Received: 5 July 2016 | Revised: 15 October 2017 | Accepted: 8 November 2017

DOI: 10.1002/nla.2139

RESEARCH ARTICLE

WILEY

# High-performance direct algorithms for computing the sign function of triangular matrices

Vadim Stotland[1] | Oded Schwartz[2] | Sivan Toledo[1]

[1]Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel

[2]The Benin School of Computer Science and Engineering, Hebrew University, Jerusalem, Israel

**Correspondence**
Sivan Toledo, Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel.
Email: sivan.toledo@gmail.com

**Funding information**
Israel Science Foundation, Grant/Award Number: 863/15, 1878/14, 1901/14; Ministry of Science and Technology, Grant/Award Number: 3-10891; Einstein Foundation and the Minerva Foundation; Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI); United States-Israel Binational Science Foundation (BSF); HUJI Cyber Security Research Center

**Summary**

Algorithms and implementations for computing the sign function of a triangular matrix are fundamental building blocks for computing the sign of arbitrary square real or complex matrices. We present novel recursive and cache-efficient algorithms that are based on Higham's stabilized specialization of Parlett's substitution algorithm for computing the sign of a triangular matrix. We show that the new recursive algorithms are asymptotically optimal in terms of the number of cache misses that they generate. One algorithm that we present performs more arithmetic than the nonrecursive version, but this allows it to benefit from calling highly optimized matrix multiplication routines; the other performs the same number of operations as the nonrecursive version, suing custom computational kernels instead. We present implementations of both, as well as a cache-efficient implementation of a block version of Parlett's algorithm. Our experiments demonstrate that the blocked and recursive versions are much faster than the previous algorithms and that the inertia strongly influences their relative performance, as predicted by our analysis.

**KEYWORDS**

blocked matrix algorithms, cache-efficient algorithms, communication-efficient algorithms, matrix functions, partitioned matrix algorithms

## 1 | INTRODUCTION

The sign of a square complex matrix $A$ is defined by extending the following scalar function:

$$\operatorname{sign}(z) = \operatorname{sign}(x + iy) = \begin{cases} 1 & x > 0 \\ -1 & x < 0 \end{cases}$$

to matrices. For a diagonalizable matrix $A = ZDZ^{-1}$, the sign can be defined by applying sign($z$) to the eigenvalues of $A$, as follows:

$$\operatorname{sign}(A) = Z \begin{bmatrix} \operatorname{sign}(d_{11}) & & & \\ & \operatorname{sign}(d_{22}) & & \\ & & \ddots & \\ & & & \operatorname{sign}(d_{nn}) \end{bmatrix} Z^{-1} .$$

The definition can be extended to the nondiagonalizable case in a variety of equivalent ways (see section 1.2 in the work of Higham et al.[1]). From here on, we use the term *function* to refer to a mapping that satisfies these equivalent definitions. The matrix sign function is not defined when $A$ has purely imaginary eigenvalues (and is clearly ill conditioned on matrices with eigenvalues whose real part is close to zero).

Originally, the interest in the sign function arose because it can be used to solve algebraic equations in control theory, including the Riccati,[2] Sylvester, and Lyapunov equations. Another application that arose soon after is in divide-and-conquer eigensolvers.[3–5]

For many years, the only algorithms for computing the sign function were iterative ones, variants of Newton and Padé iterations (for example, see other works[1,2,5]). Some of these, such as the beautifully simple Newton iteration due to Roberts,[2] which repeatedly averages the matrix and its inverse, have been found to experience slow initial convergence. These algorithms perform between $2n^3$ and $4n^3$ arithmetic operations per iteration, where $n$ is the dimension of $A$.

More recently, Higham discovered a direct way to compute the sign function (see chapter 5 in the work of Higham et al.[1]). This approach is based on a much older and much more general technique due to Parlett, which is unstable when used in its original form.[6] In this approach, we first compute a Schur decomposition of $A = QTQ^*$, where $T$ is upper triangular and $Q$ is unitary, then compute $U = \text{sign}(T)$, and finally form $\text{sign}(A) = QUQ^*$.

In this study, we focus on *direct* (as opposed to iterative) algorithms for computing the sign function of a triangular matrix, which can be used as a building block in algorithms for general matrices. Parlett's substitution-type algorithm can compute many functions of triangualr matrices.[6] The algorithm exploits the equation $UT = TU$ that any function $U$ of $T$ satisfies and the fact that if $T$ is triangular, so is $U$. Parlett's technique breaks down when $T$ has repeated eigenvalues (and becomes unstable when it has clustered eigenvalues).

Higham's improved version, which we refer to as the *Parlett–Higham* algorithm, applies only to the sign function and avoids repeated-eigenvalue breakdowns (see algorithm 5.5 in the work of Higham et al.[1]). An implementation of this algorithm, called `signm`, is part of the Matrix Function Toolbox in Matlab. A more generic way to avoid breakdowns and instability in Parlett's algorithm is to reorder the Schur form so that eigenvalues are clustered along the diagonal of $T$ and to apply a block version of Parlett's substitution.[7] This approach requires some other way to compute the sign of diagonal blocks of $T$; the off-diagonal blocks are computed by solving Sylvester equations. We refer to this method as the *Parlett–Sylvester* technique. The algorithm that computes the sign of diagonal blocks that must be able to cope with a clustered spectrum (up to the case of repeated eigenvalues); Parlett's method cannot usually be applied to these blocks. However, in the case of the sign function, clustering the eigenvalues according to their sign provides a trivial way to construct the two diagonal blocks of $U$: one is identity $I$ and the other a negated identity $-I$, typically of different dimensions.

We also note that the iterative Newton and Padé approximations can also be combined with the Schur decomposition. In this hybrid approach, one computes the Schur decomposition and then iterates on $T$ to obtain $U$; the potential advantage of this approach lies in the fact that the main ingredients of these iterations, matrix multiplication and matrix inversion, are considerably cheaper on triangular matrices than on general square ones and that the iterations preserve the triangular structure.[*]

**Our Contributions** This paper presents high-performance direct algorithms for computing the sign of a triangular matrix. To obtain high performance, we take two measures. First, we choose whether to use the Parlett–Higham substitution algorithm or the Parlett–Sylvester algorithm, by estimating the amount of work each of them would require. We show that their complexity may differ asymptotically; hence, choosing the right one is essential. Second, we reorder the operations that our algorithms perform, so as to reduce cache misses and interprocessor communication. The reordering techniques apply both to the Parlett–Higham and to the Parlett–Sylvester algorithms. The reordering that we apply to the Parlett–Higham algorithm closely resembles the reordering proposed by Deadman et al. for a variant of Parlett's method that computes the matrix square root.[8]

The description and implementation of the algorithms in this study apply to both triangular real and complex matrices, as well as to quasitriangular real matrices, which are used in the so-called real Schur form (a real factorization for real matrices with complex eigenvalues; see Section 2.1 below).

**Paper Organization** The rest of the paper is organized as follows. Section 2 presents the basic Parlett recurrence for functions of triangular matrices and Higham's stabilized version for the sign function and the Parlett–Sylvester approach. Section 2.1 analyzes the number of arithmetic operations that the two approaches perform and shows that the Parlett–Sylvester is less efficient when the inertia is balanced but much more efficient when it is not. Section 3 presents lower bounds on the asymptotic number of cache misses that these algorithms generate. Section 4 presents recursive cache-efficient variants of the Parlett–Higham algorithm, which are asymptotically optimal by the previous section.

[*] This algorithmic approach was discovered by one of the reviewers of this article, not by the authors.

Section 5 shows that the new algorithms and our implementation of the Parlett–Sylvester algorithm are indeed fast and that their performance in practice matches our theoretical predictions. We presents our conclusions in Section 6.

## 2 | BACKGROUND

Any matrix function $F = \phi(A)$ commutes with its argument, $AF = FA$. The function $F = \phi(T)$ of an upper triangular matrix $T$ is also upper triangular. Parlett used these facts to construct a substitution-type algorithm to compute $F = \phi(T)$. By rearranging the expression for the $i, j$ element in the product $TF = FT$ as follows:

$$\sum_{k=i}^{j} t_{ik} f_{kj} = \sum_{k=i}^{j} f_{ik} t_{kj} \ ,$$

where $t_{ik}$ is the $i, k$ element of $T$ and so on, we can almost isolate $f_{ij}$ as follows:

$$(t_{ii} - t_{jj}) f_{ij} = f_{ii} t_{ij} - f_{jj} t_{ij} + \sum_{k=i+1}^{j-1} \left( f_{ik} t_{kj} - t_{ik} f_{kj} \right) \ . \tag{1}$$

This allows us to obtain the value of $f_{ij}$ as a function of $f_{ik}$ for $k < j$ and $f_{kj}$ for $k < j$. These equations do not constrain the diagonal elements of $F$ (the equations are $t_{ii} f_{ii} = f_{ii} t_{ii}$), but it is easy to see that they must satisfy $f_{ii} = \phi(t_{ii})$. The complete algorithm is shown in Algorithm 1.

**Algorithm 1** Parlett's substitution algorithm to compute a function of a lower triangular matrix $T \in \mathbb{C}^{n \times n}$ with distinct diagonal elements (eigenvalues).

```
for $i = 1 : n,\ f_{ii} = \phi(t_{ii})$
for $j = 2 : n$
    for $i = j-1 : -1 : 1$
        $f_{ij} = \frac{1}{t_{ii}-t_{jj}} \left( t_{ij} \left( f_{ii} - f_{jj} \right) + \left( \sum_{k=i+1}^{j-1} f_{ik} t_{kj} - t_{ik} f_{kj} \right) \right)$
    end
end
```

Clearly, the algorithm breaks down if $T$ has repeated eigenvalues ($t_{ii} = t_{jj}$ for some $i$ and $j$). Pairs of nearby but unequal eigenvalues (small $|t_{ii} - t_{jj}|$) tend to cause growth in $F$ because of divisions by small quantities. In some cases, this is related to the ill conditioning of $F$, but not always. In some cases, the growth is associated with an instability in the algorithm rather than with poor conditioning.

One way to address this issue, at least partially, is to partition $F$ and $T$ into blocks and write the corresponding block-matrix multiplication equations that $TF = FT$ defines (see the work of Parlett,[9] cited by Higham et al.[1]). The partitioning is into square diagonal blocks and possibly rectangular off-diagonal blocks. In this version, we cannot isolate off-diagonal blocks $F_{ij}$ because they do not necessarily commute with diagonal blocks of $T$, so the equations that define $F_{ij}$ are not simple substitution-type equations, but rather, they form Sylvester equations, as shown in Algorithm 2.

**Algorithm 2** The Parlett-Sylvester substitution algorithm to compute a matrix function given a partitioning of the row and column indexes into $m$ blocks.

```
for $i = 1 : m,\ F_{ii} = \phi(T_{ii})$
for $j = 2 : m$
    for $i = j-1 : -1 : 1$
        Solve for $F_{ij}$ the Sylvester equation
        $T_{ii} F_{ij} - F_{ij} T_{jj} = F_{ii} T_{ij} - T_{ij} F_{jj} + \sum_{k=i+1}^{j-1} \left( F_{ik} T_{kj} - T_{ik} F_{kj} \right)$
    end
end
```

Here too, the equations that drive the algorithm say nothing about diagonal blocks $F_{ii}$, so they must be computed in some other way; we discuss this later. The Sylvester equation for $F_{ij}$ is singular if $T_{ii}$ and $T_{jj}$ have common eigenvalues and are ill conditioned if they have nearby eigenvalues. Hence, for this method to work well, the partitioning of $F$ and $T$ needs to be such that different diagonal blocks of $T$ share no common eigenvalues and ideally do not have nearby eigenvalues.

Davies et al. proposed a framework that uses this approach for essentially any function $\phi$.[7] Their framework begins by clustering the eigenvalues of $T$. The clusters are made as small as possible under the condition that they are well separated. Note that if eigenvalues are highly clustered, the framework may end up with a single large cluster. This is undesirable from the computational complexity perspective but avoids numerical problems. The framework then uses an algorithm by Bai et al.[10] to reorder $T$ unitarily so as to make the eigenvalues in each cluster adjacent, $T = Q\tilde{T}Q^*$. The Parlett–Sylvester algorithm then computes the function $\tilde{F} = \phi(\tilde{T})$, which is transformed back into the function $F$ of $T$, $F = Q\tilde{F}Q^*$. The diagonal blocks of $\tilde{F}$ cannot be computed by a Parlett recurrence because the diagonal blocks of $\tilde{T}$ have clustered or repeated eigenvalues. Davies et al.[7] proposed that a Padé approximation be used to compute these blocks. The Padé approach is very general but becomes very expensive if diagonal blocks are large.

However, in the special case of the sign function, we can partition the eigenvalues by the sign of their real part. In this case, the functions of the two resulting diagonal blocks of $\tilde{T}$ are trivial: the identity is the sign of the block with the positive eigenvalues (right half of the complex plane), and a negated identity is the sign of the block with the negative eigenvalues (left half of the plane); see section 5.2 in the work of Higham et al.[1]

Higham et al. also proposed another specialization of Parlett's method to the sign function (see algorithm 5.5 in the work of Higham et al.[1]). The matrix sign $U = \text{sign}(T)$ satisfies another matrix equation, $U^2 = I$. We can again rearrange the expression for the $i, j$ element of $I$ in this expression ($i < j$), as follows:

$$\sum_{k=i}^{j} u_{ik} u_{kj} = 0$$

so as to isolate the following:

$$u_{ij} = -\frac{\sum_{k=i+1}^{j-1} u_{ik} u_{kj}}{u_{ii} + u_{jj}} . \qquad (2)$$

If $u_{ii}$ and $u_{jj}$ have the opposite sign (a 1 and a $-1$), this expression breaks down. However, in this case, the signs of $t_{ii}$ and $t_{jj}$ are also different, so the plain Parlett recurrence (Equation 1) can be safely used. When both $u_{ii} + u_{jj} \neq 0$ and $t_{ii} - t_{jj} \neq 0$, we prefer to compute $u_{ij}$ using Equation 2 rather than using Equation 1 because $|u_{ii} + u_{jj}| = 2$, whereas $|t_{ii} - t_{jj}|$ can be small (even if both $t_{ii}$ and $t_{jj}$ are far from zero). Algorithm 3 shows the details of this approach.

**Algorithm 3** The Parlett-Higham substitition algorithm for the matrix sign function.

```
1: Compute a (complex) Schur decomposition $A = QTQ^*$.
2: for $i = 1 : n,\ u_{ii} = \text{sign}(t_{ii})$
3: for $j = 2 : n$
4:    for $i = j - 1 : -1 : 1$
5:      if $u_{ii} + u_{jj} = 0$
6:        then $u_{ij} = t_{ij} \frac{u_{ii} - u_{jj}}{t_{ii} - t_{jj}} + \frac{\sum_{k=i+1}^{j-1} (u_{ik} t_{kj} - t_{ik} u_{kj})}{t_{ii} - t_{jj}}$
7:        else $u_{ij} = -\frac{\sum_{k=i+1}^{j-1} u_{ik} u_{kj}}{u_{ii} + u_{jj}}$
8:    end
9: end
```

## 2.1 | Applying the algorithms to the real Schur form

So far, we assumed that $T$ is upper triangular. If the underlying problem is to compute the sign of a general (i.e., non-triangular) real matrix $A$ with complex eigenvalues, it is more efficient to compute the so-called *real Schur form* than to work with the complex Schur form. The real Schur form is $A = Q\hat{T}Q^*$ with both $\tilde{T}$ and $Q$ being real, $Q$ is unitary, and $\hat{T}$ is *quasitriangular* with both 1-by-1 and 2-by-2 blocks on the diagonal. In the real Schur form, 2-by-2 diagonal blocks correspond to a conjugate pair of eigenvalues $z = x + iy$ and $\bar{z} = x - iy$ with the same sign, 1 if the eigenvalues are in the right half-plane and $-1$ if they are in the left half-plane.

Algorithm 2 can be applied directly to the real Schur form. Algorithm 3 can be adapted easily to the real Schur form: replace line 2 by a code that sets $u_{ii} = \text{sign}(t_{ii})$ if $u_{ii}$ is a 1-by-1 block, and set $u_{ii} = \text{sign}(z)$ if $u_{ii}$ is part of a 2-by-2 diagonal block with eigenvalues $z$ and $\bar{z}$. To prove that this modification is correct, note that the $FT = TF$ equation that implies that the 2-by-2 block in $U$ that corresponds to a 2-by-2 block in $T$ must be an identity or a minus identity (again relying on section 5.2 in the work of Higham et al.,[1] for example). This establishes the correctness of $u_{ii}$. It may seem that if $t_{ii}$ and $t_{i+1,i+1}$ are parts of a 2-by-2 block, we also need to set $u_{i,i+1} = 0$, but this will happen automatically, as in this case, $u_{ii} = u_{i+1,i+1}$ and the summation in line 7 of Algorithm 3 is empty. The correctness of the remainder of the algorithm follows from the same considerations as in the triangular real and complex cases.

## 2.2 | Arithmetic efficiency

Interestingly, the arithmetic efficiency of the two algorithms can vary considerably (and asymptotically). To design a high-performance algorithm, we need to choose the most efficient approach for a given matrix.

The arithmetic complexity of the Parlett–Higham recurrence varies between $n^3/3 + o(n^3)$ and $2n^3/3 + o(n^3)$ floating-point operations (flops). The actual number of operations depends on which $u_{ij}$s are computed from the equation $U^2 = I$ (bottom choice in Algorithm 3) and which are computed from $UT = TU$ (top choice), because in the first case, the algorithm computes one inner product on indexes ranging from $i+1$ to $j-1$, and in the second the algorithm, it computes two such inner products.

The arithmetic complexity of the Parlett–Sylvester algorithm for the sign function depends on how the eigenvalues of $T$ are initially ordered along its diagonal. The Schur reordering step (the Bai–Demmel algorithm or its partitioned variant by Kressner[11]) moves eigenvalues along the diagonal of a triangular matrix by swapping adjacent eigenvalues using Givens rotations. The number $k$ of swaps required to group together positive and negative eigenvalues varies between 0 and $\frac{n^2}{4}$.[12] The Schur reordering algorithm performs $12nk$ operations (ignoring low-order terms; see section 7.6.2 in the work of Golub et al.[13]), so the cost of this step varies between nothing (if the eigenvalues are already grouped by sign) and $3n^3$. The $12nk$ operations include those required to transform $Q$, the orthonormal matrix of Schur vectors.

Once this algorithm reorders the Schur form, it needs to solve a Sylvester equation for an $n_-$ by $n_+$ off-diagonal block, where $n_-$ and $n_+$ are the numbers of negative and positive eigenvalues, respectively. The number of arithmetic operations required to solve such a Sylvester equation is as follows:

$$n^2 - n \le n_- n_+ (n_- + n_+) \le n^3/4,$$

(it is easy to see that the extreme cases are $n_- = 1$ and $n_- = n/2$). We ignore in this analysis the trivial case where all the eigenvalues are positive or negative, in which the sign is $I$ or $-I$. As in the first step, the algorithm tends to get more expensive when the numbers of positive and negative eigenvalues are roughly balanced.

Finally, the algorithm needs to transform the sign of the reordered matrix to the sign of the input matrix. If this is done by applying the Givens rotations again, the cost depends on the number of swaps that were performed during the reordering step. In the best case, we need not transform at all, and in the worst case, the cost is cubic.

## 2.3 | Algorithm selection

The critical observation is that in easy cases that require few or noswaps to reorder the Schur form, the Parlett–Sylvester approach performs only a quadratic number of flops, whereas in the worst case, it performs more than $3n^3$ operations. This means that this approach can be much more efficient than the Parlett–Higham approach (if the former performs a quadratic number of operations and the latter a cubic number) or up to nine times less efficient.

The number of swaps in the Schur reordering algorithms is the number of swaps that bubble sort performs when it sorts the diagonal of the Schur form.[10] In our application, the Schur form only needs to order the diagonal of $T$ up to the sign of diagonal elements, not according to the actual numerical values; this can reduce the number of swaps dramatically. To determine the number of swaps required to reorder the Schur form, we run bubble sort twice on the vector of signs of eigenvalues, once in the natural order ($-1 < 1$) and once in the reverse order. The bubble sort is modified so that it counts the number of swaps that it performs; this number is the only output we need. We first limit our attention to the sorting order that yields the smallest number of swaps, denoted by $k$.

The overall algorithm uses the Parlett–Sylvester algorithm when $12nk < c \times 0.66n^3$, for some constant, $c$, that should be determined experimentally, ideally in an architecture similar to the one that will be used to run the algorithms. The $12nk$ and $0.66n^3$ are approximations to the number of arithmetic operations that the two algorithms will perform, and the constant $c$ should approximate the ratio between the operations-per-second rates of the two algorithms, which could be different. This approach of choosing between algorithms using a performance model that is partially theoretical and partially based on empirical estimation of parameters (here $c$) is common in other algorithmic problems as well.

Operation counts are not the only determinants of running time, so the actual performance differences may not be as dramatic, but operation counts do matter. We address another determinant of performancenext.

# 3 | COMMUNICATION LOWER BOUNDS

We next obtain a communication cost lower bound for Algorithm 3. The bound is an application of the work of Ballard et al.,[14] which extends a technique developed to bound communication in matrix multiplication[15] to many other computations in linear algebra. The technique embeds the iteration space of three-nested loop computations into a three-dimensional cube and utilizes the Loomis–Whitney[16] inequality to relate operation counts (the volume that the iterations fill in the cube) to communication requirements (the projections of the iterations on the input and output matrices).

The lower bound is derived from the computations performed in the inner loop, lines 5–7. It ignores the computations in line 2 (which can only increase the total communication cost). Note that either half or more of the executions of line 5 take the "then" branch (line 6) or half or more take the "else" branch on line 7.

We analyze first the second case, in which at least half the time, we have $u_{ii} + u_{jj} \neq 0$. We map the computation in line 7 to equation 2.1 in the work of Ballard et al.[14] In particular, we map $u_{ik}$ here to $a(i, k)$ there, $u_{kj}$ to $b(k, j)$, and $u_{ij}$ to $c(i, j)$. We map the scalar multiplication of $u_{ik}$ by $u_{kj}$ to the abstract function $g_{i,j,k}(\cdot, \cdot)$ in equation 2.1 in the work of Ballard et al.,[14] and the summation and scaling of the sum by $(u_{ii} + u_{jj})^{-1}$ to the abstract function $f_{i,j}$. We note that all computed $u_{ij}$ are part of the algorithm's output, so none of them is discarded; this implies, in the terminology of Ballard et al.,[14] that there are no $R2/D2$ intermediate results. By applying theorem 2.2 in the work of Ballard et al.,[14] we have the following:

**Corollary 1.** *Let $G_1$ be the number of arithmetic operations computed in line 7 of Algorithm 3. Let M be the cache size. Then, the communication cost (the number of words transferred between the cache and the main memory) in Algorithm 3 is at least $G_1/(8\sqrt{M}) - M$.*

We now analyze the communication required to perform the operations in line 6 of the algorithm, when $u_{ii} + u_{jj} = 0$. We again apply equation 2.1 and theorem 2.2 in the work of Ballard et al.[14] Let $a(i, k)$ there be our $u_{i,k}$, let $b(k,j)$ there be our $t_{k,j}$, and let $c(i,j)$ there be our $u_{i,j}$. Further, let $g_{i,j,k}(\cdot, \cdot)$ function be scalar multiplication $u_{ik} \cdot t_{k,j}$, and $f_{i,j}$ function be the computation of $u_{ij}$, which calls to $g_{ijk}$. Again, we note that all computed $u_{ij}$ are part of the algorithm's output, so none of them is discarded. We also note that we can impose writes on the the $n^2$ elements of $T$ (see section 3.4 in the work of Ballard et al.[14]), losing at most, $\Theta(n^2)$ of the lower bound. Thus, using the terminology of Ballard et al.,[14] there are no $R2/D2$ arguments.

By applying theorem 2.2 in the work of Ballard et al.,[14] we have the following:

**Corollary 2.** *Let $G_2$ be the number of arithmetic operations performed in line 6 of Algorithm 3. Let M be the cache size. Then, the communication cost of the algorithm is at least $\Omega(G_2/\sqrt{M} - M - \Theta(n^2))$.*

Let $G$ be the total number of arithmetic operations performed in the doubly nested loop of Algorithm 3. Recall that $\max\{G_1, G_2\} \geq G/2$. Combining Corollary 1 and Corollary 2, we conclude the following:

**Theorem 1.** *Let $G = \Theta(n^3)$ be the number of arithmetic operations computed in lines 5–7 of Algorithm 3, and let M be the size of the cache. The communication cost of Algorithm 3 is $\Omega(G/\sqrt{M} - M - \Theta(n^2))$. Assuming $M < n^2$, the cost is $\Omega(G/\sqrt{M}) = \Omega(n^3/\sqrt{M})$.*

The Parlett–Higham algorithm does not attain this lower bound. It is easy to show that for $n = o(\sqrt{M})$ (that is, the fast memory can hold less than a small constant fraction of the input matrix), the communication cost of this algorithm is $\Theta(n^3)$. The analysis is similar to the analysis of communication in matrix multiplication algorithms implemented using three nested loops. We omit the details. In Section 4.3, we present a communication-efficient version of the Parlett–Higham algorithm that does attain the lower bound.

## 4 | COMMUNICATION-EFFICIENT ALGORITHMS

We now propose communication-efficient variants of both algorithmic approaches. We begin with the Parlett–Sylvester approach, which is more straightforward.

### 4.1 | Communication-efficient Parlett–Sylvester solver

This approach calls two subroutines: a Schur reordering subroutine and a Sylverster equation solver. Fortunately, communication-efficient variants of both algorithms have been developed. Kressner[11] developed a communication-efficient variant of the Bai–Demmel reordering algorithm. Jonsson et al.[17] developed RECSY, a recursive communication-efficient Sylvester solver.

We have implemented this algorithmic approach in two ways. One calls xTRSEN, LAPACK's implementation of the Bai and Demmel algorithm that operates on rows and columns and ignores communication efficiency, and xTRSYL, LAPACK's Sylvester solver, which is similarly not communication efficient. The other calls communication-efficient codes by Kressner and by Jonsson et al. We use the first LAPACK-based implementation to evaluate the performance improvement achieved by the new communication-efficient approach.

Krassner's analysis of his communication-efficient Schur reordering algorithm does not include a formal communication upper bound. Therefore, we only analyze this algorithm experimentally.

## 4.2 | Communication-efficient Parlett–Higham solvers

The communication-efficient algorithm is a recursion that is based on a nested partitioning of the index set $\{1, 2, \dots, n\}$. The recursion is somewhat more complex than the recursion for simpler matrix algorithms (e.g., Cholesky). To present it and to prove its correctness, we introduce a notation for the nested partitioning and for sums over subsets of a partition.

**Definition 1.** A *nested partitioning* of $\{1, 2, \dots, n\}$ is a collection of index sets $p = \{P^{(0)}, P^{(1)}, \dots P^{(L)}\}$ such that $P^{(0)} = \{1\}$, and if $P^{(\ell)} = \{i_1, i_2 \dots, i_m\}$, then $1 \le i_1 < i_2 < \cdots < i_m \le n$ and $P^{(\ell-1)} = \{i_1, i_3, i_5, \dots i_m\}$ or $P^{(\ell-1)} = \{i_1, i_3, i_5, \dots i_{m-1}\}$.

Note that the definition implies that $i_1 = 1$. The indexes in a partition represent the beginnings of a block of row/column indexes. For example, $P^{(\ell)} = \{i_1, i_2 \dots, i_m\}$ represent the partitioning of the range $1 : n$ (in Matlab notation) into $i_1 : i_2 - 1 = 1 : i_2 - 1, i_2 : i_3 - 1$, etc.

For example, let $n = 1000$ and let

$$\begin{aligned} P^{(0)} &= \{1\} \\ P^{(1)} &= \{1, 500\} \\ P^{(2)} &= \{1, 250, 500, 750\} \\ P^{(3)} &= \{1, 125, 250, 375, 500, 625, 750, 875\} \ . \end{aligned}$$

We use nested partitions to denote blocks of vectors and matrices. Using the example above, we can denote blocks of a vector $v$ and a matrix $A$ by the following:

$$\begin{aligned} v_{250}^{(3)} &= v_{250:374} \\ v_{250}^{(2)} &= v_{250:499} \\ A_{250,625}^{(3)} &= A_{250:374,625:749} \end{aligned}$$

and so on. In this notation, a block of indexes at level $\ell$ must start at some $i_j \in P^{(\ell)}$, and it ends at $i_{j+1} - 1$. We now define a function that allows us to iterate over ranges in a given partition.

**Definition 2.** Let $P$ be a nested partitioning and let $P^{(\ell)} = \{i_1, i_2 \dots, i_m\}$. The function $\eta : P^{(\ell)} \to P^{(\ell)} \cup \{n+1\}$ returns the start index of the next range in a given partition as follows:

$$\eta^{(\ell)}(i_j) = i_{j+1} \left(\text{in } P^{(\ell)}\right) \ .$$

For completeness, we define the following:

$$\eta^{(\ell)}(i_m) = n + 1 \ ,$$

so that subtracting 1 from the next range always gives the last element in the current range. We also define the function $\pi$ that returns the *previous* range, as follows:

$$\pi^{(\ell)}(i_j) = i_{j-1}$$

and

$$\pi^{(\ell)}(n+1) = i_m \ .$$

We can now define how vectors and matrices are partitioned, as well as sum over ranges in a partition.

**Definition 3.** Let $P$ be a nested partition of $\{1, 2, \dots, n\}$, let $v$ be an $n$ vector, and let $A$ be an $n$-by-$n$ matrix. Let $i,j \in P^{(\ell)}$. We denote the following:

$$v_i^{(\ell)} = \begin{bmatrix} v_i \\ \vdots \\ v_{\eta^{(\ell)}(i)-1} \end{bmatrix}$$

and

$$A_{i,j}^{(\ell)} = \begin{bmatrix} A_{i,j} & \cdots & A_{i,\eta^{(\ell)}(j)-1} \\ \vdots & & \\ A_{\eta^{(\ell)}(i)-1,j} & \cdots & A_{\eta^{(\ell)}(i)-1,\eta^{(\ell)}(j)-1} \end{bmatrix}.$$

Clearly, $v_i^{(\ell)} = \left[ v_i^{(\ell+1)} \; v_{\eta^{(\ell+1)}(i)}^{(\ell+1)} \right]^T$ and similarly for matrices. We also need the reverse notation, which maps $v_i^{(\ell)} \to v_i^{(\ell+1)}$ (the first part of $v_i^{(\ell)}$) and $v_i^{(\ell)} \to v_{\eta^{(\ell+1)}(i)}^{(\ell+1)}$ (the second part of the vector in the next level of the nested partition). We denote these by the following:

$$\left(v_i^{(\ell)}\right)_i^{(\ell+1)} = v_i^{(\ell+1)}$$
$$\left(v_i^{(\ell)}\right)_{\eta^{(\ell+1)}(i)}^{(\ell+1)} = v_{\eta^{(\ell+1)}(i)}^{(\ell+1)}$$

and similarly for matrices.

**Definition 4.** Let $P$ be a nested partitioning and let $P^{(\ell)} = \{i_1, i_2 \; \dots \; , i_m\}$, let $s \in P^{(\ell)}$, and let $e \in P^{(\ell)}$ or $e = n+1$. We define the following:

$$\sum_{j=s}^{e-1} v_j^{(\ell)} = \begin{cases} 0 & s > e \\ \sum_{j=s}^{\eta^{(\ell)}(s)-1} v_j + \sum_{j=\eta^{(\ell)}(s)}^{e-1} v_j^{(\ell)} & \text{otherwise.} \end{cases}$$

The sum consists of all the elements of $v$ starting at the beginning of a range in $P^{(k)}$ and ending just before another range in $P^{(k)}$ starts. Note that the first sum on the right-hand side is a sum over scalars that iterates over consecutive integer indexes, whereas the second sum is defined (recursively) over sums of ranges. The superscript $(\ell)$ on the argument $v$ (or the lack of superscript) indicates the type of the sum.

The following lemma relates sums over ranges in adjacent partitions in a nest.

**Lemma 1.** *Let $P$ be a nested partitioning and let $P^{(\ell)} = \{i_1, i_2 \; \dots \; , i_m\}$, let $s \in P^{(\ell)}$, and let $e \in P^{(\ell)}$ or $e = n+1$. The following relation holds:*

$$\sum_{j^{(k)}=s}^{e-1} v_{j^{(k)}}^{(\ell)} = \begin{cases} \sum_{j=s}^{e-1} v_j^{(\ell-1)} & \textit{if s is odd and e is even} \\ v_s^{(\ell)} + \sum_{j=\eta^{(\ell)}(s)}^{e-1} v_j^{(\ell-1)} & \textit{if s is even and e is even} \\ \sum_{j=s}^{e-1} v_j^{(\ell-1)} + v_{\pi^{(\ell)}(e)}^{(\ell)} & \textit{if s is odd and e is odd} \\ v_{i_s}^{(\ell)} + \sum_{j=\eta^{(\ell)}(i_s)}^{\pi^{(k)}(i_e)-1} v_j^{(\ell-1)} + v_{\pi^{(\ell)}(i_e)}^{(\ell)} & \textit{if s is even and e is odd.} \end{cases}$$

The Higham–Parlett recurrence is based on the observation that the sign $U$ of $T$ satisfies both $TU = UT$ and $U^2 = I$. Neither of these equations alone define all the elements of $U$, but together, they do. We partition $U$ and $T$ into block matrices with square diagonal blocks using a nested partition $P$. The blocks also satisfy the equations, so for any $\ell$ in the nest,

$$(TU)_{ij}^{(\ell)} = (UT)_{ij}^{(\ell)}$$
$$(UU)_{ij}^{(\ell)} = I_{ij}^{(\ell)},$$

which expands into the following:

$$T_{ii}^{(\ell)} U_{ij}^{(\ell)} - U_{ij}^{(\ell)} T_{jj}^{(\ell)} = U_{ii}^{(\ell)} T_{ij}^{(\ell)} - T_{ij}^{(\ell)} U_{jj}^{(\ell)} + \sum_{k=\eta^{(\ell)}(i)}^{j-1} \left( U_{ik}^{(\ell)} T_{kj}^{(\ell)} - T_{ik}^{(\ell)} U_{kj}^{(\ell)} \right)$$
$$U_{ii}^{(\ell)} U_{ij}^{(\ell)} + U_{ij}^{(\ell)} U_{jj}^{(\ell)} = I_{ij}^{(\ell)} - \sum_{k=\eta^{(\ell)}(i)}^{j-1} U_{ik}^{(\ell)} U_{kj}^{(\ell)}.$$

We denote the sums on the right by the following:

$$X_{ij}^{(\ell)} = \sum_{k=\eta^{(\ell)}(i)}^{j-1} \left( U_{ik}^{(\ell)} T_{kj}^{(\ell)} - T_{ik}^{(\ell)} U_{kj}^{(\ell)} \right)$$

and

$$Y^{(\ell)}_{ij} = \sum_{k=\eta^{(\ell)}(i)}^{j-1} U^{(\ell)}_{ik} U^{(\ell)}_{kj} .$$

We now relate the blocks of *X* and *Y* at level $\ell$ to those at level $\ell + 1$. The easiest one is the $(2, 1)$ block, as follows:

$$\begin{aligned}
Y^{(\ell+1)}_{\eta^{(\ell+1)}(i),j} &= \sum_{k=\eta^{(\ell+1)}(\eta^{(\ell+1)}(i))}^{j-1} U^{(\ell+1)}_{ik} U^{(\ell+1)}_{kj} \\
&= \sum_{k=\eta^{(\ell)}(i)}^{j-1} U^{(\ell+1)}_{ik} U^{(\ell+1)}_{kj} \\
&= \left( \sum_{k=\eta^{(\ell)}(i)}^{j-1} U^{(\ell)}_{ik} U^{(\ell)}_{kj} \right)^{(\ell+1)}_{\eta^{(\ell+1)}(i),j} \\
&= \left( Y^{(\ell)}_{i,j} \right)^{(\ell+1)}_{\eta^{(\ell+1)}(i),j} .
\end{aligned}$$

In the $(2, 1)$ and $(2, 2)$ blocks, we need to add a contribution at the $\ell + 1$ level, as follows:

$$\begin{aligned}
Y^{(\ell+1)}_{i,j} &= \sum_{k=\eta^{(\ell+1)}(i)}^{j-1} U^{(\ell+1)}_{ik} U^{(\ell+1)}_{kj} \\
&= U^{(\ell+1)}_{i,\eta^{(\ell+1)}(i)} U^{(\ell+1)}_{\eta^{(\ell+1)}(i),j} + \sum_{k=\eta^{(\ell+1)}(\eta^{(\ell+1)}(i))}^{j-1} U^{(\ell+1)}_{ik} U^{(\ell+1)}_{kj} \\
&= U^{(\ell+1)}_{i,\eta^{(\ell+1)}(i)} U^{(\ell+1)}_{\eta^{(\ell+1)}(i),j} + \sum_{k=\eta^{(\ell)}(i)}^{j-1} U^{(\ell+1)}_{ik} U^{(\ell+1)}_{kj} \\
&= U^{(\ell+1)}_{i,\eta^{(\ell+1)}(i)} U^{(\ell+1)}_{\eta^{(\ell+1)}(i),j} + \left( \sum_{k=\eta^{(\ell)}(i)}^{j-1} U^{(\ell)}_{ik} U^{(\ell)}_{kj} \right)^{(\ell+1)}_{i,j} \\
&= U^{(\ell+1)}_{i,\eta^{(\ell+1)}(i)} U^{(\ell+1)}_{\eta^{(\ell+1)}(i),j} + \left( Y^{(\ell)}_{i,j} \right)^{(\ell+1)}_{i,j} ,
\end{aligned}$$

and

$$Y^{(\ell+1)}_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)} = \left( Y^{(\ell)}_{i,j} \right)^{(\ell+1)}_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)} + U^{(\ell+1)}_{\eta^{(\ell+1)}(i),j} U^{(\ell+1)}_{j,\eta^{(\ell+1)}(j)} .$$

The $(1, 2)$ block requires two contributions from level $\ell + 1$, as follows:

$$Y^{(\ell+1)}_{i,\eta^{(\ell+1)}(j)} = U^{(\ell+1)}_{i,\eta^{(\ell+1)}(i)} U^{(\ell+1)}_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)} + \left( Y^{(\ell)}_{i,j} \right)^{(\ell+1)}_{i,\eta^{(\ell+1)}(j)} + U^{(\ell+1)}_{i,j} U^{(\ell+1)}_{j,\eta^{(\ell+1)}(j)} .$$

The expressions for the blocks of *X* at level $\ell + 1$ are similar.

We can now present the algorithm, which we split into three procedures. The top-level procedure *sign* (Algorithm 4) allocates *U*, *X*, and *Y* and zeros *X* and *Y*. It calls a recursive procedure that computes a diagonal block of *U* at level $\ell = 0$ called *sign-diagonal* (Algorithm 5). Sign-diagonal calls itself recursively to compute the two diagonal blocks at level $\ell + 1$ and a third procedure, *sign-off-diagonal* (Algorithm 6), which computes an off-diagonal block of *U*. Sign-off-diagonal works by calling itself four times on the four subblocks at the next level and uses them to compute its output using matrix multiply–add routines.

**Algorithm 4** A procedure that allocates two auxiliary matrices, *X* and *Y*, and calls the recursive algorithm to compute the sign of a triangular matrix *T*.

```
function $U = \mathrm{sign}(T)$
allocate $n$-by-$n$ upper triangular matrices $U, X$, and $Y$
set $X = X^{(0)} = 0,\ Y = Y^{(0)} = 0$
sign-diagonal $(1, 0, T, U, X, Y)$
return $U$
```

**Algorithm 5** A recursive algorithm to compute a diagonal block $U_{ii}^{(\ell)}$ of the sign $U$ of $T$. We assume that the arguments are passed by reference and that the code modifies elements of arguments $U, X$, and $Y$.

```
function sign-diagonal $(i, \ell, T, U, X, Y)$
if $U_{ii}^{(\ell)}$ is 1-by-1 then $U_{ii}^{(\ell)} = u_{ii} = \mathrm{sign}(t_{ii})$.
otherwise,
**sign-diagonal** $(i, \ell+1, T, U, X, Y)$
**sign-diagonal** $(\eta^{(\ell+1)}(i), \ell+1, T, U, X, Y)$
**sign-offdiagonal** $(i, \eta^{(\ell+1)}(i), \ell+1, T, U, X, Y)$
return
```

**Algorithm 6** A recursive implementation of Parlett-Higham algorithm to compute an off-diagonal block $U_{ij}^{(\ell)}$. We again assume that the arguments are passed by reference. Elements of $U$ that have been computed in previous steps (calls to level $\ell+1$) are marked in red to emphasize dependencies.

```
function sign-offdiagonal $(i, j, \ell, T, U, X, Y)$
if $U_{ij}^{(\ell)}$ is 1-by-1 then $U_{ij}^{(\ell)} = u_{ij}$, which we compute as
    $u_{ij} = \begin{cases} \frac{-y_{ij}}{u_{ii}+u_{jj}}, & u_{ii}+u_{jj} \neq 0 \\ t_{ij}\frac{u_{ii}-u_{jj}}{t_{ii}-t_{jj}} + \frac{x_{ij}}{t_{ii}-t_{jj}}, & u_{ii}+u_{jj} = 0 \end{cases}$
and return. otherwise,
**sign-offdiagonal** $(\eta^{(\ell+1)}(i), j, \ell+1, T, U, X, Y)$
$X_{i,j}^{(\ell+1)} = \left(X_{i,j}^{(\ell)}\right)_{i,j}^{(\ell+1)} + \left(U_{i,\eta^{(\ell+1)}(i)}^{(\ell+1)} T_{\eta^{(\ell+1)}(i),j}^{(\ell+1)} - T_{i,\eta^{(\ell+1)}(i)}^{(\ell+1)} U_{\eta^{(\ell+1)}(i),j}^{(\ell+1)}\right)$
$Y_{i,j}^{(\ell+1)} = \left(Y_{i,j}^{(\ell)}\right)_{i,j}^{(\ell+1)} + U_{i,\eta^{(\ell+1)}(i)}^{(\ell+1)} U_{\eta^{(\ell+1)}(i),j}^{(\ell+1)}$
**sign-offdiagonal** $(i, j, \ell+1, T, U, X, Y)$
$X_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)}^{(\ell+1)} = X_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)}^{(\ell+1)} + \left(U_{\eta^{(\ell+1)}(i),j}^{(\ell+1)} T_{j,\eta^{(\ell+1)}(j)}^{(\ell+1)} - T_{\eta^{(\ell+1)}(i),j}^{(\ell+1)} U_{j,\eta^{(\ell+1)}(j)}^{(\ell+1)}\right)$
$Y_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)}^{(\ell+1)} = \left(Y_{i,j}^{(\ell)}\right)_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)}^{(\ell+1)} + U_{\eta^{(\ell+1)}(i),j}^{(\ell+1)} U_{j,\eta^{(\ell+1)}(j)}^{(\ell+1)}$
**sign-offdiagonal** $(\eta^{(\ell+1)}(i), \eta^{(\ell+1)}(j), \ell+1, T, U, X, Y)$
$X_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)}^{(\ell+1)} = X_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)}^{(\ell+1)} + \left(U_{i,\eta^{(\ell+1)}(i)}^{(\ell+1)} T_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)}^{(\ell+1)} - T_{i,\eta^{(\ell+1)}(i)}^{(\ell+1)} U_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)}^{(\ell+1)}\right) + \left(U_{i,j}^{(\ell+1)} T_{j,\eta^{(\ell+1)}(j)}^{(\ell+1)} - T_{i,j}^{(\ell+1)} U_{j,\eta^{(\ell+1)}(j)}^{(\ell+1)}\right)$
$Y_{i,\eta^{(\ell+1)}(j)}^{(\ell+1)} = \left(Y_{i,j}^{(\ell)}\right)_{i,\eta^{(\ell+1)}(j)}^{(\ell+1)} + U_{i,\eta^{(\ell+1)}(i)}^{(\ell+1)} U_{\eta^{(\ell+1)}(i),\eta^{(\ell+1)}(j)}^{(\ell+1)} + U_{i,j}^{(\ell+1)} U_{j,\eta^{(\ell+1)}(j)}^{(\ell+1)}$
**sign-offdiagonal** $(i, \eta^{(\ell+1)}(j), \ell+1, T, U, X, Y)$
return
```

We now show that Algorithm 4 is communication optimal.

**Theorem 2.** *Algorithm 4 performs $O(n^2+n^3/\sqrt{M})$ communication, attaining the asymptotic lower bound in Theorem 1.*

*Proof.* The top-level algorithm performs $\Theta(n^2)$ work on $\Theta(n^2)$ data: it allocates three $n$-by-$n$ upper triangular matrices and initializes two of them to zero. The sign-diagonal makes $\Theta(n)$ function calls and computes the diagonal of the output sign matrix from the diagonal of the input matrix. It performs $\Theta(n)$ work on $\Theta(n)$ data. Most of the work and most of the communication in the algorithm are performed by invocations of sign-off-diagonal. Let level $\ell_0$ be the first level, at the dimension $n_0$, satisfying $4n_0^2 \leq M$. That is, the inputs fit into fast memory. In the recursive function calls and the matrix multiply–add operations at levels 1 through $\ell_0 - 1$, computation-to-communication ratio is $\Theta(\sqrt{M})$, because all of it consists of calls to a matrix multiply–add routine on matrices with dimensions that are $\sqrt{M/4}$ or larger. The computation-to-communication ratio in levels $n_0$ and on is also $\Theta(\sqrt{M})$ because calls to sign-off-diagonal on inputs of dimension $n_0$ perform $\Theta(n_0^3)$ work and only $\Theta(n_0^2)$ communication (to bring the inputs to fast memory and to write back the outputs), with the ratio being $\Theta(n_0) = \Theta(\sqrt{M})$. □

## 4.3 | Improving the arithmetic complexity

Algorithm 4 performs $n^3$ arithmetic operations, more than the $n^3/3$ to $2n^3/3$ operations that the Parlett–Higham recurrence performs. This happens because extended-sign computes both the following:

$$\sum_{k=i+1}^{j-1} u_{ik}u_{kj} \quad \text{and} \quad \sum_{k=i+1}^{j-1} \left(u_{ik}t_{kj} - t_{ik}u_{kj}\right)$$

for every $i < j$, whereas Algorithm 3 only computes one of the two for a particular $i, j$. In other words, the algorithm computes all the entries of both $X$ and $Y$, but it does not actually use all of them later. For a given position $i, j$, only one of $x_{ij}$ and $y_{ij}$ is needed, the one that the sign-off-diagonal function needs. If $u_{ii} + u_{jj} = 0$, we need $x_{ij}$; otherwise, it is $y_{ij}$.

We can improve the arithmetic complexity of the algorithm by computing only one of $x_{ij}$ and $y_{ij}$. More specifically, when calculating the contributions to $X$ and $Y$ in between the recursive calls in sign-off-diagonal, we only compute elements of the $X$ argument that are actually needed and only elements of $Y$ that are actually needed. In practice, we can only keep one matrix $Z$ and decide on the method of calculating $z_{ij}$ based on the values $u_{ii}$ and $u_{jj}$.

This approach performs fewer arithmetic operations (by a factor of 2 to 3), but it prevents us from using existing matrix multiplication codes (e.g., xGEMM), so it is unlikely to be fast in practice. We have implemented this algorithm, but the experiments below demonstrate that it is indeed slow.

## 5 | EXPERIMENTAL RESULTS

We evaluated several different algorithms experimentally. We implemented the algorithms in C and called them from Matlab for testing, and we used the BLAS and LAPACK libraries that are bundled with Matlab. We used Matlab R2013A, which uses Intel's Math Kernel Library Version 10.3.11 for the BLAS and LAPACK and is based on LAPACK version 3.4.1.

We conducted the experiments on a quad-core desktop computer running Linux. The computer had 16 GB of RAM and an Intel i7-4770 CPU processor running at 3.40 GHz. Some of the experiments used only one core (using `maxNumCompThreads(1)` in Matlab), and some used all four (same function with argument 4) but only in BLAS routines. Runs that used four cores are labeled *MT* in the graphs below.

### 5.1 | Experiments on triangular matrices

We tested all the algorithms on random triangular matrices with a prescribed inertia. We generated the matrices by creating random real square matrices with elements that are distributed uniformly in $[-50, 50]$, computing their complex Schur form and taking the real part of the Schur form. This generates matrices with roughly balanced inertia. In the experiments reported below, the fraction of negative eigenvalues ranged from 48% to 54% on the smallest matrices (dimension 50), from 49% to 51% on the next smallest dimension (657), and even narrower on larger matrices. In some of the experiments, we forced the number of negative eigenvalues to a prescribed number $k$. We did this by keeping the absolute values of the diagonal elements of the random triangular matrix, but forcing their sign to positive in all but random $k$ positions.

We tested the following algorithms:

- The Parlett–Higham algorithm (Algorithm 3). We refer to this algorithm as *Higham* in the graphs below.
- Two implementations of the Parlett–Sylvester algorithm (specialized to the sign function). The first implementation uses LAPACK's built-in routines for reordering the Schur form and for solving the Sylvester equations. Neither routine is blocked in LAPACK 3.4.1. We refer to this implementation as *LAPACK Sylvester*.
- The second implementation of the Parlett–Sylvester algorithm used RECSY, a recursive Sylvester solver by Jonsson et al.,[17] as well as a blocked Schur reordering code by Kressner.[11]
- Our recursive implementation of the Parlett–Higham algorithm (Algorithms 4–6). This implementation calls the BLAS to multiply blocks. Recursion was used only on blocks with dimension larger than 16; smaller diagonal blocks were processed by our element-by-element Parlett–Higham implementation. We refer to this implementation as *Recursive Higham MM*.
- A recursive implementation of the arithmetic-efficient Parlett–Higham algorithm described in Section 4.3. This implementation does not use the BLAS (as its operations do not reduce to matrix multiplications). We refer to it as *Recursive Higham*.

The running times on matrices with roughly balanced inertia are shown in Figure 1. Our recursive algorithm is the fastest one, both with and without multithreaded BLAS. The next-best algorithm is the recursive Parlett–Sylvester algorithm. Like our recursive algorithm, it uses the BLAS extensively so it benefits from multithreading. Our recursive but arithmetic-efficient algorithm is fairly slow, because it does not use the BLAS. The slowest algorithms are the Parlett–Sylvester implementation that uses LAPACK for Schur reordering and for solving Sylvester equations and the element-by-element Parlett–Higham algorithm.

Figure 2 puts the same results in a somewhat more familiar quantitative context. By measuring performance in terms of normalized floating-point arithmetic rates, the performance of the algorithms can be directly compared with the

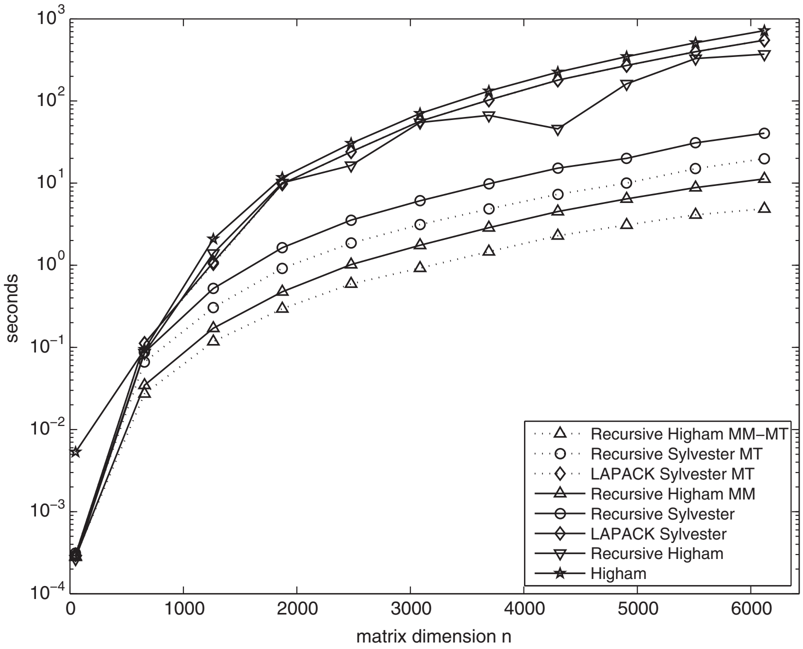

**FIGURE 1** Running times on matrices with roughly balanced inertia. The performance of the LAPACK Sylvester-based solver is essentially the same on one core and on four cores, because this code apparently uses no multithreaded BLAS routines that could have benefited from multiple cores. This is also the case in the next graphs

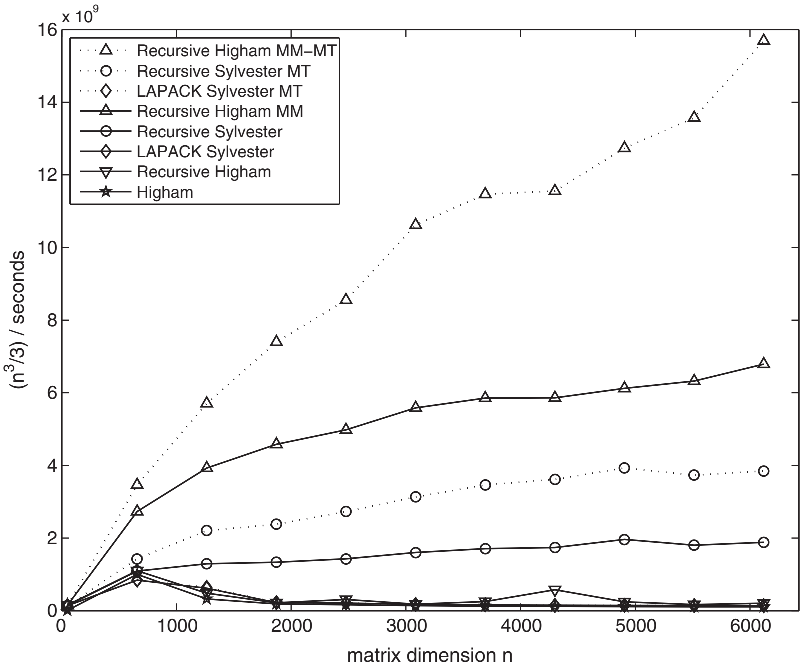

**FIGURE 2** Normalized computational rates on matrices with roughly balanced inertia. The number $n^3/3$ is used for normalization because the number of arithmetic operations in Higham's algorithm is between $n^3/3 + o(n^3)$ and $2n^3/3 + o(n^3)$; the number of operations in some of the other algorithms is different

performance of other algorithms (e.g., matrix multiplication) on the same computer. The rates are normalized relative to $n^3/3$ because the number of arithmetic operations in Higham's algorithm is between $n^3/3 + o(n^3)$ and $2n^3/3 + o(n^3)$; other algorithms may perform more or less arithmetic.

Our recursive algorithm always performs $2n^3/3 + o(n^3)$; on large matrices, it runs single threaded at a rate of about 12 Gflop/s (not normalized). Multithreading on the quad-core computer speeds up the algorithm by more than a factor of 2 on large matrices (the speedup is around 2 rather than 4 because only matrix multiplications exploit more than one core). The recursive Parlett–Sylvester is about three times slower. The performance of the nonrecursive algorithms (and of our recursive algorithm that does not use the BLAS) is quite dismal.

When inertia is highly imbalanced, the picture changes. Figure 3 shows that Parlett–Sylvester algorithms are the fastest on such matrices. This makes sense, as they only perform $\Theta(n^2)$ operations, not $\Theta(n^3)$ like all the other algorithms. The differences are quite dramatic. The best single-threaded Parlett–Sylvester algorithm (*Recursive Sylvester*) runs in 0.35 s on matrices of dimension 6120, whereas the fastest single-threaded recursive Higham algorithm takes 10.9 s (more than 30 times slower). Figure 4 shows the corresponding normalized rates for completeness, but they are less interesting because the normalization factor is way off for the Parlett–Sylvester algorithms.

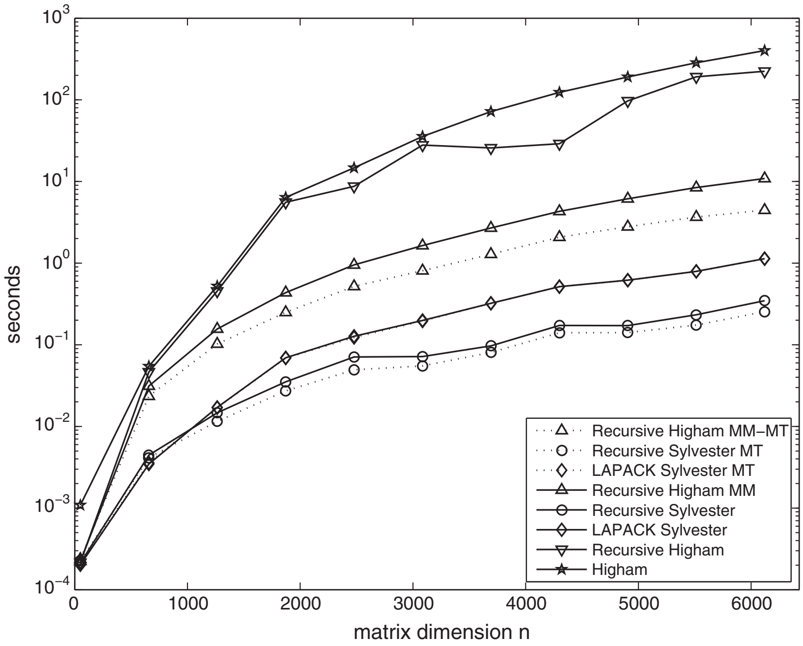

**FIGURE 3** Running times on matrices with exactly 3 negative eigenvalues and $n - 3$ positive eigenvalues

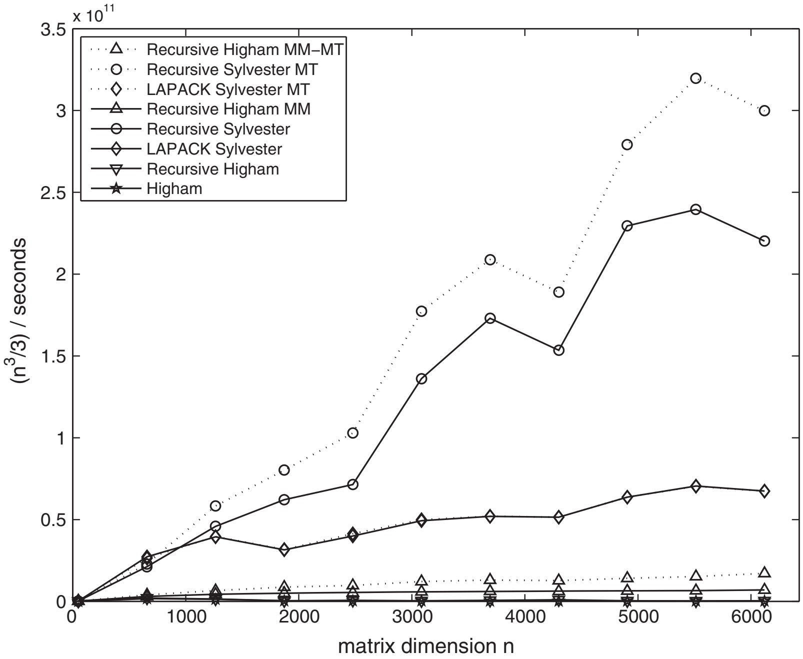

**FIGURE 4** Normalized computational rates on matrices with exactly 3 negative eigenvalues and $n - 3$ positive eigenvalues

## 5.2 | Further experiments for a wider context

We now discuss a second set of experiments that were designed and performed to put the algorithms that compute the sign of triangular matrices in a wider context. In particular, these experiments compare the performance of two families of algorithms that compute the sign of general (nontriangular) matrices. One family reduces the matrix to triangular form (Schur decomposition) and then applies one of the algorithms discussed in this study. Algorithms in the second family use a Newton iteration to compute the sign of the matrix directly. In general, Newton-type iterations repeatedly average the matrix and its inverse, sometimes scaling the matrix before averaging. Various Newton-type iterations differ mainly in the scaling that is applied and in the stopping criterion that is used.

The computationally expensive part of a Newton iteration is the inversion of the current iterate. The inversion is typically done using an LU decomposition and thus costs $\Theta(n^3)$ operations. The computation of the scaling factor and the evaluation of the stopping criterion are lower order terms ($\Theta(n)$ or $\Theta(n^2)$ and negligible on large matrices). Thus, in order to compare the expected performance of Newton-type algorithms with the performance of Schur-based algorithms, we need to compare the cost of inversions with the cost of the building blocks in Schur-based algorithms and also to assess the number of iterations that Newton-type algorithms perform. The two building blocks in Schur-based algorithms are the Schur decomposition itself and the computation of the sign of the triangular factor.

We begin by assessing of the number of iterations that Newton-type algorithms perform when computing the sign function in real-world applications. We used the CAREX benchmark collection by Benner et al.[18] to produce a set of continuous algebraic Ricatti equations. This class of matrix equations can be solved by computing the sign of the Hamiltonian matrix of the equations and then by solving a set of linear equations that are constructed from blocks of the sign matrix.[19] The benchmark collection consists of 20 such instances that arise in a range of applications; some of the instances are fixed and some parameterized; default values are provided for all parameters. We evaluated a Newton algorithm on all 20 problems with their default parameters and on larger instances of one of the parameterized problems, number 17.

We used a well-engineered state-of-the art Newton-type algorithm in the evaluation, which is algorithm 5.14 in the work of Higham et al.[1] We used determinantal scaling, using the expression in page 120 in the work of Higham et al.,[1]

**TABLE 1** The number of Newton iterations required to solve the continuous algebraic Ricatti equations in the CAREX collection[18] with their default parameters (table on the left) and with a parameter that changes the size of the Hamiltonian matrix

| Index | $n$ | Iteration count | Index | $n$ | Iteration count |
|---:|---:|---:|---:|---:|---:|
| 1 | 4 | 2 | 17 | 200 | 53 |
| 2 | 4 | 4 | 17 | 400 | 66 |
| 3 | 8 | 599 | 17 | 600 | 59 |
| 4 | 16 | 18 | 17 | 800 | 95 |
| 5 | 18 | 61 | 17 | 1,000 | 76 |
| 6 | 60 | 15 | 17 | 1,200 | 73 |
| 7 | 4 | 3 | 17 | 1,400 | 67 |
| 8 | 4 | 8 | | | |
| 9 | 4 | 8 | | | |
| 10 | 4 | 9 | | | |
| 11 | 4 | 782,644 | | | |
| 12 | 6 | 53 | | | |
| 13 | 8 | 7 | | | |
| 14 | 8 | 12 | | | |
| 15 | 78 | 9 | | | |
| 16 | 128 | 43 | | | |
| 17 | 42 | 23 | | | |
| 18 | 200 | 23 | | | |
| 19 | 120 | 15 | | | |
| 20 | 842 | 70 | | | |

*Note.* For each instance, the table shows the index of the instance from the work of Benner et al.,[18] the dimension of the Hamiltonian, and the number of iterations required for the algorithm to converge

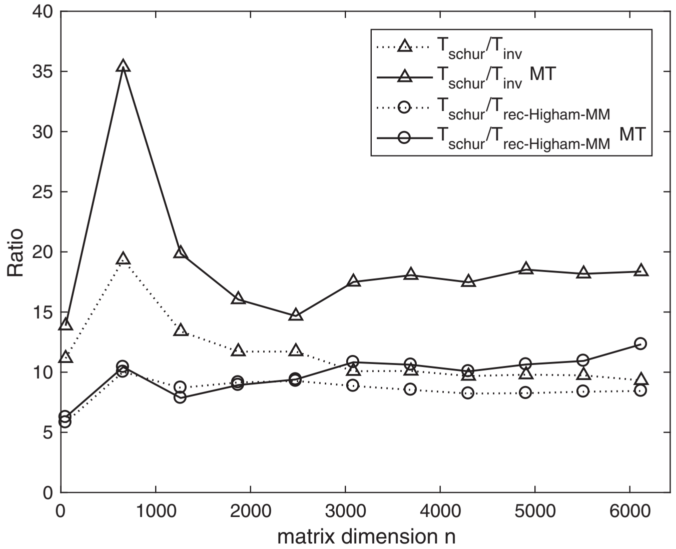

**FIGURE 5** The relative running times of Schur decomposition, matrix inversions, and computation of the sign of triangular matrices. The solid lines marked MT shows ratios of runs on all four cores of the computer, whereas dotted lines show ratios of runs on one core

to avoid overflows and underflows. The parameter tol_scale that tells the algorithm when to stop scaling was set to $10^{-2}$, and the parameter tol_cgce that controls termination was set to $5 \times 10^{-14}$; both are suggested in page 125 in the work of Higham et al.[1]

The results are shown in Table 1. First, we observe that it often takes a few dozen iterations for Newton methods to converge, even on small instances. In almost all cases, convergence occurred in less than 100 iterations, but sometimes not much less. Second, the experiment on instance 11 shows that Newton for the sign function methods sometimes fails to converge in a reasonable number of iterations; this is known and is not very surprising, considering the use of explicit inverses.

We next compare the cost of matrix inversions, the dominant building block in Newton methods, with the cost of the Schur decomposition and with the cost of the computation of the sign of the triangular factor. Figure 5 shows the results. The performance Schur decompositions and matrix inversions were estimated by applying the schur and inv functions in Matlab to five random real matrices of each dimension and by taking the average of the three fastest runs. The performance of the algorithm to compute the sign of the triangular factor is the data from Figure 1 (using the Recursive Higham method with calls to the BLAS, denoted *Recursive Higham MM* in Figures 1 and 5, which is the fastest on large matrices). The figure shows the running time ratios using both one core and all four cores. On one core, computing the Schur decomposition is about as expensive as 10 inversions and about as expensive as 10 computations of the sign of the triangular factor.

We conclude that the break-even point between Newton-type methods and Schur–Parlett methods is at the point where the Newton method performs about 11 iterations. For many of the CAREX problems, the Schur–Parlett approach is preferable. When using all 4 cores, the break-even point is around 20 Newton iterations, probably due to the not-so-effective parallelization of the Schur decomposition in Matlab.

## 6 | CONCLUSIONS

The reader may have been somewhat surprised by some aspects of this work. They also surprised us.

The first surprise is that arithmetic performance (the number of operations) can differ so dramatically between the Parlett–Higham recurrence and its block variant that we refer to as Parlett–Sylvester. The striking efficiency of the Parlett–Sylvester approach on matrices with highly imbalanced inertia is the result of three contributing factors: (a) The performance of the Schur reordering algorithm depends strongly on the number of eigenvalue swaps required to order the matrix. (b) Solving Sylvester equations on high-aspect ratio matrices is very inexpensive. (c) Computing the diagonal blocks in the Parlett–Sylvester algorithm *for the sign function* is trivial.

This finding implies that a production code for the sign function should choose between these two algorithms, ideally through an auto-tuning and/or performance-prediction framework, possibly based on inertia estimation.

Second was the difficulty of expressing clearly the recursive variant of the Higham–Parlett algorithm. We have tried a number of approaches based on conventional notational schemes and failed. We resorted to develop the somewhat complex notation that we present and use in Section 4; it may seem overly complex, but we found it impossible to present the algorithm without it.

The third (and relatively minor) surprise is the benefit of performing more arithmetic in order to use matrix multiplication. The arithmetic-efficient variant of the recursive Parlett–Higham algorithm (Section 4.3) is slower in practice, although it is cache efficient. Rather than using existing matrix multiplication routines (xGEMM), it uses a custom kernel with a condition in the next-to-inner loop. This demonstrated the performance penalty for trying to do less arithmetic in an algorithm using a conditional, thus making performance optimization difficult.

We have also carried out a set of experiments to provide a wider perspective on the performance of algorithms to compute the sign function. These experiments show that Schur–Parlett methods are expected to be faster than Newton-type methods on many problems.

## ACKNOWLEDGEMENTS

This research was supported in part by grants 863/15, 1878/14, and 1901/14 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities) and by grant 3-10891 from the Ministry of Science and Technology, Israel. Research is also supported by the Einstein Foundation and the Minerva Foundation. This paper was supported by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI) and by a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel. This work was also supported by the HUJI Cyber Security Research Center in conjunction with the Israel National Cyber Bureau in the prime minister's office.

## ORCID

*Sivan Toledo* http://orcid.org/0000-0002-9524-7115

## REFERENCES

1. Higham NJ. Functions of matrices: Theory and algorithm. Philadelphia, USA: SIAM; 2008.
2. Roberts JD. Linear model reduction and solution of the algebraic Riccati equation by use of the sign function. Internat J Control. 1980;32(4):677–687. First issued as report CUED/B-Control/TR13, Department of Engineering, University of Cambridge, 1971.
3. Benner P, Quintana-Ortí ES. Solving stable generalized Lyapunov equations with the matrix sign function. Numer Algorithms. 1999;20(1):75–100.
4. Benner P, Quintana-Ortí ES, Quintana-Ortí G. Solving stable Sylvester equations via rational iterative schemes. J Sci Comput. 2006;28:51–83.
5. Kenney CS, Laub AJ. The matrix sign function. IEEE Trans Automat Control. 1995;40(8):1330–1348.
6. Parlett BN. A recurrence among the elements of functions of triangular matrices. Linear Algebra Appl. 1976;14:117–121.
7. Davies PI, Higham NJ. A Schur–Parlett algorithm for computing matrix functions. SIAM J Matrix Anal Appl. 2003;25(2):464–485.
8. Deadman E, Higham NJ, Ralha R. Blocked Schur algorithms for computing the matrix square root. In: Manninen P, Öster P, editors. Revised papers from the 11th International Conference on Applied Parallel and Scientific Computing (PARA 2012). Berlin Heidelberg: Springer; 2013.
9. Parlett BN. Computation of functions of triangular matrices. Memorandum ERL-M481. Berkeley: Electronics Research Laboratory, College of Engineering, University of California; 1974.
10. Bai Z, Demmel JW. On swapping diagonal blocks in real Schur form. Linear Algebra Appl. 1993;186:73–95.
11. Kressner D. Block algorithms for reordering standard and generalized Schur forms. ACM Trans Math Softw. 2006;32:521–532.
12. Choi Ng K. Contributions to the computation of the matrix exponential. Berkeley: University of California; 1984. Center for Pure and Applied Mathematics. PhD thesis. Technical Report PAM-212.
13. Golub G, Van Loan C. Matrix computations. 4th ed. Baltimore, MD: The Johns Hopkins University Press; 2013.
14. Ballard G, Demmel J, Holtz O, Schwartz O. Minimizing communication in linear algebra. SIAM J Matrix Anal Appl. 2011;32:866–901.
15. Irony D, Toledo S, Tiskin A. Communication lower bounds for distributed-memory matrix multiplication. J Parallel Distrib Comput. 2004;64:1017–1026.
16. Loomis LH, Whitney H. An inequality related to the isoperimetric inequality. Bull Am Math Soc. 1949;55:961–962.

17. Jonsson I, Kågström B. Recursive blocked algorithms for solving triangular systems: Part II: Two-sided and generalized Sylvester and Lyapunov matrix equations. ACM Trans Math Softw. 2002;28(4):416–435. Available from: http://www8.cs.umu.se/~isak/recsy
18. Benner P, Laub AJ, Mehrmann V. Benchmarks for the numerical solution of algebraic Ricatti equations. IEEE Control Syst. 1997;17:18–28.
19. Kenney CS, Laub AJ, Papadopoulos PM. Matrix-sign algorithms for Riccati equations. IMA J Math Control Inf. 1992;9:331–344.

**How to cite this article:** Stotland V, Schwartz O, Toledo S. High-performance direct algorithms for computing the sign function of triangular matrices. *Numer Linear Algebra Appl*. 2017;e2139. https://doi.org/10.1002/nla.2139