_1 B_3 W_5^T V_5^T \\ U_2 B_2 V_1^T & D_2 & U_2 R_2 B_3 W_4^T V_4^T & U_2 R_2 B_3 W_5^T V_5^T \\ U_4 R_4 B_6 W_1^T V_1^T & U_4 R_4 B_6 W_2^T V_2^T & D_4 & U_4 B_4 V_5^T \\ U_5 R_5 B_6 W_1^T V_1^T & U_5 R_5 B_6 W_2^T V_2^T & U_5 B_5 V_4^T & D_5 \end{bmatrix}.$$

2.3. Accelerated tridiagonal divide-and-conquer algorithm

The procedure of ADC algorithms is expressed in the following algorithm, which is similar to the DC algorithm [7].

ALGORITHM 1. [ADC(T, Q, Λ)] Compute the whole eigendecomposition of a symmetric tridiagonal matrix by using the ADC algorithm. Let $m(= 25)$ be a small integer constant.

`if` the row dimension of T is less than m

use QR algorithm [42] to compute $T = Q\Lambda Q^T$;

`return` Q and Λ;

`else`

form $T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + b_k v v^T$;

call ADC(T_1, Q_1, Λ_1);

call ADC(T_2, Q_2, Λ_2);

form $M = D + b_k u u^T$ from $Q_1, Q_2, \Lambda_1, \Lambda_2$;

`if` the size of M is small

find the eigenvalues Λ and eigenvectors Q' of M;

compute $Q = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \cdot Q'$;

`else`

find the eigenvalues Λ of M and construct an HSS matrix $H_Q \approx Q'$;

compute $Q = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \cdot H_Q$ via the HSS matrix multiplication algorithm;

`end if`

`return` Q and Λ;

`end if`

Remark 3

Algorithm 1 is more like a framework for accelerating the tridiagonal DC algorithm, because the HSS matrices can be replaced by other rank-structured matrices such as $\mathcal{H}$-,$\mathcal{H}^2$-matrix. The difference between Algorithm 1 and the standard DC algorithm is that Algorithm 1 uses the HSS matrix techniques to update the eigenvectors when the size of matrix M is large, to reduce the complexity cost. If the sizes of secular equations are always small, that is, most eigenvalues are computed by deflation, ADC is equivalent to the standard DC.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

3. RANDOMIZED HIERARCHICALLY SEMISEPARABLE CONSTRUCTION ALGORITHM

The random HSS construction algorithm proposed in [15] is built on two low-rank approximation algorithms: random sampling (RS) [16, 17] and interpolative decomposition (ID) [36, 37]. The form of ID has appeared in the rank-revealing QR [38] and rank-revealing lower upper factorization [39], and it also has a close relationship to the matrix skeleton and CUR factorization [40, 41].

We introduce RS first. For a given $m \times n$ matrix B with $m < n$, we want to find a tall matrix Q with orthogonal columns such that

$$\|B - QQ^*B\| < \epsilon,$$

where ϵ is a small constant. The random sampling method right multiplies B with a Gaussian random matrix $\Omega \in \mathbb{R}^{n\times(r+p)}$, and obtain a 'compressed' matrix $Y = B\Omega$ with much fewer columns, $(r + p) \ll n$, where r is the numerical rank of B and p is the oversampling parameter, usually $p = 5, 10$ or 20. Then, the matrix Q can be obtained by applying the rank-revealing QR [38, 42] or the truncated SVD [1] to Y. It is shown in [15, 16] that the RS algorithm computes a good low-rank approximation with quite high probability. For example, the computed Q satisfies

$$\|B - QQ^TB\| \leqslant \left(1 + 11\sqrt{(r + p)\min(m, n)}\right)\sigma_{r+1},$$

with probability at least $1 - 6p^{-p}$ [15].

Remark 4
In general, the rank r is rarely known in advance. For the symmetric tridiagonal DC algorithm, we can use (11) as a guide to estimate r.

The ID method computes an approximate low-rank factorization of B such that

$$B \approx B(:, J)\tilde{X} \cdot P = B(:, J)X,$$

where J is a subset of the column indices of B, $\tilde{X}$ is an $r \times n$ matrix with an $r \times r$ identity matrix as a submatrix and all its entries are less than one in magnitude and P is a permutation matrix. A stable and accurate method for computing ID is proposed in [36], similar to the rank-revealing QR algorithm in [38]. We can combine RS with ID to obtain a more efficient low-rank approximation algorithm [17]. For a given $n \times n$ matrix B, generate an $n \times (r + p)$ Gaussian random matrix Ω as before and compute the row sampling and column sampling matrices $Y = B\Omega$ and $Z = B^T\Omega$. Then, use ID to determine the r selected rows and columns of B from Y and Z,

$$[X^{row}, I^{row}] = \texttt{interpolative}(Y^T), \quad \left[X^{col}, J^{col}\right] = \texttt{interpolative}(Z^T), \quad (16)$$

and B can be approximated by

$$B \approx X^{row} \cdot B\left(I^{row}, J^{col}\right) \cdot \left(X^{col}\right)^T.$$

3.1. Random hierarchically semiseparable construction for Cauchy-like matrices

The main idea is to apply the randomized ID to the row and column sampling matrices by traversing the HSS tree level-by-level, from bottom to top. To illustrate it, let A be a matrix as defined in (8), $Y = A\Omega^{(1)}$ and $Z = A^T\Omega^{(2)}$ be the sampling matrices, where $\Omega^{(i)}$ is a Gaussian random matrix for $i = 1, 2$. To construct an HSS matrix, we need to find the low-rank approximations of all HSS blocks, H_i^{row} and H_i^{col}. Recall that H_i^{row} and H_i^{col} are the i-th HSS block row and column, satisfying

$$A_{t_i\times\mathcal{I}} = H_i^{row} + D_i, \quad A_{\mathcal{I}\times t_i} = H_i^{col} + D_i. \quad (17)$$

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

In this section, we show how to obtain the low-rank approximations from Y and Z by using the randomized ID method. For a leaf node i, its compressed HSS block row and column are, respectively,

$$\Phi_i = Y_i - D_i\Omega_i^{(1)}, \quad \Theta_i = Z_i - D_i^T\Omega_i^{(2)},$$

where $(\star)_i$ means $(\star)(t_i,:)$ for $(\star) = Y, Z, \Omega^{(1)}$ and $\Omega^{(2)}$. By applying the ID method to Φ_i and Θ_i, we can easily obtain the low-rank approximations to H_i^{row} and H_i^{col}, respectively.

For a parent node, its compressed HSS blocks can be neatly obtained from those of its children recursively, see Section 4.1 in [15] and Algorithm 2 in the following. Then, its generators can be obtained similarly by applying ID to the compressed HSS blocks.

ALGORITHM 2. (Random HSS construction for Cauchy-like matrices) Given the generators of Cauchy-like matrix A, compute its HSS matrix approximation accurately.
First, use (11) to estimate the HSS rank r of A and generate two $N \times (r+p)$ Gaussian random matrices $\Omega^{(1)}$ and $\Omega^{(2)}$. Then, compute $Y = A\Omega^{(1)}$ and $Z = A^T\Omega^{(2)}$.

`do` $\ell = L, \cdots, 1$

`for` node i at level ℓ

`if` i is a leaf node,

1. $D_i = A_{t_i,t_i}$;
2. compute $\Phi_i = Y_i - D_i\Omega_i^{(1)}$, $\Theta_i = Z_i - D_i^T\Omega_i^{(2)}$;
3. compute the ID of $\Phi_i \approx U_i\Phi_i|_{\tilde{I}_i}$, $\Theta_i \approx V_i\Theta_i|_{\tilde{J}_i}$;
4. compute $\widehat{Y}_i = V_i^T\Omega_i^{(1)}$, $\widehat{Z}_i = U_i^T\Omega_i^{(2)}$;

`else`

1. store the generators $B_{i_1} = A(\tilde{I}_{i_1}, \tilde{J}_{i_2})$, $B_{i_2} = A(\tilde{I}_{i_2}, \tilde{J}_{i_1})$;
2. compute $\Phi_i = \begin{bmatrix} \Phi_{i_1}|_{\tilde{I}_{i_1}} - B_{i_1}\widehat{Y}_{i_2} \\ \Phi_{i_2}|_{\tilde{I}_{i_2}} - B_{i_2}\widehat{Y}_{i_1} \end{bmatrix}$, $\Theta_i = \begin{bmatrix} \Theta_{i_1}|_{\tilde{J}_{i_1}} - B_{i_2}^T\widehat{Z}_{i_2} \\ \Theta_{i_2}|_{\tilde{J}_{i_2}} - B_{i_1}^T\widehat{Z}_{i_1} \end{bmatrix}$;
3. compute the ID of $\Phi_i \approx U_i\Phi_i|_{\tilde{I}_i}$, $\Theta_i \approx V_i\Theta_i|_{\tilde{J}_i}$;
4. Compute $\widehat{Y}_i = V_i^T \begin{bmatrix} \widehat{Y}_{i_1} \\ \widehat{Y}_{i_2} \end{bmatrix}$, $\widehat{Z}_i = U_i^T \begin{bmatrix} \widehat{Z}_{i_1} \\ \widehat{Z}_{i_2} \end{bmatrix}$;

`end if`

`end for`

`end do`

It can be verified that the complexity of Algorithm 2 is $C_M + O(Nr^2)$, where C_M is the cost of multiplying A with (two) random matrices and r is the HSS rank of A. In practice, we can let $\Omega^{(1)} = \Omega^{(2)}$. FMM can be used to compute the sample matrices Y and Z, which only costs $O((r+p)N\log N)$ flops. For large matrices, FMM can be much faster than the plain matrix-matrix multiplications. If using FMM, the complexity of RSHSS is $O(Nr^2)$ flops; see reference [15]. This HSS construction algorithm in theory can be faster than the algorithm proposed in [13], which costs $O(N^2r)$ flops.

Most time of RSHSS is taken to compute the sample matrices. In the sequential case it takes about 80% of the construction time and about 30% in the fully parallel case; refer to Table III. In [16], it is proposed to use the subsampled random Fourier (SRFT) or Hadamard (SRHT) transforms

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

to compute the sample matrices. We do not use this technique in RSHSS or ADC2 because the SRFT would introduce complex matrices and the SRHT requires the dimension of matrix A to be powers of two. Furthermore, the construction algorithm is usually much faster than the HSS matrix multiplication algorithm; see the results in Table III. Note that if the SRHT is applicable, the complexity of RSHSS is also about $O(Nr^2)$ flops.

Another issue we want to mention is the accuracy of RSHSS. If the singular values of HSS blocks do not decay rapidly, the RS method may lose a bit of accuracy. A power scheme was proposed to improve the quality of sample matrices in [16], for instance, compute $Y = (AA^T)^q A\Omega$. We find it is very difficult to incorporate this technique into Algorithm 2, and moreover, using the power scheme would require about $2q + 1$ times as many operations as Algorithm 2. For accuracy, we choose a relatively large oversampling parameter p and try to compute the ID of sampled matrices as accurately as possible. In practice, we let $\epsilon = 1e\text{-}16$ in (11) to estimate the rank, and let the oversampling parameter $p = 10$. The number of used random vectors is usually larger than the HSS rank. This strategy in practice is quite robust and does not fail for any experiments during all our tests. Note that RSHSS still has a risk of losing accuracy, for example, if $r + p$ is smaller than the HSS rank in some rare cases.

4. IMPLEMENTATION DETAILS

The ADC algorithm consisted of three other algorithms: *the HSS construction*, *HSS matrix multiplication algorithms* and the standard DC algorithm. Almost all modern CPUs have multiple cores, and we implement the ADC algorithms in parallel to exploit the multicore architecture. We use OpenMP to implement these algorithms. This section introduces the parallel implementation details of these three algorithms.

4.1. Parallel RSHSS algorithm

As illustrated in Algorithm 2 and Section 4.1 of [15], the computations for different nodes at the same level can be performed simultaneously. We can exploit the parallelism of the HSS tree, and the computations for different nodes are carried out by different *processes*. Furthermore, the work for each node can also be carried out by *multi-threads* by calling a multithreaded BLAS library.

Recall that Algorithm 2 computes three or four generators for each node, D_i, U_i, V_i and B_i. Note that parent nodes do not have the generator D_i, U_i is computed from the row compression and V_i is from the column compression. The generators D_i and B_i are submatrices of the original matrix A and are Cauchy-like. Besides the number of flops, the running time of algorithms is also determined by the amount of data movements. To have good data locality, we store the same type of generators for nodes at the same level continuously. For example, we first store all the generators U_i at level ℓ, then the generators V_i and finally the generators B_i at level ℓ, for $\ell = L, \ldots, 1$. All the generators are stored continuously in one array, name it A_H, and the generators D_i are stored in the front part of A_H. This form of storage is good for HSS matrix multiplications (see Algorithm 3 in the following), where the computations follow the HSS tree level by level and the generators of the same type are used one after the other. For example, the computations of (18) use all the generators D_i at the bottom level, and so do the generators V_i.

Another point we want to mention is that the Cauchy-like matrices D_i and B_i are computed from its generators respectively, which are four vectors (Equation (3)). We find that recomputing the entries of D_i and B_i is usually faster than subtracting them from the original matrix A.

Our parallel version of RSHSS is similar to Algorithm 2. The only difference is that the `do-loop` in Algorithm 2 is replaced by the following process after some computation details are ignored.

```
par_for leaf node i,
    compute the Cauchy-like matrix D_i via its generators and store it in A_H;
end par_for
do ℓ = L, ··· , 1
```

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

`par_for` node i at level ℓ, compute its generator U_i from Φ_i and store U_i in A_H;
`end par_for`

`par_for` node i at level ℓ, compute its generator V_i from Θ_i and store V_i in A_H;
`end par_for`

`par_for` node i at level ℓ, compute the Cauchy-like matrix B_i and store it in A_H;
`end par_for`

`end do`

The abbreviation `par_for` stands for 'parallel for', which means the following computations can be carried out in parallel. In practice, we use (11) to estimate the HSS rank r, based on the partition of the original matrix A (Figure 1(a)). The matrix A is partitioned by letting all leaf nodes have roughly m rows and columns. Because $\{d_i\}$ and $\{\lambda_i\}$ are ordered increasingly, each partition of Q (8) can also be seen as a partition of interval $[\hat{a}, \hat{b}]$ which contains both $\{d_i\}$ and $\{\lambda_i\}$. For the partition in Figure 1(a), the interval $[\hat{a}, \hat{b}]$ is divided into four segments, and the first m_1 entries of $\{d_i\}$ and $\{\lambda_i\}$ lie in the first segment of $[\hat{a}, \hat{b}]$, the second m_2 entries lie in the second segment, and so on. The rank estimated by (11) depends on the distance of two segments, which is defined in Section 2.1. We use the distances of neighbouring segments to estimate rank and choose the *maximum* rank estimated by (11) as the HSS rank. In Figure 1(a), there are *three* pairs of neighbouring segments, and the estimated ranks are of H_1^{row}, H_3^{row} and H_4^{row}, respectively, and the maximum of them is used as an estimate of HSS rank r. If some eigenvalues are clustered, that is, the distance between I_i and I_{i+1} is small, the estimated rank by (11) may be too large to be useful. We use the following tricks to obtain a more reasonable estimate of r.

(1) If the distance between I_i and I_{i+1} is too small, we modify the partition of matrix A, that is, move the boundary forward or backward to let the distance be large. In our implementation, we modify the partition when the distance between I_i and I_{i+1} is less than $1e-10$, and the boundary is moved forward or backward by at most k(=5) rows and columns.
(2) If the computed rank by (11) is still too large, larger than 100, we fix the rank to be 100. (We find that HSS rank r is rarely larger than 100 in the tridiagonal DC algorithm.)

Note that these techniques are unfortunately lack of theoretical support, but they make the rank estimation method more useful and robust.

4.2. Hierarchically semiseparable matrix multiplication from the right

After an HSS matrix is represented in its HSS form, there exist fast algorithms for multiplying it with a vector in $O(Nr)$ flops [22, 43]. An HSS matrix multiplication algorithm has been introduced in [22] for $H \times A$, where H is an HSS matrix and A is a general matrix. For completeness, this section introduces the process of multiplying an HSS matrix with a general matrix from right, that is, compute $A \times H$. From Algorithm 3, it is easy to see that the HSS matrix multiplication algorithms are naturally parallelizable.

Table II. The parallelism of Algorithm 3 introduced by the HSS tree structure.

	Threads								
	1	3	5	7	9	11	13	15	16
Time(s)	11.34	4.06	2.63	2.07	1.73	1.51	1.40	1.31	1.20
Speedups	1.00	2.79	4.31	5.48	6.55	7.51	8.10	8.66	9.45

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

ALGORITHM 3. [HSS matrix multiplication from right] Assume that the HSS tree $\mathcal{T}$ is a full binary tree and there are $L+1$ levels, the root is at level 0 and the leaf nodes are at level L. Let j be the sibling of i. Let X be a $P \times N$ matrix and partition the columns of X as $X = [X_i]$, where $X_i = X(:, t_i)$, $i \in LN$ is a leaf node.

(1) `upsweep for` G_i

- `par_for` i at the bottom level, compute $G_i = X_i \cdot U_i$; `end par_for`
- `for` $\ell = L-1 : -1 : 1$
  `par_for` i at level ℓ, compute $G_i = \begin{bmatrix} G_{i_1} & G_{i_2} \end{bmatrix} \cdot U_i$; `end par_for`
- `end for`

(2) `downsweep for` F_i

- `par_for` i at the second top level, compute $F_i = G_j \cdot B_j$, $\begin{bmatrix} F_{i_1} \\ F_{i_2} \end{bmatrix} = F_i \cdot V_i^T$; `end par_for`
- `for` $\ell = 2 : L-1$
  `par_for` i at level ℓ, compute $F_i = G_j B_j + F_i$, $\begin{bmatrix} F_{i_1} \\ F_{i_2} \end{bmatrix} = F_i \cdot V_i^T$; `end par_for`
- `end for`

(3) compute X

- `par_for` i at the bottom level, compute

$$X_i = X_i D_i + F_i V_i^T; \tag{18}$$

- `end par_for`

All the computations for the nodes at the same level are independent of each other. Furthermore, almost all the operations are matrix–matrix multiplications, and we can take advantage of the highly optimized routine `DGEMM` in MKL. We explore both the parallelism in the HSS tree and the parallelism from the blas operations by using MKL. Table II shows the speedups of Algorithm 3 when only exploiting the parallelism in the HSS tree. The dimension of the HSS matrix is 10 000, which is defined in the same way as the matrix Q in Example 1, and we multiple it with a $10\,000 \times 10\,000$ random matrix via Algorithm 3. The times cost by Algorithm 3 are presented in the third row of Table II, and the compiled codes are linked to a sequential BLAS library. The results in Table II show that the scalability of Algorithm 3 is good. Some more numerical results are included in Example 2 in Section 5.

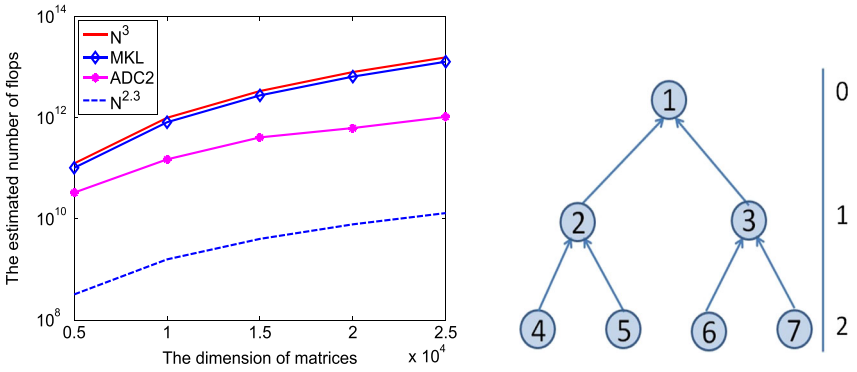
4.3. Accelerate the process of divide-and-conquer algorithm

The LAPACK routine `dstevd` implements a DC algorithm for symmetric tridiagonal matrices. It computes the eigenvalues and eigenvectors explicitly by calling `dlaed0`. The routine `dlaed0` solves each subproblem in a DC way. `dlaed1` called by `dlaed0` computes the eigendecomposition of the merged subproblem, and it calls `dlaed2` to deflate a diagonal matrix with rank-one modification and calls `dlaed3` to update the eigenvector matrix via matrix–matrix multiplications.

Our implementation has the same structure as LAPACK. We add the HSS techniques in the routine `dlaed1` and rename it `mdlaed1`. When the size of the deflated matrix is small, it calls `dlaed3` as usual. Otherwise, it calls `mdlaed3` to compute the eigenvalues and update the eigenvectors. In our implementation, we use the HSS matrix techniques when the size of the deflated matrix is larger than 2000. The routine `mdlaed3` is similar to `dlaed3`, and it computes the eigenvalues $\{\lambda_i\}$, the recomputed vector $\{\hat{u}_i\}$, $\gamma_i = \lambda_i - d_i$ (the distance between λ_i and d_i) and $\mu_i = d_{i+1} - \lambda_i$ (the distance between λ_i and d_{i+1}), for $i = 1, \ldots, n$. The secular equation in

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla



(a) Comparison of flops estimated by Vtune (b) The divide and conquer tree

Figure 2. The comparison of flops and the divide-and-conquer tree. (a) Comparison of flops estimated by Vtune and (b) The divide-and-conquer tree.

`mdlaed3` is solved in parallel by calling `dlaed4`. Then the eigenvector matrix of the diagonal matrix with rank-one modification is approximated by an HSS matrix, and the eigenvectors of the original matrix T are updated via fast HSS matrix multiplications [22]; see also Algorithm 3. Using the HSS matrix techniques to update the eigenvectors saves more flops because the complexity is reduced from $O(N^3)$ to $O(N^2 r)$.

The DC algorithm is also organized in a binary tree structure. Figure 2(b) shows the tree structure of DC algorithm. A big problem is recursively splitted into two small problems, and two small problems are merged together into a big one. The subproblems at the same level of DC tree can be solved in parallel. The subproblems at the bottom level are solved by using the QR algorithm in parallel, calling the LAPACK routine `dlasdq` in our implementation. For the problems at the other levels, we had tried to use the nested parallelism of OpenMP to exploit both the parallelism of DC tree and BLAS operations, but it did not give us any speedup increases. Furthermore, if using nested parallel computing, each thread would require more private memory to store the intermediate eigenvectors, and the memory cost would be greatly increased. Therefore, the problems above the bottom level of DC tree are solved sequentially, and we only exploit the parallelism of BLAS operations and HSS techniques.

As is well known, solving the secular equations costs about $O(N^2)$ operations. We further parallelize the process of solving the secular equations, which is inspired by the work in [44]. We simply add OMP PARALLEL DO directives in `mdlaed3` when calling `dlaed4`. Our parallel implementation is simpler than that in [44]. In this paper, we use OpenMP and follow the fork-join model, while it followed a task-flow model in [44] and used a dynamic runtime system to schedule the tasks. Comparing the numerical results in [44] with those in the next section, we can see that the rank-structured matrix techniques are good in the case when there are few deflations and that the task-flow model used in [44] is good in the case when there more deflations. The advantage of the task-flow model is that it introduces a huge level of parallelism through a fine task granularity and that the tasks are scheduled by a runtime system, some synchronization barriers are removed. As the algorithm in [44] still uses plain matrix–matrix multiplications to update the eigenvectors, when the deflations are few, its advantage decreases as the dimensions of matrices increase. Therefore, a good research direction is to combine the rank-structured matrix techniques with the task-flow model.

5. NUMERICAL RESULTS

All the results are obtained on a server with 128 GB memory and an Intel(R) Xeon(R) CPU E5-2670, which has two sockets, eight cores per socket, and 16 cores in total. The codes are written in Fortran 90. For compilation, we used Intel fortran compiler (`ifort`) and the optimization flag `-O2 -openmp`, and then linked the codes to Intel MKL (composer_xe_2013.0.079).

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

Table III. The execution time of hierarchically semiseparable algorithms in seconds.

Method		Threads								
		1	3	5	7	9	11	13	15	16
SRRSC	Const	2.29	0.84	0.52	0.41	0.34	0.33	0.32	0.31	0.31
	Mult	4.75	1.87	1.32	1.08	0.96	0.97	0.89	0.85	0.85
RSHSS	DGEMM	2.24	0.76	0.46	0.34	0.30	0.25	0.21	0.19	0.16
	Const	2.85	1.07	0.75	0.63	0.57	0.53	0.48	0.45	0.42
	Mult	8.04	2.98	1.94	1.54	1.29	1.14	1.09	1.02	0.95

Example 2

In this example, we use the matrix defined in Example 1 to show the scalability of the HSS construction and matrix multiplication algorithms when implemented in parallel by using multi-threading. The dimension of this matrix is 10 000. The row dimensions of HSS blocks for the leaf nodes are around 200. The scalability of the HSS construction algorithm based on SRRSC and RSHSS are tested, and the results are shown in Table III. The results for Algorithm 3 are also included. The elapsed times of HSS constructions are shown in the rows denoted by `Const`, and the times of HSS multiplications are included in those denoted by `Mult`. The row denoted by `DGEMM` in Table III shows the times of computing the sample matrices Y and Z.

The results in Table III are obtained by letting `OMP_NUM_THREADS` and `MKL_NUM_THREADS` equal to $1, 3, 5, \cdots, 15$ and 16. From the results, we can see that the HSS construction algorithm is usually faster than the HSS multiplication algorithm. For the RSHSS algorithm, we let p equal to 10 and the estimated rank by (11) is 79, which is larger than 57, the HSS rank computed by SRRSC. Most ranks of the HSS blocks are around 40. The HSS matrix multiplications for RSHSS is slower than those for SRRSC, because the ranks of HSS blocks computed by RSHSS are usually larger than those computed by SRRSC. From the results in Table III, we can see that our parallel implementation achieves good speedups.

Example 3

For several classes of matrices [45], few or no eigenvalues are deflated in the DC algorithm. Some of such matrices include the Clement-type, Legendre-type, Laguerre-type, Hermite-type and Toeplitz-type matrices, which are defined as follows. We use these matrices to show the performance of the ADC algorithms.

The Clement-type matrix [45] is given by

$$T = \text{tridiag}\begin{pmatrix} \sqrt{n} & \sqrt{2(n-1)} & \sqrt{(n-1)2} & \sqrt{n} \\ 0 & 0 & \cdots & 0 & 0 \\ \sqrt{n} & \sqrt{2(n-1)} & \sqrt{(n-1)2} & \sqrt{n} \end{pmatrix}, \tag{19}$$

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

The Legendre-type matrix is defined as in [45, 46],

$$T = \text{tridiag}\begin{pmatrix} 2/\sqrt{3\cdot 5} & 3/\sqrt{5\cdot 7} & n/\sqrt{(2n-1)(2n+1)} \\ 0 & 0 & \cdots & 0 \\ 2/\sqrt{3\cdot 5} & 3/\sqrt{5\cdot 7} & n/\sqrt{(2n-1)(2n+1)} \end{pmatrix}, \tag{20}$$

where the off-diagonal entries are $i/\sqrt{(2i-1)(2i+1)}, i = 2, \ldots, n$.

The Laguerre-type matrix is defined as in [45],

$$T = \text{tridiag}\begin{pmatrix} 2 & 3 & n-1 & n \\ 3 & 5 & \cdots & 2n-1 & 2n+1 \\ 2 & 3 & n-1 & n \end{pmatrix}. \tag{21}$$

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

The Hermite-type matrix is given as in [45],

$$T = \text{tridiag}\begin{pmatrix} \sqrt{1} & \sqrt{2} & \sqrt{n-2} & \sqrt{n-1} \\ 0 & 0 & \dots & 0 & 0 \\ \sqrt{1} & \sqrt{2} & \sqrt{n-2} & \sqrt{n-1} \end{pmatrix}. \tag{22}$$

The Toeplitz-type matrix is a symmetric tridiagonal matrix with diagonals 2 and off-diagonal entries 1. In this example, we compare ADCs with DC in Intel MKL with `OMP_NUM_THREADS`=16. The speedups of ADC1 and ADC2 over DC are similar, and the results are respectively reported in Tables IV and V; see also Figure 3. Because ADCs require fewer flops than the standard DC, they can achieve even better speedups for larger matrices. For example, when the dimension of Toeplitz-type matrix increases from $25k$ to $40k$, the speedup of ADC2 over DC increases from 5.79 to 8.06. During our experiments, HSS techniques are only used when the size of the current secular equation is larger than 2000, which is a parameter depending on the computer architecture, compiler and the optimized BLAS library. The row dimensions of the HSS blocks for the leaf nodes are also around 200, and 16 threads are used.

Table IV. The speedups of ADC1 compared with Intel MKL.

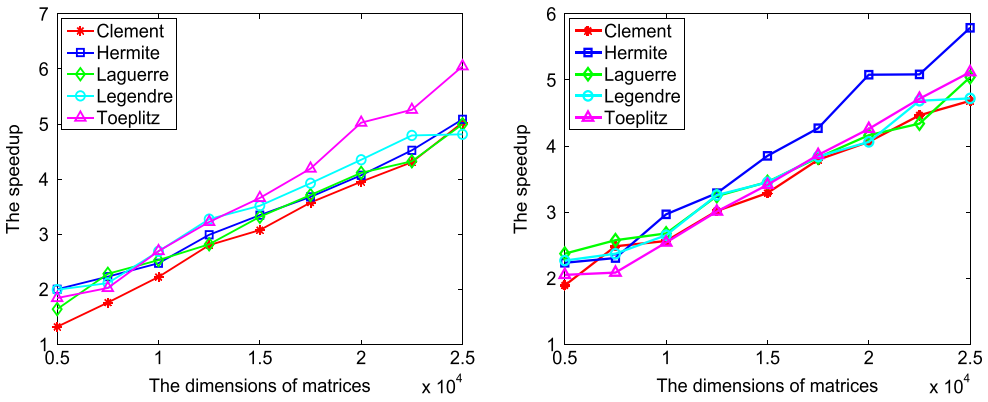
	Dim								
Matrix	$5k$	$7.5k$	$10k$	$12.5k$	$15k$	$17.5k$	$20k$	$22.5k$	$25k$
Clement	1.32x	1.76x	2.22x	2.80x	3.10x	3.57x	3.95x	4.31x	5.03x
Legendre	1.99x	2.11x	2.69x	3.27x	3.52x	3.92x	4.35x	4.79x	4.81x
Laguerre	1.64x	2.28x	2.53x	2.82x	3.32x	3.71x	4.11x	4.32x	5.00x
Hermite	2.00x	2.23x	2.47x	2.99x	3.35x	3.68x	4.07x	4.52x	5.08x
Toeplitz	1.84x	2.02x	2.69x	3.22x	3.66x	4.19x	5.02x	5.26x	6.05x

k denotes one thousand.

Table V. The speedups of ADC2 compared with Intel MKL.

	Dim								
Matrix	$5k$	$7.5k$	$10k$	$12.5k$	$15k$	$17.5k$	$20k$	$22.5k$	$25k$
Clement	2.05x	2.08x	2.54x	3.01x	3.41x	3.87x	4.26x	4.71x	5.12x
Legendre	2.27x	2.37x	2.65x	3.25x	3.45x	3.84x	4.06x	4.68x	4.72x
Laguerre	1.89x	2.48x	2.56x	3.01x	3.28x	3.79x	4.06x	4.46x	4.68x
Hermite	2.37x	2.58x	2.68x	3.33x	3.46x	3.82x	4.16x	4.33x	5.04x
Toeplitz	2.23x	2.31x	2.97x	3.29x	3.85x	4.27x	5.07x	5.08x	5.79x

k denotes one thousand.



(a) ADC1 (b) ADC2

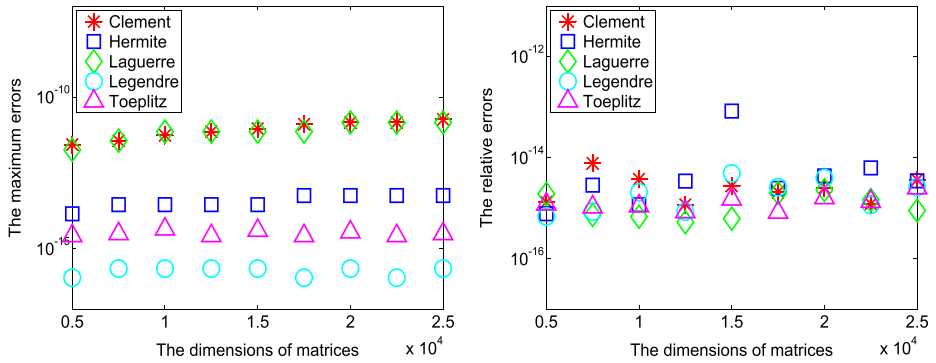
Figure 3. The weak scalability of ADC1 and ADC2.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

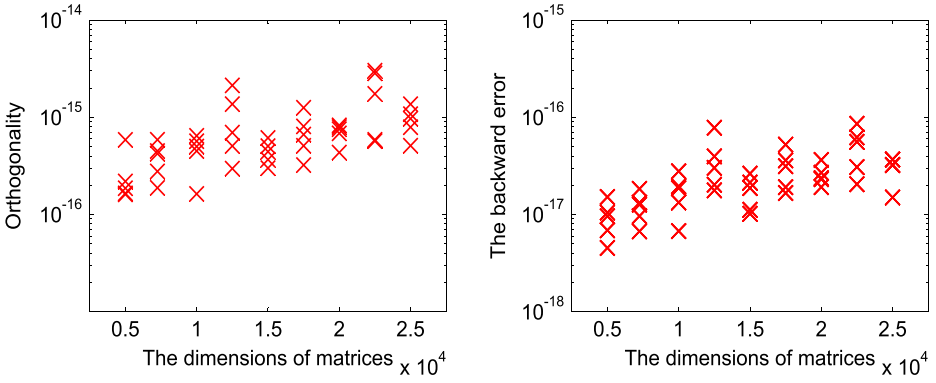
The DC algorithm is relatively complex, there are deflations and the secular equations are solved via iterative methods, and it is difficult to count the total number of flops cost by DC or ADCs by hand. We use some tools based on event-based sampling technology to estimate the floating point operations. PAPI [47] and Intel Vtune Amplifier XE [48] are popular performance analysis tools, which can make reasonable estimates. Figure 2(a) shows the comparisons of flops costed by ADC2 and DC in MKL. These results are for the Toeplitz-type matrices with different dimensions. From it, we can see that DC requires nearly $O(N^3)$ flops, while ADC2 requires much fewer flops. Figure 4(a) and 4(b) shows the maximum errors and maximum relative errors of the eigenvalues computed by ADC1 compared with those by DC, respectively. From the results, we can see that the computed eigenvalues by ADC1 nearly have the same accuracy as those computed by MKL. Note that the results for relative error are included here, but the DC algorithms in general are not guaranteed to have high relative accuracy.

The results for the orthogonality of the computed eigenvectors are shown in Figure 5(a), which are defined as $\frac{\|I-QQ^T\|}{N}$. Figure 5(b) shows the results for the backward error of ADC2, computed as $\frac{\|T-Q\Sigma Q^T\|}{\|T\|\times N}$. While ADC2 is a little less accurate than ADC1 but ADC2 can also be used reliably for applications, the orthogonality of the computed eigenvectors by ADC2 is about $1e$-12 and the maximum error of the computed eigenvalues by ADC2 compared with those by Intel MKL is about $1e$-14. One advantage of ADC2 over ADC1 is that it requires fewer flops when FMM or SRHT is applicable, which will be done in the future work.



(a) Max. error (b) Max. Relative error

Figure 4. Errors of the eigenvalues computed by ADC1 compared with those by MKL. (a) Max. error. (b) Max. relative error.

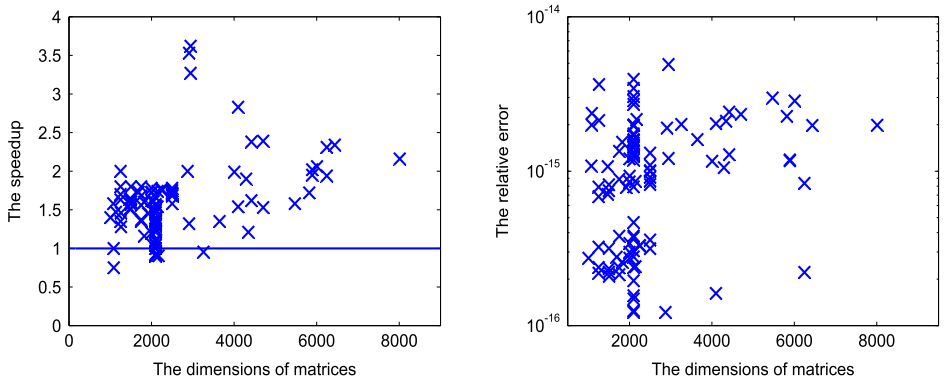


(a) Orthogonality $\frac{\|I-QQ^T\|}{N}$ (b) The backward error $\frac{\|T-Q\Lambda Q^T\|}{\|T\|\times N}$

Figure 5. The correctness of the eigenvectors computed by ADC2. (a) Orthogonality $\frac{\|I-QQ^T\|}{N}$. (b) The backward error $\frac{\|T-Q\Sigma Q^T\|}{\|T\|\times N}$.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla



(a) The speedups of ADC2 compared with MKL (b) The relative errors of computed eigenvalues

Figure 6. The results for matrices in `stetester`. (a) The speedups of ADC2 computed with MKL. (b) The relative errors of computed eigenvalues.

We further use all the matrices in the LAPACK `stetester` [45] with dimensions larger than 1000 to test ADC2. Figure 6 shows the speedups of ADC2 over DC in MKL and the relative errors of the eigenvalues computed by ADC2 compared with those by MKL. The results show that for almost all matrices, ADC2 is faster than the DC implementation in MKL and that the computed eigenvalues are highly accurate compared with those computed by DC in MKL. The experiments are performed by letting `OMP_NUM_THREADS` and `MKL_NUM_THREADS` equal to 16. For some rare matrices, ADC2 is a little slower than `dstevd` in MKL but never slower by more than 0.8e-02 s. Note that the HSS techniques are only used when the size of secular equation is larger than 2000. For the matrices with dimensions from 1000 to 2000, the speedups of ADC2 over DC in MKL is due to that ADC2 computes the bottom subproblems of the DC tree and the secular equations in parallel.

6. CONCLUSIONS

In this paper, two accelerated tridiagonal DC algorithms are proposed by using the HSS matrix techniques. One uses SRRSC for the HSS construction and the other uses a randomized HSS construction algorithm that is first introduced in [15]. For the later one, we propose a method to estimate the HSS rank by using the function approximation theory. The main point is using the rank-structured matrix techniques to update the eigenvectors. Roughly speaking, the worst case complexity of ADCs is reduced to $O(N^2 r)$ for an $N \times N$ symmetric tridiagonal matrix instead of $O(N^3)$, where r is a modest number that depends on the property of the tridiagonal matrix. We implement ADCs in parallel, including the HSS construction and HSS matrix multiplication algorithms, and compare them with the multithreaded Intel MKL library. For some matrices of large dimensions with few deflations, our ADC algorithms can be more than six times faster than the DC algorithm in MKL.

ACKNOWLEDGEMENTS

The authors would like to thank Ming Gu for valuable suggestions and Ziyang Mao, Lihua Chi, Yihui Yan, Xu Han, Xinbiao Gan and Qingfeng Hu for some helpful discussions. The authors also thank the referee for their valuable comments, which greatly improve the presentation of this paper. This work is partial supported by the National Natural Science Foundation of China (No. 11401580, 91530324, 91430218, 61133005, and 61402495) and the 863 Program of China under grant 2012AA01A301.

REFERENCES

1. Golub G, Loan C. *Matrix Computations* (3rd edn). The Johns Hopkins University Press: Baltimore, MD, 1996.
2. Parlett B. *The Symmetric Eigenvalue Problem*. SIAM: Philadelphia, 1998.
3. Dhillon I, Parlett B. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and its Applications* 2004; **387**:1–28.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

4. Cuppen JJM. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik* 1981; **36**:177–195.
5. Gu M, Eisenstat S. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM Journal on Matrix Analysis and Applications* 1995; **16**:172–191.
6. Demmel J, Marques O, Parlett B, Vömel C. Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers. *Technical Report*, University of Tennessee: Knoxville, 2007.
7. Demmel J. *Applied Numerical Linear Algebra*. SIAM: Philadelphia, 1997.
8. Tisseur F, Dongarra J. A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures. *SIAM Journal on Scientific Computing* 1999; **20**:2223–2236.
9. Rutter J. Computer Science Division. *Technical Report CSD-94-799*, University of California at Berkeley, 1994.
10. Chandrasekaran S, Gu M, Pals T. A fast ULV decomposition solver for hierarchical semiseparable representations. *SIAM Journal on Matrix Analysis and Applications* 2006; **28**:603–622.
11. Xia J, Chandrasekaran S, Gu M, Li X. Fast algorithm for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications* 2010; **17**:953–976.
12. Bunch J, Nielsen C, Sorensen D. Rank one modification of the symmetric eigenproblem. *Numerische Mathematik* 1978; **31**:31–48.
13. Li S, Gu M, Cheng L, Chi X, Sun M. An accelerated divide-and-conquer algorithm for the bidiagonal SVD problem. *SIAM Journal on Matrix Analysis and Applications* 2014; **35**:1038–1057.
14. Gu M, Xia J. *A multi-structured superfast Toeplitz solver*, 2009. preprint.
15. Martinsson P. A fast randomized algorithm for computing a hierarchically semi-separable representation of a matrix. *SIAM Journal on Matrix Analysis and Applications* 2011; **32**(4):1251–1274.
16. Halko N, Martinsson P, Tropp J. Finding structure with randomness probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review* 2011; **53**:217–288.
17. Liberty E, Woolfe F, Martinsson P, Rokhlin V, Tygert M. Randomized algorithms for the low-rank approximation of matrices. *PNAS* 2007; **104**:20167–20172.
18. Carrier J, Greengard L, Rokhlin V. A fast adaptive multipole algorithm for particle simulations. *SIAM Journal on Scientific and Statistical Computing* 1988; **9**:669–686.
19. Greengard L, Rokhlin V. A fast algorithm for particle simulations. *Journal of Computational Physics* 1987; **73**: 325–348.
20. Braess D, Hackbusch W. Approximation of 1/x by exponential sums in $[1, \infty)$. *IMA Journal of Numerical Analysis* 2005; **25**:685–697.
21. Wang S, Li X, Xia J, Situ Y, Hoop M. Efficient scalable algorithms for hierarchically semiseparable matrices. *SIAM Journal on Scientific Computing* 2013; **35**:C519–C544.
22. Lyons W. Fast algorithms with applications to PDEs. *Ph.D. Thesis*, University of California, Santa Barbara, 2005.
23. Gu M, Eisenstat S. A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem. *SIAM Journal on Matrix Analysis and Applications* 1994; **15**:1266–1276.
24. Löwner K. Über monotone matrixfunktionen. *Mathematische Zeitschrift* 1934; **38**:177–216.
25. Hackbusch W. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: introduction to $\mathcal{H}$-matrices. *Computing* 1999; **62**:89–108.
26. Hackbusch W, Khoromskij B. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part II: application to multi-dimensional problems. *Computing* 2000; **64**:21–47.
27. Hackbusch W, Börm S. Data-sparse approximation by adaptive $\mathcal{H}^2$-matrices. *Computing* 2002; **69**:1–35.
28. Hackbusch W, Khoromskij B, Sauter S. On $\mathcal{H}^2$-matrices. In *Lecture on Applied Mathematics*, Bungartz H, Hoppe RHW, Zenger C (eds). Springer: Berlin, 2000; 9–29.
29. Eidelman Y, Gohberg I. On a new class of structured matrices. *Integral Equations and Operator Theory* 1999; **34**:293–324.
30. Vandebril R, Van Barel M, Mastronardi N. *Matrix Computations and Semiseparable Matrices, Volume I: Linear Systems*. Johns Hopkins University Press: Baltimore, Maryland, 2008.
31. Chandrasekaran S, Dewilde P, Gu M, Pals T, Sun X, van der Veen AJ, White D. Fast stable solvers for sequentially semi-separable linear systems of equations and least squares problems. *Technical Report*, University of California: Berkeley, CA, 2003.
32. Chandrasekaran S, Dewilde P, Gu M, Pals T, Sun X, van der Veen AJ, White D. Some fast algorithms for sequentially semiseparable representation. *SIAM Journal on Matrix Analysis and Applications* 2005; **27**:341–364.
33. Chandrasekaran S, Gu M, Pals T. Fast and stable algorithms for hierarchically semi-separable representations. *Technical Report*, University of California: Berkeley, CA, 2004.
34. Starr P. On the numerical solution of one-dimensional integral and differential equations. *Ph.D. Thesis*, Department of Computer Science, Yale University, New Haven, CT, 1991.
35. Xia J, Gu M. Robust approximate Choleksy factorization of rank-structured symmetric positive definite matrices. *SIAM Journal on Matrix Analysis and Applications* 2010; **31**:2899–2920.
36. Cheng H, Gimbutas Z, Martinsson P, Rokhlin V. On the compression of low rank matrices. *SIAM Journal on Scientific Computing* 2005; **26**:1389–1404.
37. Martinsson P, Rokhlin V, Tygert M. A randomized algorithm for the approximation of matrices. *Applied and Computational Harmonic Analysis* 2011; **30**:47–68.
38. Gu M, Eisenstat S. Efficient algorithms for computing a strong-rank revealing QR factorization. *SIAM Journal on Scientific Computing* 1996; **17**:848–869.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla

39. Pan C. On the existence and computation of rank revealing LU factorizations. *Linear Algebra and its Applications* 2000; **316**:199–222.
40. Goreinov S, Tyrtyshnikov E, Zamarashkin N. Theory of pesudo-skeleton matrix approximations. *Linear Algebra and its Applications* 1997; **261**:1–21.
41. Stewart G. Four algorithms for the efficient computation of truncated pivoted QR approximations to a sparse matrix. *Numerische Mathematik* 1999; **83**:313–323.
42. Chan T, Hansen P. Some applications of the rank-revealing QR factorization. *SIAM Journal on Scientific and Statistical Computing* 1992; **13**:727–741.
43. Chandrasekaran S, Dewilde P, Gu M, Lyons W, Pals T. A fast solver for HSS representations via sparse matrices. *SIAM Journal on Matrix Analysis and Applications* 2006; **29**:67–81.
44. Pichon G, Haidar A, Faverge M, Kurzak J. Divide and conquer symmetric tridiagonal eigensolver for multicore architectures. *In the 29th IEEE International Parallel and Distributed Processing Symposium*, Hyderabad, India, 2015; 51–60.
45. Marques O, Voemel C, Demmel J, Parlett B. Algorithm 880: A testing infrastructure for symmetric tridiagonal eigensolvers. *ACM Transactions on Mathematical Software* 2008; **35**:8:1–8:13.
46. Abramowitz M, Stegun I. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables* (9th edn). Dover: New York, 1965.
47. Browne S, Dongarra J, Garner N, Ho G, Mucci P. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications* 2000; **14**:189–204.
48. *Intel VTune Amplifier*. (Available from: https://software.intel.com/en-us/intel-vtune-amplifier-xe) [Accessed on 30 December 2015].

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/nla