Algorithms for Hessenberg-Triangular Reduction of Fiedler Linearization of Matrix Polynomials

Karlsson, Lars and Tisseur, Francoise

2014

MIMS EPrint: **2014.25**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

ALGORITHMS FOR HESSENBERG–TRIANGULAR REDUCTION OF FIEDLER LINEARIZATION OF MATRIX POLYNOMIALS *

LARS KARLSSON† AND FRANÇOISE TISSEUR‡

Abstract. Small- to medium-sized polynomial eigenvalue problems can be solved by linearizing the matrix polynomial and solving the resulting generalized eigenvalue problem using the QZ algorithm. The QZ algorithm, in turn, requires an initial reduction of a matrix pair to Hessenberg–triangular form. In this paper, we discuss the design and evaluation of high-performance parallel algorithms and software for Hessenberg–triangular reduction of a specific linearization of matrix polynomials of arbitrary degree. The proposed algorithm exploits the sparsity structure of the linearization to reduce the number of operations and improve the cache reuse compared to existing algorithms for unstructured inputs. Experiments on both a workstation and an HPC system demonstrate that our structure-exploiting parallel implementation can outperform both the general LAPACK routine `DGGHRD` and the prototype implementation `DGGHR3` of a general blocked algorithm.

Key words. Hessenberg-triangular reduction, polynomial eigenvalue problem, linearization, blocked algorithm, parallelization

1. Introduction.

A complex *matrix polynomial* of degree d takes the form

$$P(\lambda) = \lambda^d P_d + \lambda^{d-1} P_{d-1} + \cdots + \lambda P_1 + P_0,$$

where $P_k \in \mathbb{C}^{n\times n}$ and $P_d \neq 0$. Throughout this paper we assume that $P(\lambda)$ is regular, that is, that $\det P(\lambda)$ is not identically zero. The *polynomial eigenvalue problem* consists of finding scalars $\lambda \in \mathbb{C}$ and nonzero vectors $x, y \in \mathbb{C}^n$ satisfying the equations $P(\lambda)x = 0$ and $y^*P(\lambda) = 0$. The vectors x and y are *right and left eigenvectors*, respectively, associated with the *eigenvalue* λ. The finite eigenvalues of P are roots of the scalar equation $\det P(\lambda) = 0$. When $\det P(\lambda) = 0$ has degree $r < dn$ then we add $dn - r$ eigenvalues at infinity, which correspond to the zero eigenvalues of the reversal polynomial $\lambda^d P(1/\lambda)$.

A standard approach to solve a polynomial eigenvalue problem is to construct from the polynomial P a *linear* polynomial

$$L(\lambda) = A + \lambda B, \quad A, B \in \mathbb{C}^{dn\times dn}$$

with the same spectrum as P. The associated *generalized eigenvalue problem*

$$L(\lambda)z = 0, \quad w^*L(\lambda) = 0 \tag{1.1}$$

can then be solved using standard techniques and readily available software. Unless the dimension dn of L is very large, dense transformation-based methods can be used to solve (1.1). This amounts to reducing the matrix pair (A, B) to *Hessenberg–triangular* (HT) form by a unitary equivalence transformation $(A, B) \mapsto (Q^*AZ, Q^*BZ)$, where $Q, Z \in \mathbb{C}^{dn\times dn}$ are unitary matrices, followed by the QZ algorithm to obtain a generalized Schur form, again with a unitary equivalence transformation.

---

*Version of May 21, 2014

†Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden (larsk@cs.umu.se). This author was supported by UMIT Research Lab via Balticgruppen, and eSSENCE: a strategic collaborative eScience programme funded by the Swedish Research Council.

‡School of Mathematics, The University of Manchester, Manchester, M13 9PL, UK (francoise.tisseur@manchester.ac.uk). This author was supported by Engineering and Physical Sciences Research Council grant EP/I005293.

There are infinitely many linearizations to choose from and they tend to be structured and relatively sparse [1, 4, 14]. In this paper, we exploit the structure of a particular Fiedler linearization to devise an efficient algorithm for HT reduction that has lower arithmetic and communication costs than general HT reduction algorithms. The Fiedler linearization of interest is illustrated below for the case $d = 5$,

$$L(\lambda) = A + \lambda B = \begin{bmatrix} P_4 & P_3 & P_2 & P_1 & -I \\ -I & 0 & 0 & 0 & 0 \\ 0 & -I & 0 & 0 & 0 \\ 0 & 0 & -I & 0 & 0 \\ 0 & 0 & 0 & P_0 & 0 \end{bmatrix} + \lambda \begin{bmatrix} P_5 & 0 & 0 & 0 & 0 \\ 0 & I & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix}, \tag{1.2}$$

with a straightforward generalization to arbitrary degrees (see also Appendix A). The sparsity structure is illustrated in pictorial form in Figure 1.1. This particular linearization is used by the eigensolver `quadeig` for dense quadratic ($d = 2$) eigenvalue problems [5]. It has several desirable properties. First of all, $L(\lambda)$ is always a lin-

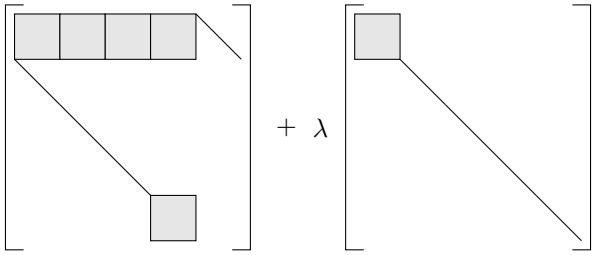

FIG. 1.1. *Illustration of the sparsity structure of the linearization* (1.2) *for the case* $d = 5$.

earization of $P(\lambda)$ in the sense that $E(\lambda)L(\lambda)F(\lambda) = \operatorname{diag}\big(P(\lambda), I_{(d-1)n}\big)$ for some matrix polynomials $E(\lambda)$ and $F(\lambda)$ with constant nonzero determinant. As a result, $\det P(z) = \gamma \cdot \det L(\lambda)$ for some nonzero constant γ, so that P and L have the same finite eigenvalues. In fact $L(\lambda)$ is a strong linearization: it has the same finite and infinite elementary divisors as P [13, Thm. 3]. The pencil $L(\lambda)$ in (1.2) is easy to construct from the coefficients P_i of $P(\lambda)$, without performing any arithmetic operation. The right and left eigenvectors x and y of $P(\lambda)$ are easily recovered from those of $L(\lambda)$. Indeed, if z and w are right and left eigenvectors of $L(\lambda)$ then

$$x = \begin{cases} (e_1^T \otimes I)z & \text{if } \lambda = \infty, \\ (e_j^T \otimes I)z, & 1 \le j \le d-1 \text{ if } \lambda \in \mathbb{C}, \end{cases} \qquad y = \begin{cases} (e_1^T \otimes I)w & \text{if } \lambda = \infty, \\ (e_j^T \otimes I)w & j = 1, d \text{ if } \lambda \in \mathbb{C}, \end{cases}$$

where e_j denotes the jth column of the $n \times n$ identity matrix and $\otimes$ denotes the Kronecker product. Also, when $\|P_i\|_2 \approx 1$ for $i = 0\colon d$ then $L(\lambda)$ has good conditioning and backward stability properties [3, 6, 7]. Moreover the pair (A, B) is almost already in HT form since A is in upper block Hessenberg form with $2n - 1$ subdiagonals and B is in block upper triangular form. It is straightforward to transform (A, B) using unitary equivalence such that A becomes upper block Hessenberg with only n subdiagonals and B becomes upper triangular. Furthermore we show in Appendix A that prior to the HT reduction, deflation of zero and infinite eigenvalues contributed by singular trailing and leading coefficient matrices can be performed without altering the block structure of (A, B) in (1.2). This is in part due to the position of P_0 and P_d in A and B.

The rest of the paper is organized as follows. The preprocessing step is described in Section 2. A known algorithm for reduction from block HT form to proper HT form is recalled in Section 3 and its effect on the sparsity structure when applied to inputs of the form (1.2) is analyzed. This is combined with recent work on cache-efficient algorithms for the unstructured case in Section 4, and the construction of cache-efficient algorithms tailored for the structured case is described thereafter. In Section 5 we discuss a parallelization scheme suitable for multicore-based systems. Numerical results are presented in Section 6 and Section 7 concludes.

2. Preprocessing. If P_0 is singular then $P(\lambda)$ has zero eigenvalues since $P(0)x = P_0x = 0$ for $x \in \text{null}(P_0)$. Similarly, if P_d is singular then $P(\lambda)$ has eigenvalues at infinity: these are the zero eigenvalues of the reversal polynomial $\lambda^d P(1/\lambda)$. Although the QZ algorithm handles eigenvalues at infinity well [15], numerical experiments by Kågström and Kressner [8] show that if infinite eigenvalues are not extracted before starting the QZ steps then they may never be detected due to the effect of rounding errors in floating point arithmetic. It is then desirable to deflate these eigenvalues before the HT reduction step. Starting from the linearization (1.2) we show in Appendix A how to deflate zero and infinite eigenvalues of $P(\lambda)$ contributed by singular leading and trailing coefficients, resulting in a pencil $A + \lambda B$ of dimension $(d-2)n + r_0 + r_d$ with the same block structure as in (1.2) but with blocks of different dimensions. Here $r_i = \text{rank}(P_i)$ for $i = 0, d$.

To avoid clutter in the notation and for simplicity we assume in what follows that no deflation occurred so that all the blocks in (1.2) are $n \times n$. The block upper triangular matrix B can immediately be reduced to upper triangular form by a QR factorization of its top left block (i.e., P_d). The lower bandwidth of the block upper Hessenberg matrix A can be reduced from $2n-1$ to n by a QR factorization of its bottom right subdiagonal block (i.e., P_0). More specifically, let $P_d = Q_d R_d$ and $P_0 = Q_0 R_0$ be QR factorizations of P_d and P_0, respectively. By defining the transformation matrices

$$Q = \begin{bmatrix} Q_d & 0 & 0 \\ 0 & I_{(d-2)n} & 0 \\ 0 & 0 & Q_0 \end{bmatrix}, \quad Z = \begin{bmatrix} I_n & 0 & 0 \\ 0 & I_{(d-2)n} & 0 \\ 0 & 0 & Q_0 \end{bmatrix} \tag{2.1}$$

(see Figure 2.1 for their pictorial form), the unitary equivalence transformation $(A, B) \leftarrow (Q^*AZ, Q^*BZ)$ accomplishes the desired preprocessing step. Following this transformation, the matrix pair (A, B) is in block HT form with n subdiagonals in A. An example for the case $d = 5$ takes the form

$$A + \lambda B = \begin{bmatrix} Q_5^*P_4 & Q_5^*P_3 & Q_5^*P_2 & Q_5^*P_1 & -Q_5^*Q_0 \\ -I & 0 & 0 & 0 & 0 \\ 0 & -I & 0 & 0 & 0 \\ 0 & 0 & -I & 0 & 0 \\ 0 & 0 & 0 & R_0 & 0 \end{bmatrix} + \lambda \begin{bmatrix} R_5 & 0 & 0 & 0 & 0 \\ 0 & I & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix}. \tag{2.2}$$

The pictorial form of (2.2) is illustrated in Figure 2.2.

3. Hessenberg–Triangular Reduction via Bulge-Chasing. Assume that the pair (A, B) has already been preprocessed as explained in Section 2 and that Q and Z are the resulting transformation matrices. The aim of this section is to recall a known algorithm that takes a matrix pair in block HT form and reduces it to proper HT form via a bulge-chasing procedure. This algorithm appears as the second stage of a two-stage HT reduction algorithm for general dense matrix pairs [2, 9].

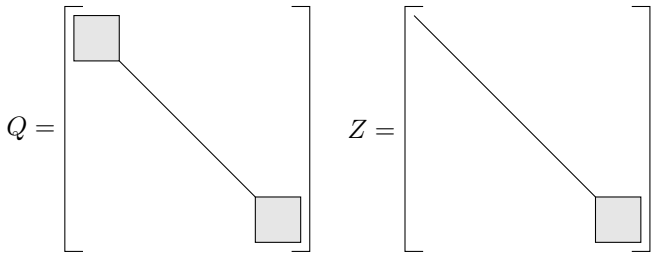

FIG. 2.1. *Illustration of the sparsity structure of the Q and Z resulting from the preprocessing in the case $d = 5$.*

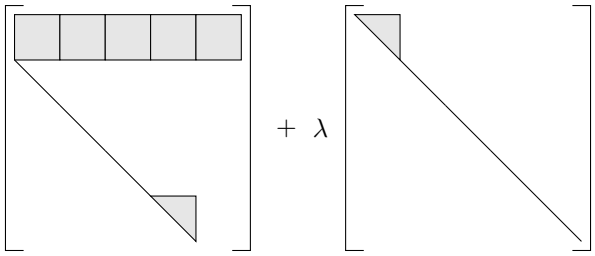

FIG. 2.2. *Illustration of the sparsity structure of the preprocessed linearization* (2.2) *for the case $d = 5$.*

Algorithm 1 is a reformulation of the bulge-chasing algorithm using our notation. The inputs to Algorithm 1 include the degree, d, and size, n, of the polynomial matrix as well as the preprocessed pair (A, B) and the resulting transformation matrices Q and Z. The algorithm reduces (A, B) from block HT form with n nonzero subdiagonals to proper HT form by systematically applying a carefully chosen set of plane rotations. The product of the rotations correspond to a unitary equivalence transformation of (A, B). The algorithm returns the reduced matrix pair (A, B) in proper HT form along with the updated transformation matrices Q and Z.

In Section 3.1 we define the computational kernels that appear in Algorithm 1. The algorithm itself and its effects on unstructured inputs are described in Section 3.2. In Section 3.3 we present a block partitioning of A and B that comes naturally from the algorithm. The definition of the block partitioning depends on the iteration index c in the sense that the block boundaries shift one step south-east from one iteration to the next. To avoid ambiguities we introduce notation for referring to a specific block in a specific iteration. Finally, the effect of the algorithm on the sparsity structure of a preprocessed pair (A, B) from Section 2 is analyzed in Section 3.4. The result of this analysis forms the basis for our further developments.

3.1. Computational kernels introduced in Algorithm 1. The four computational kernels that appear in Algorithm 1 are described below.

- $G \leftarrow \texttt{Zero}(A, r, c)$
  Constructs a row rotation G that zeros out $A(r, c)$ using $A(r-1, c)$ as a pivot.
- $G \leftarrow \texttt{RightZero}(A, r, c)$
  Constructs a column rotation G that zeros out $A(r, c)$ using $A(r, c+1)$ as a pivot.
- $\texttt{LeftRot}(G, A, r_1, r_2, c_1, c_2)$
  Applies a row rotation G to rows r_1 and r_2 of A. The effect is limited to the

Algorithm 1: $A, B, Q, Z \leftarrow$ `HTBasic`(n, d, A, B, Q, Z): Given a matrix pair (A, B) with matrices of size $dn \times dn$ in upper block HT form with n nonzero subdiagonals in A, this algorithm reduces (A, B) to proper HT form and accumulates the associated transformations into the given transformation matrices Q and Z.

```
1  for c ← 1 to dn − 2 do
2      for r ← min{dn, c + n} down to c + 2 do
3          c̃ ← c;
4          r̃ ← r;
5          while r̃ ≤ dn do
               /* Zero out A(r̃, c̃) with a row rotation.                    */
6              G ← Zero(A, r̃, c̃);
7              LeftRot(G, A, r̃ − 1, r̃, c̃, dn);
8              LeftRot(G, B, r̃ − 1, r̃, r̃ − 1, dn);
9              RightRot(G, Q, 1, dn, r̃ − 1, r̃);
               /* Move from A to B.                                         */
10             c̃ ← r̃ − 1;
               /* Zero out B(r̃, c̃) using a column rotation.                */
11             G ← RightZero(B, r̃, c̃);
12             RightRot(G, A, 1, min{dn, c̃ + n + 1}, c̃, c̃ + 1);
13             RightRot(G, B, 1, r̃, c̃, c̃ + 1);
14             RightRot(G, Z, 1, dn, c̃, c̃ + 1);
               /* Move from B to A.                                         */
15             r̃ ← c̃ + n + 1;
```

column range $c_1 : c_2$.

- `RightRot`$(G, A, r_1, r_2, c_1, c_2)$
  Applies a column rotation G to columns c_1 and c_2 of A. The effect is limited to the row range $r_1 : r_2$.

3.2. The bulge-chasing algorithm and its effect on unstructured inputs. Algorithm 1 systematically zeros out the undesired nonzero entries below the first subdiagonal of A. The entries are zeroed one column at a time from left to right (loop over c on line 1) and from the bottom up within each column (loop over r on line 2). The entry $A(r, c)$ is zeroed out by a row rotation involving rows $r - 1$ and r constructed on line 6.

A row rotation on rows $r - 1$ and r introduces an undesired nonzero entry (a *fill* or *bulge*) in the subdiagonal entry $B(r, r - 1)$. Unless the bulge is removed, the lower triangular part of B would rapidly fill in and its sparsity structure would be destroyed. Therefore, the bulge is removed using a column rotation that acts on columns $r-1$ and r. The rotation is constructed on line 11 and applied to A, B, and Z on the following lines. This creates, however, a second bulge at $A(r + n, r - 1)$ unless $r + n > dn$ in which case no new bulge appears. At most one bulge is present at any given time in either A or in B. It helps to interpret the algorithm as if it moves a single bulge back and forth between A and B until the bulge finally disappears near the bottom right corner. The precise movement of the bulge is illustrated in Figure 3.1.

Algorithm 1 uses the two variables $\tilde{r}$ and $\tilde{c}$ to keep track of the current position

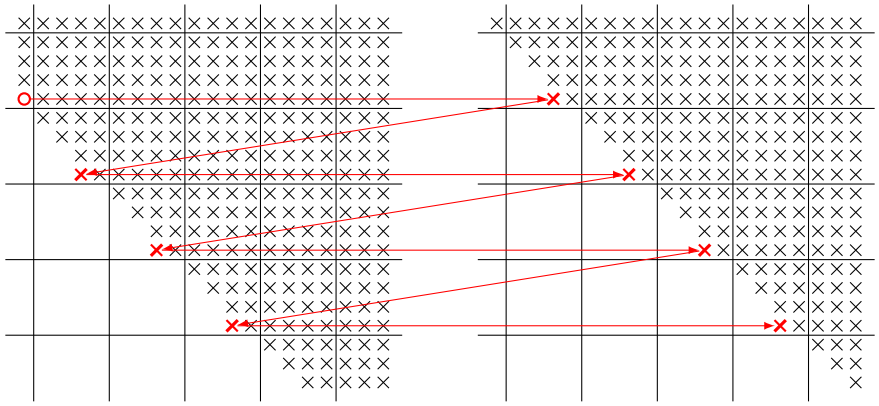

FIG. 3.1. *Illustration of the bulge movement in Algorithm 1 for the case $d = 5$, $n = 4$, $c = 1$, and $r = 4$ when the algorithm is applied to unstructured inputs. The matrix A is displayed on the left and B on the right.*

of the bulge. In each iteration of the while-loop on line 5, an entry is zeroed out in A, then a bulge is generated and zeroed out in B, and finally the bulge reappears in A. The loop continues to chase the bulge until it disappears.

3.3. Block partitioning induced by Algorithm 1. The computations associated with iteration c of the outer loop in Algorithm 1 induce a natural block partitioning of A and B, as illustrated by the solid lines in Figures 3.1 and 3.2 for the first iteration ($c = 1$). The block partitioning depends on the loop index c and for iteration $c \in \{1, 2, \ldots, dn-2\}$, the $(1,1)$ block has size $c \times c$. The other blocks on the diagonal all have size $n \times n$, except possibly the last block which may be smaller. It follows that the blocks on the subdiagonal generally have size $n \times n$, except possibly the first and the last blocks, which may be rectangular. As the partitioning is symmetric, the number of blocks in either dimension is

$$N_c = \left\lceil \frac{dn - c}{n} \right\rceil + 1. \tag{3.1}$$

The dependence on c manifests itself as a shift by one down the diagonal from one iteration to the next, as illustrated by the red dashed lines in Figure 3.2 for the second iteration ($c = 2$).

To unambiguously refer to a specific block relative to a given partitioning/iteration, we introduce the following notation. The block $(i,j)_c^X$ refers to block (i,j) of matrix X (either $X = A$ or $X = B$) with respect to the block partitioning in iteration c.

3.4. The effect of Algorithm 1 on structured inputs. We now study the effect of Algorithm 1 on the sparsity structure when the input is a preprocessed matrix pair as defined in Section 2. Figure 3.3 illustrates the generic sparsity structure of A and B for the case $d = 5$ and $n = 4$. Even though Algorithm 1 eventually destroys the structure of all four of the involved matrices (A, B, Q, and Z), the process turns out to be gradual and the evolving sparsity structure can be exploited, as we will soon explain, to reduce both the arithmetic and communication costs.

Informally, the initial sparsity structure shown in Figure 3.3 evolves during the execution of Algorithm 1 as follows. The dense upper triangular block of B grows slowly by one column to the right for each iteration of the outer for-loop. The dense

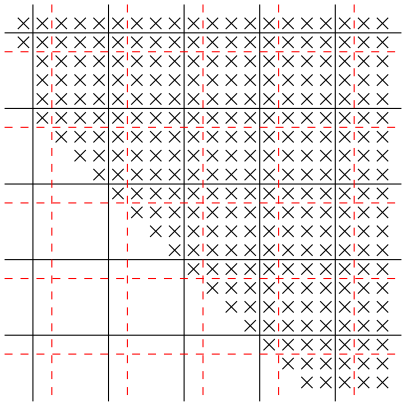

FIG. 3.2. *Illustration of the block partitioning of A for $c = 1$ (solid black) and $c = 2$ (dashed red).*

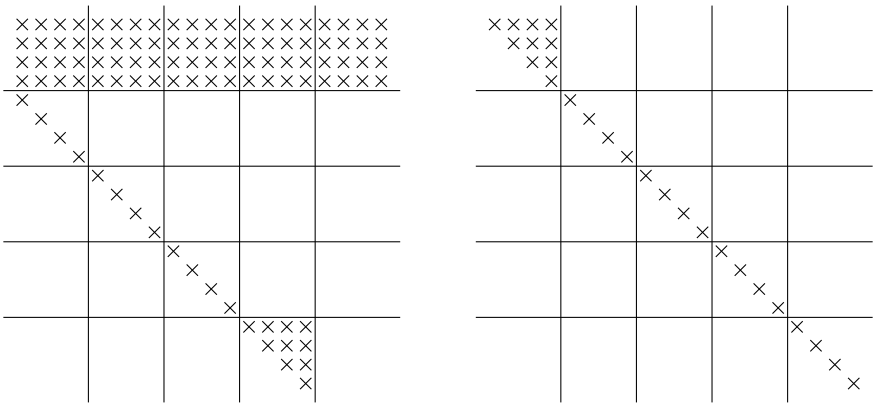

FIG. 3.3. *Generic sparsity structure of a preprocessed matrix pair (A, B) for the case $d = 5$ and $n = 4$.*

first block row of A also grows slowly by one row for each iteration of the outer for-loop. The upper triangular block of A, however, grows rapidly by one row for each iteration of the *inner* for-loop. Moreover, this block also grows slowly by one column for each iteration of the *outer* for-loop.

We refer to one iteration of the inner for-loop on line 2 of Algorithm 1 as a *sweep*, and Figure 3.4 illustrates the sparsity structure obtained after the first sweep. Note the appearance of a new column on the triangular block of B, a new row on the dense first block row of A, and a new row and column on the upper triangular block of A. The two remaining sweeps in the current iteration of the outer for-loop each add one row to the triangular block of A, but they have no other effect on the sparsity structures of either A or B. After completing the first iteration of the outer for-loop, the sparsity structure takes on the form illustrated in Figure 3.5. The effect on the sparsity structure is similar for all iterations. It follows that after $d - 2$ iterations the two dense blocks of A will fuse together, and after n iterations the triangular block of A will reach the east boundary of the matrix.

These observations suggest that some of the arithmetic operations required to update the matrix B can be eliminated since they operate on zeros and thus have no effect. A fraction of the arithmetic operations required to update the matrix A

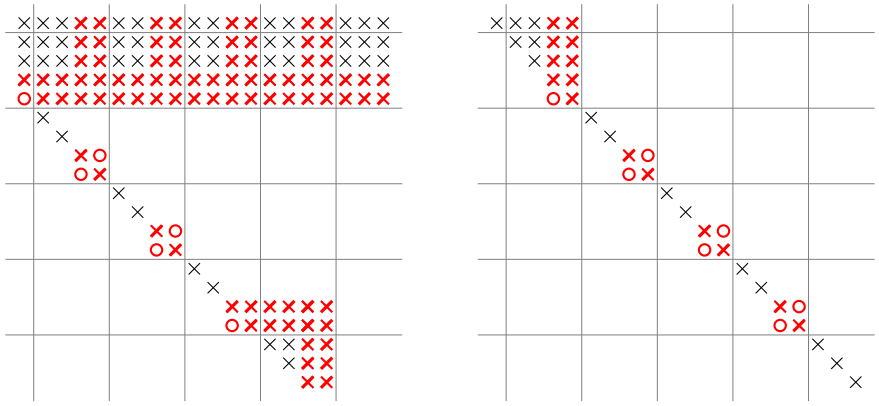

FIG. 3.4. *Generic sparsity structure of a preprocessed matrix pair* (A, B) *after iteration* $c = 1$, $r = 5$ *of Algorithm 1. Modified entries are highlighted and entries that at some point during the iteration were nonzero but at the end are zero are shown as circles.*

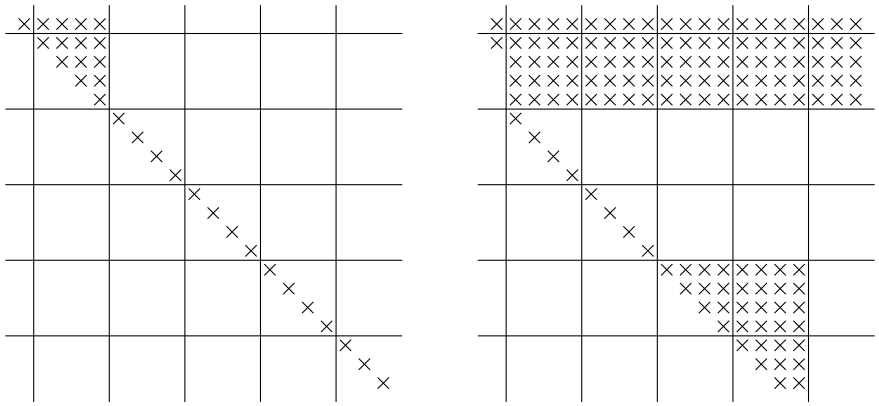

FIG. 3.5. *Generic sparsity structure of a preprocessed matrix pair* (A, B) *after iteration* $c = 1$ *of Algorithm 1.*

can be eliminated as well, but in general this cost reduction is more modest since the structure of A is rapidly destroyed.

4. Cache-Efficient Algorithms. Algorithm 1 accesses memory with a large stride and performs only a few arithmetic operations for every accessed matrix entry. Following previous work on HT reduction for general inputs [9], we apply in this section a set of standard techniques that reduce the number of cache and TLB misses as well as the volume of communication between cache and main memory.

In Section 4.1 we explain how to rearrange the steps of Algorithm 1 into a more cache-friendly pipelined algorithm. In addition, the pipelined algorithm exploits the structure of B. In Section 4.2 we further improve on the pipelined algorithm by blocking for the memory hierarchy and avoiding a redundancy that exists in the accumulation of Q and Z.

4.1. Pipelined algorithm. The steps of Algorithm 1 can be reordered such that all bulges stemming from iteration c are chased together in a pipelined fashion. This is made possible thanks to the associativity of matrix multiplication. In floating

point arithmetic the different ordering leads to differences in the computed results but both the basic and the pipelined variants are backwards stable. We refer to Section 4.1.2 below for descriptions of the computational kernels that appear in the pipelined algorithm.

Algorithm 2: $A, B, Q, Z \leftarrow$ `HTPipelined`(n, d, A, B, Q, Z)

```
   /* For each column left to right.                                    */
 1 for c ← 1 to dn − 2 do
       /* Locate column c of A.                                         */
 2     r1, r2 ← c + 1, min{dn, c + n};
       /* Zero out column c of A using row rotations.                   */
 3     G ← LeftZero(A, r1, r2, c);
 4     Left(G, A, r1, r2, c + 1, dn);
 5     Right(G, Q, 1, dn, r1, r2);
       /* Move from A to B.                                             */
 6     c1, c2 ← r1, r2;
       /* Generate and zero out bulges in B.                            */
 7     G ← LeftRightZero(G, B, r1, r2, c1, c2);
 8     Right(G, B, 1, r1 − 1, c1, c2);
 9     Right(G, Z, 1, dn, c1, c2);
       /* Chase bulges until they disappear.                            */
10     loop
           /* Move from B to A.                                         */
11         r1, r2 ← c1 + n, min{dn, c2 + n};
           /* Update A using column rotations.                          */
12         Right(G, A, 1, min{dn, r1 − 1}, c1, c2);
           /* Exit if all bulges have disappeared.                      */
13         if r1 > dn then exit loop;
           /* Generate and zero out bulges in A.                        */
14         G ← RightLeftZero(G, A, r1, r2, c1, c2);
15         Left(G, A, r1, r2, c2 + 1, min{dn, (d − 1)n + c});
16         Right(G, Q, 1, dn, r1, r2);
           /* Move from A to B.                                         */
17         c1, c2 ← r1, r2;
           /* Generate and zero out bulges in B.                        */
18         Right(G, Z, 1, dn, c1, c2);
```

The pipelined Algorithm 2 uses the concept of a *sequence of rotations*, that is, a sequence $\mathcal{G} = \langle G_1, G_2, \ldots, G_m \rangle$ consisting of m rotations acting on adjacent pairs of rows or columns. More specifically, for an *increasing* sequence $\mathcal{G}$, the rotation G_k acts on rows/columns $i + k - 1$ and $i + k$ for some constant base index i. In other words, the rotation sequence $\mathcal{G}$ affects the $m + 1$ rows or columns $i \colon i + m$. For a *decreasing* sequence $\mathcal{G}$, rotation G_k acts on rows or columns $i + m - k$ and $i + m - k + 1$. Again, the sequence affects the rows or columns $i \colon i + m$. We only encounter decreasing sequences in what follows.

The pipelined algorithm exploits the structure of B and the mostly zero columns at the east side of A. The variables r_1, r_2, c_1, and c_2 keep track of the submatrix (of A or B, depending on the context) that currently contains (or is about to contain) the

set of bulges. The outer loop iterates over the columns from left to right. Column c of A is zeroed out on line 3, and the resulting sequence of rotations is immediately applied to A and accumulated into Q. The resulting bulges in B are generated and immediately zeroed out on line 7 followed by further updates of B and accumulation into Z. Entering the loop on line 10, the column rotations are applied to A on line 12. A sequence of row rotations is constructed on line 14 to zero out the resulting bulges in A followed by updates of A and accumulation into Q. The affected portion of B is an identity matrix and therefore any row rotation generates a bulge that is zeroed out by the same rotation applied to the columns. Hence, the net effect on B is nil and there is no need to explicitly generate the bulges or even reference B. The only thing that remains to do is to accumulate the row rotations (turned column rotations) into Z.

4.1.1. Column striping. Assuming that the matrices are represented in the column-major storage format, a column rotation can be implemented such that it accesses two contiguous streams of memory, but a row rotation will necessarily access memory with a large stride. Since the pipelined algorithm consistently applies sequences of rotations, row rotations can be more efficiently applied by partitioning the columns into stripes and apply a sequence fully to each stripe in turn. In this way, any matrix entries that are brought into the cache by the application of one rotation is likely to still be present in the cache by the time the next rotation is applied. Such a column striping technique does not only improve the cache reuse but also reduces the number of TLB misses [9].

4.1.2. Computational kernels introduced in Algorithm 2. Algorithm 2 introduces five computational kernels, which we describe below.

- $\mathcal{G} \leftarrow$ `LeftZero`(A, r_1, r_2, c)
  Constructs a decreasing sequence $\mathcal{G}$ consisting of $r_2 - r_1$ row rotations that zeroes out all entries except the first one in the column vector $A(r_1 \colon r_2, c)$. The vector is updated in the process.
- `Left`$(\mathcal{G}, A, r_1, r_2, c_1, c_2)$
  Applies a decreasing sequence consisting of $r_2 - r_1$ row rotations $\mathcal{G}$ to the submatrix $A(r_1 \colon r_2, c_1 \colon c_2)$.
- `Right`$(\mathcal{G}, A, r_1, r_2, c_1, c_2)$
  Applies a decreasing sequence consisting of $c_2 - c_1$ column rotations $\mathcal{G}$ to the submatrix $A(r_1 \colon r_2, c_1 \colon c_2)$.
- $\hat{\mathcal{G}} \leftarrow$ `LeftRightZero`$(\mathcal{G}, B, r_1, r_2, c_1, c_2)$
  Applies a decreasing sequence consisting of $r_2 - r_1$ row rotations $\mathcal{G}$ to the upper triangular and square submatrix $B(r_1 \colon r_2, c_1 \colon c_2)$ while at the same time constructing and applying a decreasing sequence consisting of $c_2 - c_1$ column rotations $\hat{\mathcal{G}}$ that zeroes out the bulges created by $\mathcal{G}$. The updated submatrix is again upper triangular.
- $\hat{\mathcal{G}} \leftarrow$ `RightLeftZero`$(\mathcal{G}, A, r_1, r_2, c_1, c_2)$
  Applies a decreasing sequence consisting of $c_2 - c_1$ column rotations $\mathcal{G}$ to the upper trapezoidal submatrix $A(r_1 \colon r_2, c_1 \colon c_2)$ while at the same time constructing and applying a decreasing sequence consisting of $r_2 - r_1$ row rotations $\hat{\mathcal{G}}$ that zeroes out the bulges created by $\mathcal{G}$. The updated submatrix is again upper trapezoidal.

4.2. Blocked algorithm. The pipelined algorithm has better cache reuse than the basic algorithm yet it still applies only a constant number of floating point op-

Algorithm 3: $A, B, Q, Z \leftarrow \texttt{HTBlocked}(n, d, A, B, Q, Z)$

$\check{c} \leftarrow 1$;
while $\check{c} \leq dn - 2$ **do**
  Choose any $\hat{c} \in \{\check{c}, \ldots, dn-2\}$;
  `/* * * Phase I: Panel factorization * * *` */
  `/* For each iteration` c `in the current panel.` */
  for $c \leftarrow \check{c}$ **to** $\hat{c}$ **do**
    `/* Locate column` c `of` A`.` */
    $r_1, r_2 \leftarrow c+1, \min\{dn, c+n\}$;
    `/* Zero out column` c `of` A `using row rotations.` */
    $\mathcal{G}_c^Q \leftarrow \texttt{LeftZero}(A, r_1, r_2, c)$;
    $\texttt{Left}(\mathcal{G}_c^Q, A, r_1, r_2, c+1, dn)$;
    `/* Move from` A `to` B`.` */
    $c_1, c_2 \leftarrow r_1, r_2$;
    `/* Generate and zero out bulges in` B`.` */
    $\mathcal{G}_c^Z \leftarrow \texttt{LeftRightZero}(\mathcal{G}_c^Q, B, r_1, r_2, c_1, c_2)$;
    $\texttt{Right}(\mathcal{G}_c^Z, B, \check{c}+1, r_1-1, c_1, c_2)$;
    $\mathcal{G} \leftarrow \mathcal{G}_c^Z$;
    `/* Chase bulges until they disappear.` */
    for $k \leftarrow 1, 2, \ldots$ **do**
      `/* Move from` B `to` A`.` */
      $r_1, r_2 \leftarrow c_1+n, \min\{dn, c_2+n\}$;
      `/* Update` A `using column rotations.` */
      $\texttt{Right}(\mathcal{G}, A, \check{c}+1, \min\{dn, r_1-1\}, c_1, c_2)$;
      `/* Exit if all bulges have disappeared.` */
      if $r_1 > dn$ **then** exit loop ;
      `/* Generate and zero out bulges in` A`.` */
      $\mathcal{G}_{c,k}^{Q/Z} \leftarrow \texttt{RightLeftZero}(\mathcal{G}, A, r_1, r_2, c_1, c_2)$;
      $\texttt{Left}(\mathcal{G}_{c,k}^{Q/Z}, A, r_1, r_2, c_2+1, \min\{dn, (d-1)n+c\})$;
      $\mathcal{G} \leftarrow \mathcal{G}_{c,k}^{Q/Z}$;
      `/* Move from` A `to` B`.` */
      $c_1, c_2 \leftarrow r_1, r_2$;
    $k_c \leftarrow k-1$;
  `/* * * Phase II: Delayed updates * * *` */
  Accumulate G_c^Q for $c \in \{\check{c}, \ldots, \hat{c}\}$ into matrices U_i^Q for $i \in \{1, \ldots, n_Q\}$;
  Accumulate G_c^Z for $c \in \{\check{c}, \ldots, \hat{c}\}$ into matrices U_i^Z for $i \in \{1, \ldots, n_Z\}$;
  Accumulate $G_{c,k}^{Q/Z}$ for $c \in \{\check{c}, \ldots, \hat{c}\}$ into matrices $U_i^{Q/Z}$ for $i \in \{1, \ldots, n_{Q/Z}\}$;
  Apply $U_i^{Q/Z}$ to Q for all i;
  Copy columns $\check{c}+n\colon \min\{dn, \hat{c}+n\}$ from Q to Z;
  Apply U_i^Q to Q and U_i^Z to Z for all i;
  Apply $U_i^{Q/Z}$ followed by $U_{i'}^Z$ to rows $1\colon \check{c}$ of A for all i and i';
  Apply U_i^Z to rows $1\colon \check{c}$ of B for all i;
  `/* Advance to the next panel.` */
  $\check{c} \leftarrow \hat{c}+1$;

erations (approximately 3 to 6) per accessed matrix entry and iteration of the outer loop. Since each iteration accesses most of the matrices, it follows that the arithmetic intensity will be small for sufficiently large matrices. Significant improvements of the cache reuse therefore necessarily require the reordering of operations spanning several iterations of the outer loop. The main difficulty is to handle the large number of data dependencies that are caused by the intrinsic interleaving of row and column rotations.

Fortunately, the accumulation of rotations into Q and Z is performed exclusively using column rotations and can be effectively reorganized for improved cache reuse. However, much of the updates of A and some of the updates of B involve interleaved row and column updates that cannot be easily separated. Some improvements of the cache reuse are nonetheless possible.

Related works such as [9, 11, 12] describe how to apply a blocking technique to the accumulation of Q and Z as well as to those parts of A and B that are only affected by column rotations. Recent work by one of the authors demonstrates how to block also the updates of A (and B) [10]. For the latter technique to be effective, however, the lower bandwidth should be much smaller than the size of the matrices, which is typically not the case here since the degree tends to be small relative to n and hence the bandwidth, n, is large relative to the size, dn, of the matrices.

Let us now isolate the parts of the matrices which are updated only by column rotations from those parts which are updated by both row and column rotations. This will help us understand the blocked algorithm that follows. As mentioned above, all the accumulations into Q and Z are performed using column rotations. The processing of the $(2,2)_c^B$ block of B, via the kernel `LeftRightZero` on line 7 (of Algorithm 2) involves row and column rotations. The remaining updates of B are all based on column rotations. The processing of the blocks $(i,j)_c^A$ for $i = 2, 3, \ldots, N_c$ with N_c defined in (3.1) and $j = i, \ldots, \min\{N_c, d\}$ involves both row and column rotations. The remaining updates of A, that is, to the blocks $(1,j)_c^A$ for $j = 2, 3, \ldots, N_c$ are all based on column rotations. In summary, only the submatrices $A(c+1\colon dn, c+1\colon dn)$ and $B(c+1\colon c+n, c+1\colon c+n)$ involve both row and column rotations, the rest only column rotations.

The blocked Algorithm 3 improves on the pipelined algorithm from Section 4.1 in part by introducing blocking and in part by avoiding a redundancy that occurs in the accumulation of Q and Z, the latter of which is explained in Section 4.2.1. The blocked algorithm zeroes out columns of A from left to right one column panel $\check{c}\colon \hat{c}$ at a time. Each iteration of the outer loop on line 2 reduces one panel. Each block iteration consists of two phases. In Phase I ("panel factorization"), the columns $\check{c}\colon \hat{c}$ are zeroed out and the resulting bulges are chased using what amounts to the pipelined algorithm. This is accomplished by the outer for-loop on line 4. Only the submatrices $A(\check{c}+1\colon dn, \check{c}+1\colon dn)$ and $B(\check{c}+1\colon \check{c}+n, \check{c}+1\colon \check{c}+n)$ are updated in Phase I since the other updates are not required to construct the rotations. In Phase II ("delayed updates"), the rotations are regrouped and accumulated into small unitary matrices, which are then applied using matrix multiplication to Q, Z, $A(1\colon \check{c}, \check{c}+1\colon dn)$ and $B(1\colon \check{c}, \check{c}+1\colon \hat{c}+n)$.

4.2.1. Elimination of redundancy. The transformation matrices Q and Z obtained from the preprocessing step (Section 2) are actually identical except for their top left $n \times n$ block. Due to the identities on the main block diagonal of B, many of the row and column rotations are identical too (see Section 4.1). As it turns out, while Q and Z gradually diverge over the course of the pipelined and blocked algorithms, they

maintain a shrinking number of identical columns. Specifically, the number of columns shared by Q and Z shrinks by one column for every column reduced. Immediately before reducing column c, the matrices Q and Z can be partitioned into column blocks

$$Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \quad \text{and} \quad Z = \begin{bmatrix} Z_1 & Z_2 \end{bmatrix}$$

such that both Q_1 and Z_1 have $\min\{dn, n+c-1\}$ columns and $Q_1 \neq Z_1$ (in general) and $Q_2 = Z_2$ (always). During the reduction of column c, a decreasing sequence of rotations will engage the right-most parts of Q_1 as well as the left-most column of Q_2. Similarly, another rotation sequence will engage the right-most parts of Z_1 and the left-most column of Z_2. All other sequences affect entries exclusively in Q_2 and Z_2 and, crucially, do not touch the left-most columns in neither Q_2 nor Z_2. Moreover, the same rotations are applied to both Q_2 and Z_2. It follows that at the end of iteration c, the left-most columns of Q_2 and Z_2 will differ but the other columns will remain identical. This allows us to avoid the redundant accumulation of rotations into both Q_2 and Z_2 by only accumulating into Q_2. Prior to iteration c, we copy the left-most column of Q_2 into the corresponding column of Z_2. See Figure 4.1 for a pictorial representation.

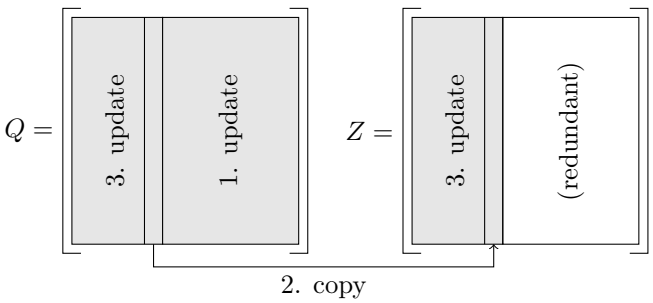

FIG. 4.1. *Illustration of the elimination of redundancy in the computation of Q and Z. First, rotations shared by Q and Z are applied to Q only. Then, a column is copied from Q to Z. Finally, rotations unique to Q and Z are applied to the respective matrices.*

4.2.2. Regrouping rotations for increased cache reuse. We increase the cache reuse in Phase II by adopting a known technique for applying multiple overlapping sequences of rotations [12]. This technique exploits the commutativity of rotations acting on disjoint sets of rows or columns and rearranges the rotations into diamond-shaped groups, each of which can be applied cache-efficiently.

However, we cannot apply this technique directly since the rotations obtained from one bulge-chasing sweep do not form one but several non-overlapping sequences. Fortunately, these shorter sequences can be merged by inserting identity rotations to artificially create one long sequence.

Consider the first three iterations of the outer for-loop over c for the case $d = 4$, $n = 5$. Let G_k^c denote the row rotation constructed in iteration c to annihilate $a_{k,c}$ using $a_{k-1,c}$ as a pivot. This rotation acts on rows $k-1$ and k. Let I_k denote an *identity rotation* acting on rows $k-1$ and k. Figure 4.2(a) illustrates the row rotations and their dependencies for the first three column reductions. Each vertex represents a rotation and each arc represents a dependence caused by non-commutativity. Any topological ordering of the graph corresponds to a valid reordering of the rotations (and vice versa). The region with a dashed outline is an example of how to group

rotations into a diamond-like shape. Such a group has the property that there is no directed path leaving and re-entering the region. This implies that the rotations in the group can be applied in sequence without any other rotation in between. In particular, the rotations in a group can be explicitly accumulated into a unitary matrix and applied using matrix multiplication.

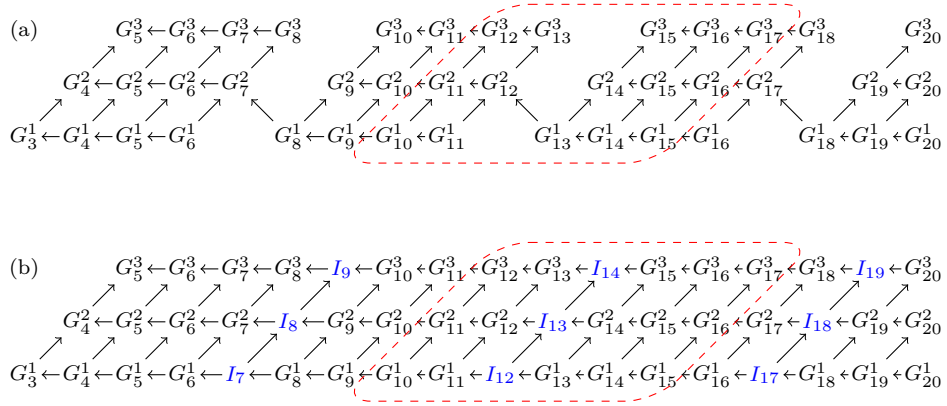

FIG. 4.2. *Row rotations and their dependencies for the first three iterations of the case $d = 4$, $n = 5$. In (a), the rotations G_k^c for $c = 1, 2, 3$ are shown together with their dependencies. The region with a dashed outline exemplifies how to create a group of rotations that can be applied separately from the others. In (b), nine identity rotations have been added to make the graph structure more regular. Any ordering of the (proper) rotations allowed by (b) is also allowed by (a). Hence, replacing (a) by (b) does not compromise correctness.*

The graph in Figure 4.2(a) has "gaps" (e.g., rotation G_7^1 does not exist). In Figure 4.2(b), the gaps have been filled with identity rotations that have no effect other than to make the structure of the graph more regular. The graph in (b) is a restriction of the graph in (a) in the sense that any topological ordering of (b) contains a topological ordering of (a). The graph structure in Figure 4.2(b) is identical to the one in [12] and hence the technique described therein applies directly.

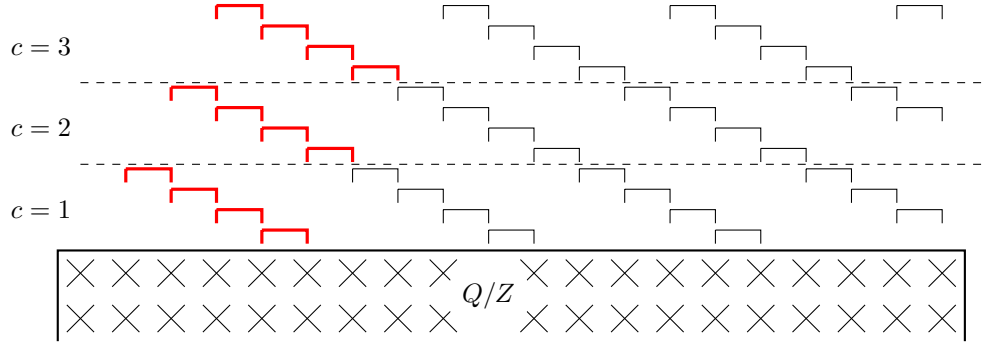

FIG. 4.3. *Illustration of the rotations stemming from the processing of the first three columns of A for the case $d = 4$, $n = 5$. Each rotated square bracket represents a column rotation. A red symbol represents* two different *rotations (one rotation associated with Q and another associated with Z) while a black symbol represents a rotation shared between both Q and Z. The horizontal dashed lines group the rotations based on the iteration from which they originate.*

Another viewpoint is illustrated in Figure 4.3. The bottom part of the figure

symbolizes a matrix, either Q or Z, on top of which all the rotations are shown as rotated square brackets. Rotations closer to the matrix in Figure 4.3 should be applied first, unless they do not overlap in which case they commute and can be applied in any order. The figure also illustrates which of the rotations are shared between Q and Z (black), and which are unique to Q and Z (thick red). Hence, a rotation shown in red actually represents *two* rotations, one acting on Q and the other on Z. The insertion of identity rotations and the regrouping of rotations are illustrated in Figure 4.4.

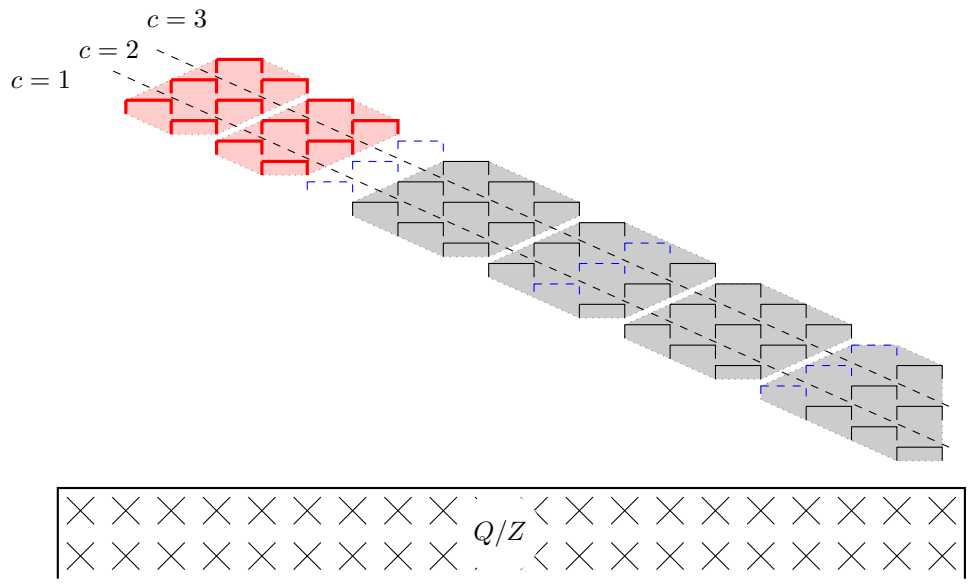

FIG. 4.4. *Illustration of merged sequences of rotations from the first three iterations for the case $d = 4$ and $n = 5$. Inserted identity rotations are shown using dashed blue lines. The shaded regions illustrate an alternative grouping of the rotations into diamond-shaped groups for increased cache reuse.*

4.2.3. Accumulating transformations. The sparsity structure of all the transformation matrices (Q, Z, and the small matrices U_i in Algorithm 3) are thickly banded matrices with some additional structure within the band. More precisely, they each take the form of a skyline matrix where the top and bottom skylines are non-decreasing. Formally, let t_j and b_j denote the row index of the first respectively last (structural) nonzero in column j. Then $t_{j+1} \geq t_j$ and $b_{j+1} \geq b_j$. By explicitly keeping track of the t_j and b_j during the accumulation procedure, it is straightforward and inexpensive to determine the minimal range of rows to which a given rotation must be applied and thereby optimally reduce the number of arithmetic operations.

5. Parallel Algorithm. Even though our proposed blocked algorithm requires less computation and communication through the memory hierarchy than the conventional algorithm it is still quite costly in terms of the number of flops with an arithmetic complexity of $\mathcal{O}(d^3n^3)$. We have therefore constructed a parallel algorithm for multicore-based systems implemented using threads.

The parallel algorithm is structured similarly to Algorithm 3 and consists of iterations over panels. Each iteration consists of two phases, each of which is executed in parallel. Successive phases are separated with a barrier synchronization. The parallelization of Phase II is described in Section 5.1 and the (more technical) parallelization

of Phase I is described in Section 5.2.

5.1. Parallel Phase II. The parallel algorithm for Phase II consists of an accumulation step followed by an update step with a barrier synchronization in between. In the accumulation step, the rotations are regrouped and each rotation group is accumulated into a small unitary matrix. The accumulations of different groups are independent and the accumulation step is therefore perfectly parallel.

In the update step, the groups containing shared rotations are first applied to Q. This is then followed by the copying of a set of columns from Q to Z as part of the elimination of redundancy. The update step ends by applying the rotation groups that contain non-shared rotations to Q and Z. Also the update step is perfectly parallel in the sense that the rows can be independently updated. However, the load is not uniformly distributed over the rows due to the sparsity structure of Q and Z. We address this problem by calculating ahead of time the number of flops per row and partition the rows of Q and Z such that the flops are approximately balanced over the threads.

5.2. Parallel Phase I. The parallel algorithm for Phase I consists of multiple iterations, each of which interleaves row and column rotations with the zeroing out of matrix entries.

The steps in each iteration of the loop in Phase I (see Algorithm 3) can be rewritten in terms of coarse operations acting on the blocks of the natural block partitioning. In doing so we also take the opportunity to further increase the cache reuse by exploiting the fact that all the blocks in the upper triangle are subject to both a row sequence and a column sequence. By fusing the application of these two sequences together into a single kernel we can reduce its communication volume by up to 50%. The new kernel is defined as follows:

- `RightLeft`$(\mathcal{G}, \widetilde{\mathcal{G}}, A, r_1, r_2, c_1, c_2)$
  Applies a decreasing sequence of row rotations $\mathcal{G}$ and a decreasing sequence of column rotations $\widetilde{\mathcal{G}}$ to the submatrix $A(r_1 \colon r_2, c_1 \colon c_2)$. The number of row rotations is $r_2 - r_1$ and the number of column rotations is $c_2 - c_1$.

The new version of Phase I is given in Algorithm 4.

Algorithm 4: Phase I expressed using operations on blocks.

```
for c ← č to ĉ do
    LeftZero on (2,1)_c^A;
    LeftRightZero on (2,2)_c^B;
    for i ← 3 to N_c do
        RightLeftZero on (i,i−1)_c^A;
    for j ← 3 to N_c do
        RightLeft on (2,j)_c^A;
    for i ← 3 to N_c do
        for j ← i to min{d, N_c} do
            RightLeft on (i,j)_c^A;
```

With each operation in Algorithm 4 executed by a single thread, the number of tasks would be limited to $\mathcal{O}(d^2)$ and the average degree of concurrency would be only $\mathcal{O}(d)$. Therefore, a large degree would be necessary to saturate a typical

multicore-based system. Since the degrees are predominantly small in practice we further decompose each operation into finer grained tasks. This in turn requires an additional set of computational kernels, which we define in Section 5.2.1. The task decomposition, dependencies, and scheduling of the tasks is described in Section 5.2.2.

5.2.1. Task decomposition of Phase I. The task decomposition that we use is to a great extent induced by a partitioning of each block into square sub-blocks defined by a sub-block size b and aligned with the top left corner of the block. This in turn induces a partitioning of the sequences of rotations into sub-sequences of length b. The reason for square sub-blocks is to ensure that a sub-block of the row rotations that become column rotations align with the sub-block partitioning of the columns.

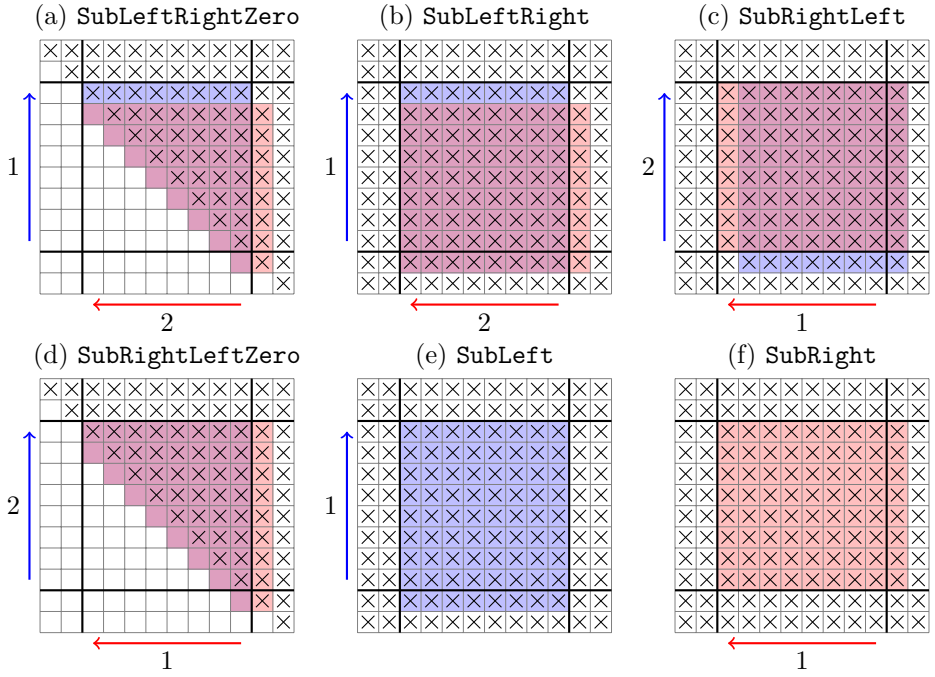

FIG. 5.1. *Computational kernels employed in the parallel implementation of Phase I. Each subfigure illustrates one sub-block and its immediate surroundings indicated by thick lines. Empty cells should be interpreted as zeros and the others as nonzeros. The colored regions indicate which entries are updated by a sequence of row rotations (blue), respectively, column rotations (red). The arrows indicate the direction of and the order in which the sequences are applied.*

The `LeftZero` operation involves little work and is not decomposed into tasks in order to avoid the associated scheduling overhead. (Moreover, the operation is inherently sequential.) Instead it is merged with the `LeftRightZero` operation that follows.

The `RightLeft` operation takes two sequences as input: one row sequence and one column sequence. Figure 5.1(c) illustrates the kernel that forms the basis of the implementation. Shown is one of the sub-blocks and its immediate surroundings. The kernel applies to the sub-block first the column sequence and then the row sequence. Note that the column (row) rotations affect one additional column (row) to the east (south). The region affected by the row rotations has also been shifted one column to the east since the left-most column in the sub-block is not fully up-to-date with

respect to the column rotations. Special cases occur at the south and east boundaries of the block since the updates do not extend beyond the block. The task graph and the relationships between the tasks and the sub-blocks are illustrated in Figure 5.2(a). With $n_b = \lceil n/b \rceil$ sub-blocks in either dimension, the task graph contains n_b^2 tasks and has depth $2n_b - 1$.

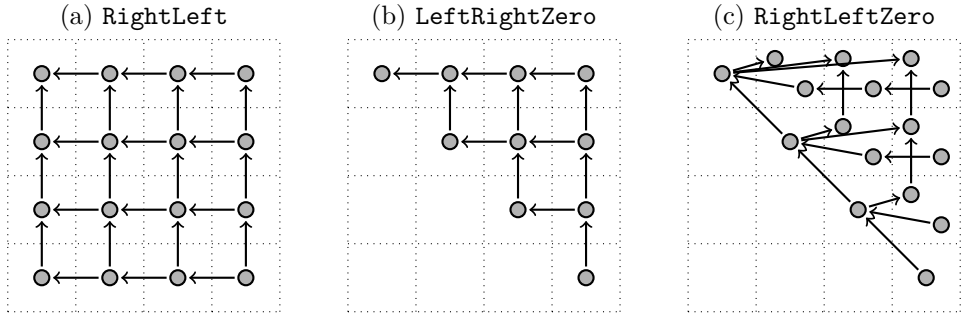

FIG. 5.2. *Task graphs for the three operations in the parallel implementation of Phase I.*

The `LeftRightZero` operation is built upon the two computational kernels illustrated in Figures 5.1(a–b) and its task graph is illustrated in Figure 5.2(b). The `SubLeftRightZero` kernel applies to the sub-blocks on the diagonal and the `SubLeftRight` kernel to the sub-blocks above the diagonal. Again, the regions affected by row, respectively, column rotations are not perfectly aligned to the sub-blocks, as is evident from Figure 5.1. With n_b sub-blocks in either dimension, the task graph contains $n_b(n_b + 1)/2$ tasks and has depth $2n_b - 1$.

The `RightLeftZero` operation is built upon the three computational kernels illustrated in Figures 5.1(d–f). Interestingly, even though the name and functionality of this operation is similar to `LeftRightZero`, they are not symmetric and as a consequence their task decompositions are rather different. The task graph of `RightLeftZero` is illustrated in Figure 5.2(c) and the tasks on the diagonal use the kernel `SubRightLeftZero`. Each sub-block in the strictly upper triangular part contains one `SubRight` task (closer to the bottom right corner) and one `SubLeft` task (closer to the upper left corner). With n_b sub-blocks in either dimension, the task graph contains n_b^2 tasks and has depth $n_b + 1$.

5.2.2. Scheduling Phase I to promote cache reuse. In the scheduling of Phase I we also attempt to increase the cache reuse. Specifically, the parallel implementation of Phase I begins by explicitly constructing the task graph for the first iteration. The tasks are then statically scheduled using the critical path heuristic (largest height first). We then *reuse* the same schedule for all subsequent iterations in the phase. One consequence of this approach is that each thread is statically assigned a subset of the tasks and by extension also a subset of the sub-blocks. This is what increases the cache reuse.

Figure 5.3 illustrates the task graph associated with one iteration of the first instance of Phase I for a polynomial of degree four. The sub-block size is set to one fourth of the block size. The labels on the nodes indicate the kernel used and are the same as the labels used in Figure 5.1. The graph contains 170 tasks and has depth 14 which yields a 12.143 average degree of concurrency. This should be contrasted to the case of doing no decomposition of the blocks at all (i.e., setting the sub-block size equal to the block size). In that case the graph would have only 11 tasks and a depth

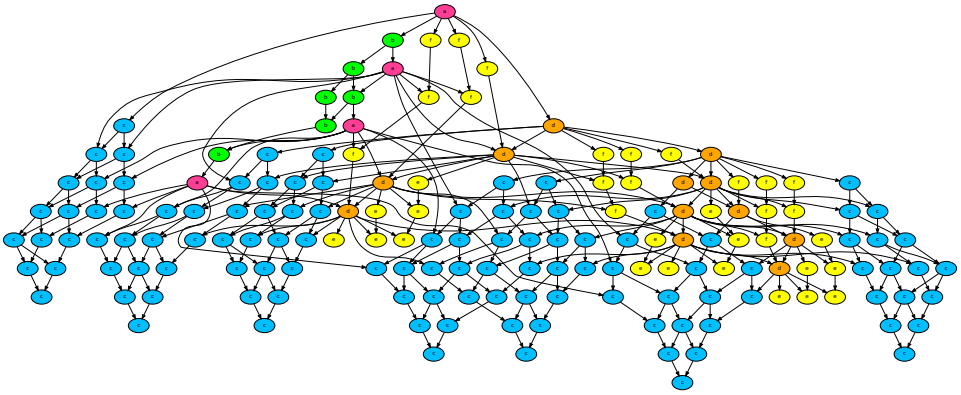

FIG. 5.3. *Task graph for one iteration of Phase I for a polynomial of degree four and with four sub-blocks in either dimension of each block. The node colors and labels indicate the kernel used by the task and correspond to the labels in Figure 5.1.*

of 5 for a 2.2 average degree of concurrency. (Since consecutive iterations partially overlap in practice, the average degree of concurrency is actually slightly larger in both cases.)

6. Experiments. We developed two separate implementations for different types of systems: a sequential implementation for workstations following Algorithm 3 and a thread-based implementation based on the parallel algorithm described in Section 5. The sequential algorithm is intended for use on workstations via a MATLAB MEX interface and takes advantage of potential implicit parallelism by calling MATLAB's BLAS library. The parallel implementation is intended for use on multicore-based HPC systems.

We measured the performance of our codes on two systems (one workstation and one HPC system) with different characteristics:

- *Workstation.* The workstation has an Intel Core i7-2600 processor with four cores and a nominal clock frequency of 3.4 GHz.
- *HPC system.* The HPC system is a cluster consisting of interconnected shared-memory nodes. Each node consists of four AMD Opteron 6238 processors each with twelve cores split into two NUMA domains. The nominal clock frequency is 2.6 GHz.

Both systems run a Linux-based operating system. All experiments were performed in double precision (64-bit) real floating point arithmetic. All execution times are given in seconds.

6.1. Results from the workstation. On the workstation we measured the execution time of our MEX code with implicit parallelism via the parallel BLAS used in MATLAB and compared it against the HT reduction function `hess` in MATLAB, our parallel code with $p \in \{1, 2, 3, 4\}$ threads, and a prototype implementation, `DGGHR3`, of the blocked one-stage HT reduction algorithm described in [9]. The results were obtained using MATLAB version R2011b for Linux.

The results are summarized in Table 6.1 for $n \in \{500, 1000\}$ and $d \in \{2, 3, 4\}$. The times are given in seconds. The column labeled `hess/mex` shows the ratio of the execution time of the `hess` function to the execution time of our MEX code. The

TABLE 6.1

Comparison between the sequential `mex` *code, MATLAB's* `hess` *function and the parallel implementation on the workstation.*

		MATLAB mex			Parallel				
n	d	`mex`	`hess`	`hess/mex`	$p=1$	2	3	4	`hess`/$p=4$
500	2	1.2	5.7	4.6	2.0	1.3	1.1	0.91	6.3
500	3	3.6	21	5.9	5.6	3.4	2.6	2.2	9.4
500	4	8.2	55	6.8	12	7.2	5.6	4.7	11.7
1000	2	14	56	4.0	19	11	9.6	8.2	6.9
1000	3	40	192	4.8	49	27	22	19	10.0
1000	4	85	506	5.9	101	57	46	41	12.3

TABLE 6.2

Execution time of the prototype implementation `DGGHR3` *of the blocked one-stage HT reduction algorithm described in [9] linked with multi-threaded BLAS on the workstation.*

n	d	$p=1$	2	3	4
500	2	2.3	1.9	1.8	1.8
500	3	7.3	6.0	5.7	5.6
500	4	17	14	13	13
1000	2	17	14	13	13
1000	3	56	46	43	41
1000	4	133	107	99	97

column labeled `hess`/$p=4$ shows the ratio of the execution time of the `hess` function to the execution time of our parallel code with $p=4$ threads.

The size of the improvement depends on both n and d. The observed improvements of the MEX code ranges from 4.0 (for $n=1000$, $d=2$) to 6.8 (for $n=500$, $d=4$). The observed improvements of the parallel code with four threads range from 6.3 (for $n=500$, $d=2$) to 12.3 (for $n=1000$, $d=4$). As expected, the increasing sparsity of the linearization for large degrees leads to increased improvements over the generic `hess` function. A larger size of the coefficient matrices does seem to slightly decrease the improvement of the MEX code while on the other hand increasing slightly the improvement of the parallel code.

We obtained a prototype implementation, `DGGHR3`, of the blocked one-stage algorithm presented in [9], which represents the state of the art for generic HT reduction. The execution times for a varying number of threads are shown in Table 6.2, which can be directly compared column by column to Table 6.1 (parallel results). `DGGHR3` significantly outperforms the `hess` function in MATLAB and performs only slightly worse than our structured code. The ratios of the execution times of `DGGHR3` to our parallel code range from 0.9 ($n=1000$, $d=2$, $p=1$) to 2.8 ($n=500$, $d=4$, $p=4$) with a mean of 1.8.

6.2. Results from the HPC system. On the HPC system we measured the performance of our parallel code and compared it against the LAPACK `DGGHRD` generic HT reduction routine as well as `DGGHR3`. The matrix B must be upper triangular on input, so we performed a structure-exploiting pre-processing step that reduces the top left $n \times n$ dense block of B to upper triangular form before calling `DGGHRD` or `DGGHR3`.

6.2.1. Total execution time. The `DGGHRD` routine in LAPACK is based on Level 1 BLAS operations without explicit parallelism and thereby does not scale on multicore-based systems. Table 6.3 summarizes the total execution times for LAPACK linked with OpenBLAS 0.2.8 (sequential) for $n \in \{500, 1000, 1500, 2000\}$ and $d \in \{2, 3, 4\}$.

TABLE 6.3

The execution time of LAPACK's generic HT reduction routine `DGGHRD` *following a custom structure-exploiting reduction of B to upper triangular form on the HPC system.*

$n = 500$			1000			1500			2000		
$d = 2$	3	4	$d = 2$	3	4	$d = 2$	3	4	$d = 2$	3	4
11	43	105	106	361	894	362	1240	3052	886	3060	7693

TABLE 6.4

The parallel execution time (including both the pre-processing step and the bulge-chasing part) of our polynomial HT reduction algorithm on the HPC system.

	$n = 500$			1000			1500			2000		
p	$d = 2$	3	4	$d = 2$	3	4	$d = 2$	3	4	$d = 2$	3	4
1	2.9	7.8	17	24	64	151	90	233	478	248	587	2249
2	2.5	6.7	14	20	53	122	67	179	374	182	424	1524
4	1.7	4.0	8.4	13	32	73	43	108	220	121	256	864
6	1.4	3.2	6.5	11	25	56	35	84	167	99	197	628
8	1.3	2.8	5.4	9.5	21	47	31	73	141	87	169	505
10	1.3	2.7	5.0	8.7	19	43	29	66	127	80	154	442
12	1.2	2.6	4.7	8.3	18	39	27	61	119	76	142	379
14	1.2	2.6	4.8	7.8	16	36	26	59	116	72	139	351
16	1.3	2.9	5.3	7.6	15	35	25	59	118	70	139	336
18	1.3	3.1	5.7	7.2	15	36	25	60	123	67	144	340
20	1.5	3.6	6.9	7.0	16	39	24	63	138	68	155	367
22	1.5	3.8	7.4	7.1	17	41	25	68	152	67	169	394
24	1.5	4.0	8.0	7.2	17	45	24	72	167	68	179	423

These measurements should be compared to our parallel code solving problems of the same size with $1 \leq p \leq 24$ threads. The time measurements include both the pre-processing step and the bulge-chasing part of the reduction (only the latter of which is parallel) and are summarized in Table 6.4. The row with $p = 1$ can be directly compared to the timings in Table 6.3 and shows the effects of blocking and exploiting structure. The ratios of the execution times range from 3.4 (for $n = 2000$, $d = 4$) to 6.5 (for $n = 1000$, $d = 4$).

The subsequent rows in Table 6.4 show the parallel execution times for an increasingly large number of threads (up to two full processors) in our parallel code. In all cases except for $n \geq 1500$ and $d = 2$, the parallel code reaches its scalability limit in the sense that the optimal number of threads is less than the maximum of 24 used in the experiments.

The largest observed improvement of the parallel code over `DGGHRD` is 28 (for $n = 1000$, $d = 4$, $p = 16$). The largest relative speedup of the parallel code (i.e., relative improvement over $p = 1$) ranges from 2.4 ($n = 500$, $d = 2$, $p = 12$) to 6.7 ($n = 2000$, $d = 4$, $p = 16$).

Table 6.5 shows the parallel execution time of the `DGGHR3` prototype implementation with up to 12 threads (one full processor) in the multi-threaded version of the BLAS library. In all cases except for small values of n, the code reaches its scalability limit. The code scales modestly, which is expected considering that the code is only implicitly parallelized via the BLAS.

Comparing the minimum execution time of `DGGHR3` with the minimum execution time of our parallel code, we find that the relative improvement of our code ranges from 2.6 ($n = 2000$, $d = 2$, with $p = 8$ threads in `DGGHR3` and $p = 18$ threads in our parallel code) to 5.1 ($n = 500$, $d = 4$, with $p = 4$ threads in `DGGHR3` and $p = 12$ threads in our parallel code).

TABLE 6.5

The parallel execution time on the HPC system of the `DGGHR3` *prototype implementation of the blocked one-stage algorithm described in [9] (including both the pre-processing step and the actual HT reduction).*

	$n = 500$			1000			1500			2000		
p	$d = 2$	3	4	$d = 2$	3	4	$d = 2$	3	4	$d = 2$	3	4
1	3.5	12	26	26	87	204	88	288	677	208	680	1626
2	3.6	12	26	27	86	202	87	285	669	206	672	1602
4	3.5	11	24	25	78	182	79	254	594	184	595	1413
6	3.5	11	24	24	77	177	77	248	576	179	577	1368
8	3.5	11	24	24	76	176	77	245	567	177	568	1342
10	3.6	11	24	25	78	180	78	250	578	181	578	1362
12	3.7	11	25	25	78	180	79	252	577	182	578	1353

6.2.2. Analysis of the parallel bulge-chasing part. The parallel bulge-chasing (Section 5) is the most time consuming part of the reduction. In this section, we analyze the behavior of the internals of the parallel implementation.

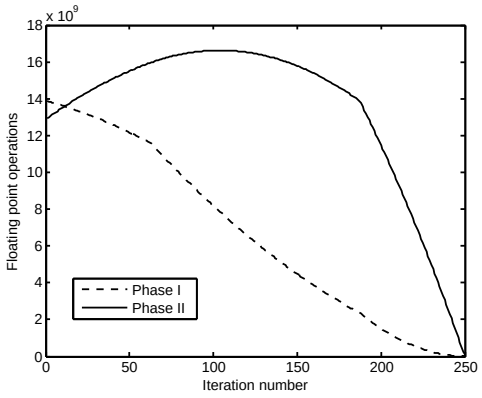

FIG. 6.1. *Distribution of flops over the two phases as a function of the iteration number* ($n = 2000$, $d = 4$).

The code was instrumented to (approximately) count the number of floating point operations performed in each phase of each panel iteration. These counts are plotted against the panel iteration number in Figure 6.1 for the case $n = 2000$, $d = 4$. (Each panel consists of 32 columns for a total of $nd/32 = 250$ iterations.)

Despite most of the flops being accounted for by Phase II, it is typically Phase I that consumes the majority of the execution time due to its low arithmetic intensity and greater complexity. Figure 6.2 shows the performance in Gflops, i.e., the ratio of the floating point operation counts from Figure 6.1 and the measured execution times for a varying number of threads. The graphs on the left side correspond to Phase I and the graphs on the right correspond to Phase II. It is observed from the latter that Phase II scales up to $p = 18$. The graphs for Phase I also indicate good scaling up to $p = 18$ but there is also a sharp decrease in the performance towards the second half of the iterations.

We investigated the source of the performance degradations observed in Phase I by measuring the per-thread time spent in computational kernels per instance of Phase I. More specifically, consider any instance of Phase I and let T_k denote the time that thread k spent computing in that instance. Then the total cost of computation is

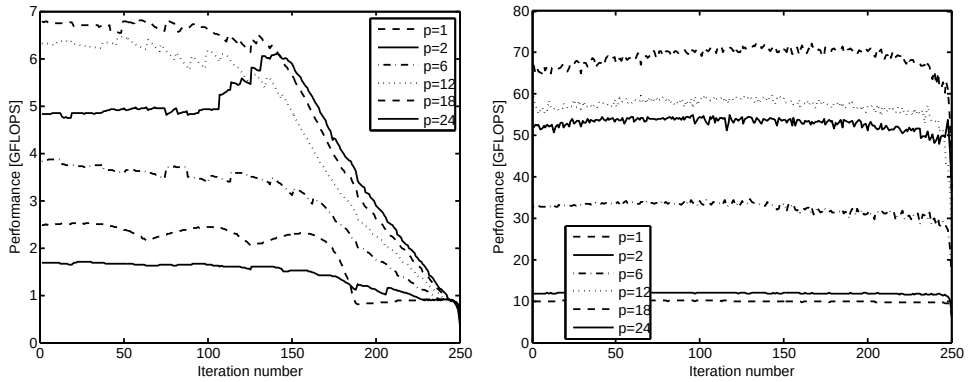

FIG. 6.2. *Performance of Phase I (left) and Phase II (right) for $n = 2000$, $d = 4$, and varying number of threads on the HPC system.*

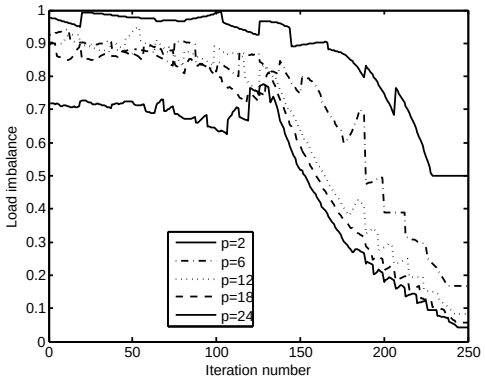

FIG. 6.3. *The load imbalance of Phase I for varying numbers of threads as a function of the iteration number ($n = 2000$, $d = 4$) on the HPC system.*

given by $\sum_{k=1}^{p} T_k$. If we assume that the work could have been perfectly distributed over the p threads then $\frac{1}{p}\sum_{k=1}^{p} T_k$ is a lower bound for the parallel execution time. The most heavily loaded thread computes for a total of $\max_k T_k$ time units. It follows that the ratio

$$L = \frac{\sum_k T_k}{p \cdot \max_k T_k}$$

in the range $(0, 1]$ is a measure of the load imbalance with $L = 1$ indicating perfect load balance. In general, the most heavily loaded thread has $1/L$ times the optimal load. The amount of load imbalance for the case $n = 2000$, $d = 4$ and a varying number of threads is shown in Figure 6.3. From the strong correlation between the increasing load imbalance and the decreasing performance of Phase I shown in Figure 6.2(left), we conclude that the load imbalance is a primary explaining factor and runtime autotuning of the sub-block size might improve the performance further.

7. Conclusion and future work. We have identified a special Fiedler linearization of matrix polynomials of arbitrary degree, which shares many of the properties

of standard companion linearizations but also allows (a) the deflation of some of the zero and infinite eigenvalues without affecting the block structure of the linearization and (b) the exploitation of the sparsity structure in the HT reduction. We have presented a backward stable high-performance parallel algorithm for the HT reduction of such linearization. Our algorithm exploits the sparsity structure of the linearization and the redundancy in the computation of the transformation matrices, and thereby requires fewer flops than existing algorithms for unstructured pairs of matrices, including the state of the art HT reduction in [9]. Special care has been taken to improve cache reuse. Experiments on both a workstation and an HPC system have demonstrated that our structure-exploiting parallel implementation can outperform both the general LAPACK routine `DGGHRD` and the prototype implementation `DGGHR3` of a general blocked algorithm.

Future work include runtime auto-tuning of the sub-block size to better balance the load, alternative parallelization schemes that address the trade-off between cache reuse and concurrency differently, and a hybrid approach that switches to a (parallel variant of) `DGGHR3` during the last iterations where the structure is less pronounced.

Acknowledgments. We thank Daniel Kressner for providing us with a copy of the `DGGHR3` prototype implementation. This research was conducted using the resources of the High-Performance Computing Center North (HPC2N).

Appendix A: Deflation of infinite and zero eigenvalues on special Fiedler linearization. If either one or both of P_d and P_0 are rank deficient, then it is possible to deflate structurally induced zero and infinite eigenvalues prior to the HT reduction. This is desirable since these eigenvalues might otherwise go partially undetected due to roundoff errors accumulated during the QZ algorithm. We show in this appendix how to transform the $dn \times dn$ Fiedler pencil $L(\lambda)$ in (1.2) into block upper triangular form

$$A + \lambda B = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & 0_{n-r_0} \end{bmatrix} + \lambda \begin{bmatrix} 0_{n-r_2} & B_{12} & B_{13} \\ 0 & B_{22} & B_{23} \\ 0 & 0 & B_{33} \end{bmatrix}, \tag{A.1}$$

where the $dn \times dn$ matrices A and B are partitioned conformably. Here r_0 denote the rank of P_0 and r_d the rank of P_d. Note that if A_{11} (or B_{33}) is singular then $\det P(\lambda) = \gamma \det L(\lambda) \equiv 0$ and hence $P(\lambda)$ is singular. When A_{11} and B_{33} are nonsingular, (A.1) reveals $n - r_0$ zero eigenvalues and $n - r_d$ infinite eigenvalues. The remaining eigenvalues are those of the $m \times m$ pencil $A_{22} + \lambda B_{22}$, where $m = (d-2)n + r - 1 + r_d$. Now to use our HT reduction algorithm, $A_{22} + \lambda B_{22}$ must retain most of the sparsity structure of the Fiedler linearization (1.2). We show that our deflation yields a deflated pencil $A_{22} + \lambda B_{22}$ of the form (when $d = 4$)

$$\left[\begin{array}{c|c|c|c} X_{n,r_d} & X_{n,n} & X_{n,n} & X_{n,r_0} \\ \hline -I_{r_d} & 0 & 0 & 0 \\ \hline 0 & -I_n & 0 & 0 \\ \hline 0 & 0 & X_{r_0,n} & 0 \end{array}\right] + \lambda \left[\begin{array}{c|c|c|c} X_{n,n} & 0 & 0 & 0 \\ \hline 0 & I_{r_d} & 0 & 0 \\ \hline 0 & 0 & I_n & 0 \\ \hline 0 & 0 & 0 & I_{r_0} \end{array}\right], \tag{A.2}$$

where $X_{m,n}$ denotes a dense matrix of size $m \times n$ and I_n is an identity matrix of size $n \times n$.

We illustrate the proposed deflation procedure for the case $d = 4$, which is large enough to clearly exhibit the general pattern. The procedure can be readily generalized to any degree $d \geq 3$. However, the extreme case $d = 2$ requires a little bit of special treatment, which we defer to the end of this appendix.

The appendix is outlined as follows. Fiedler linearizations and our special choice are defined in Section A.1. The deflation procedure consists of two steps:

1. Deflation of zero eigenvalues induced by P_0 described in Section A.2.
2. Deflation of infinite eigenvalues induced by P_d described in Section A.3.

The special treatment of the case $d = 2$ is described in Section A.4.

A.1. Fiedler linearizations. With the notation

$$M_0 = \operatorname{diag}(I_{n(d-1)}, -P_0), \quad M_d = \operatorname{diag}(P_d, I_{(d-1)n}),$$
$$M_j = \operatorname{diag}\left(I_{n(d-j-1)}, \begin{bmatrix} -P_j & I_n \\ I_n & 0 \end{bmatrix}, I_{n(j-1)}\right), \quad j = 1\colon d-1$$

and any permutation $\sigma = (i_0, i_1, \ldots, i_{d-1})$ of the indices $(0, 1, \ldots, d-1)$ we define the associated Fiedler pencil by [1]

$$L_\sigma(\lambda) = -M_{i_0} M_{i_1} \cdots M_{i_{d-1}} + \lambda M_d.$$

The pencils $L_\sigma(\lambda)$ are all linearizations of $P(\lambda)$ [1]. The Fiedler pencils we are interested in corresponds to the permutation $\sigma = (d-1, \ldots, 3, 2, 0, 1)$, that is,

$$\begin{aligned} L_\sigma(\lambda) &= -\underbrace{M_{d-1} \cdots M_2}_{\text{if } d > 2} M_0 M_1 + \lambda M_d \\ &= \begin{bmatrix} P_{d-1} & P_{d-2} & \cdots & P_1 & -I_n \\ -I_n & & & & \\ & \ddots & & & \\ & & -I_n & & \\ & & & P_0 & \end{bmatrix} + \lambda \begin{bmatrix} P_d & & & & \\ & I & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & I \end{bmatrix}. \end{aligned} \tag{A.3}$$

A.2. Deflation of zero eigenvalues induced by P_0. If P_0 has full rank then we have nothing to do in this step. Therefore, assume that P_0 has rank $r_0 < n$ and compress its rows upwards by pre-multiplication with a carefully crafted unitary matrix U^*, i.e., choose a unitary matrix U such that

$$U^* P_0 = \begin{bmatrix} \widetilde{P}_0 \\ 0 \end{bmatrix},$$

where $\widetilde{P}_0$ has r_0 rows and full rank. (This can be accomplished for instance by using a singular value decomposition of P_0 or a QR factorization with column pivoting.) Transform (A, B), where $A + \lambda B$ is the Fiedler linearization described in (A.3) by pre-multiplying the last block row with U^*. This changes the bottom right block of B from I to U^*. Restore B by post-multiplying the last block column with U. This also changes the top right block of A from $-I$ to $-U$ but this we accept. The pair (A, B) now takes the form

$$A = \left[\begin{array}{c|c|c|c|c} P_3 & P_2 & P_1 & -U_1 & -U_2 \\ \hline -I & 0 & 0 & 0 & 0 \\ \hline 0 & -I & 0 & 0 & 0 \\ \hline 0 & 0 & \widetilde{P}_0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \end{array}\right], \qquad B = \left[\begin{array}{c|c|c|c|c} P_4 & 0 & 0 & 0 & 0 \\ \hline 0 & I & 0 & 0 & 0 \\ \hline 0 & 0 & I & 0 & 0 \\ \hline 0 & 0 & 0 & I & 0 \\ \hline 0 & 0 & 0 & 0 & I \end{array}\right],$$

where the last block row and column has $n - r_0$ rows and columns, respectively. This effectively deflates $n - r_0$ zero eigenvalues at the bottom right corner. After removing the last $n - r_0$ rows and columns (colored red above) we obtain a reduced problem that takes the form

$$A = \left[\begin{array}{c|c|c|c} P_3 & P_2 & P_1 & -U_1 \\ \hline -I & 0 & 0 & 0 \\ \hline 0 & -I & 0 & 0 \\ \hline 0 & 0 & \widetilde{P}_0 & 0 \end{array}\right], \quad B = \left[\begin{array}{c|c|c|c} P_4 & 0 & 0 & 0 \\ \hline 0 & I & 0 & 0 \\ \hline 0 & 0 & I & 0 \\ \hline 0 & 0 & 0 & I \end{array}\right], \tag{A.4}$$

where each black block has size $n \times n$, the red block has size $r_0 \times r_0$, and each blue block has size $n \times r_0$ or $r_0 \times n$. This completes our description of the first step of the deflation procedure.

A.3. Deflation of infinite eigenvalues induced by P_d. The input to this step takes the form (A.4) for $d = 4$ and the aim is to deflate infinite eigenvalues induced by a rank deficiency in P_d. If P_d has full rank then there is nothing to do. Therefore, assume that P_d has rank $r_d < n$ and compress the columns of P_d towards the right by post-multiplication with a carefully crafted unitary matrix $\widehat{U}$, i.e., choose a unitary matrix $\widehat{U}$ such that

$$P_d \widehat{U} = \begin{bmatrix} 0 & \widehat{P}_d \end{bmatrix},$$

where $\widehat{P}_d$ has r_d columns and full rank. Transform (A, B) by post-multiplying the first block column with $\widehat{U}$. This changes the $(2,1)$ block of A from $-I$ to $-\widehat{U}$. Restore the structure of A by pre-multiplying the second block row with $\widehat{U}^*$. This simultaneously changes the $(2,2)$ block of B from I to $\widehat{U}^*$. Continue restoring identities until finally the $(d, d-1)$ block of A changes from $\widehat{P}_0$ to $\widehat{P}_0\widehat{U}$ and the process ends. The transformed pair (A, B) now takes the form

$$A = \left[\begin{array}{cc|cc|cc|c} \widehat{P}_{3,1} & \widehat{P}_{3,3} & \widehat{P}_{2,1} & \widehat{P}_{2,3} & \widehat{P}_{1,1} & \widehat{P}_{1,3} & -U_{1,1} \\ \widehat{P}_{3,2} & \widehat{P}_{3,4} & \widehat{P}_{2,2} & \widehat{P}_{2,4} & \widehat{P}_{1,2} & \widehat{P}_{1,4} & -U_{1,2} \\ \hline -I & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -I & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & -I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -I & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & \widehat{P}_{0,1} & \widehat{P}_{0,2} & 0 \end{array}\right], \quad B = \left[\begin{array}{cc|cc|cc|c} 0 & \widehat{P}_{4,1} & 0 & 0 & 0 & 0 & 0 \\ 0 & \widehat{P}_{4,2} & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & I & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & I \end{array}\right].$$

With two exceptions, the odd-numbered block rows and columns have $n - r_d$ rows and columns and the even-numbered block rows and columns have r_d rows and columns. The two exceptions are the last block row and column, each of which have r_0 rows and r_0 columns.

Next take the top left 3×1 block submatrix of A and transform it to upper triangular form by means of a QR factorization,

$$\begin{bmatrix} \widehat{P}_{3,1} \\ \widehat{P}_{3,2} \\ -I \end{bmatrix} = Q \begin{bmatrix} R \\ 0 \\ 0 \end{bmatrix}. \tag{A.5}$$

Then transform the first three block rows of A and B by a pre-multiplication with

Q^*. This changes the structure of (A, B) to

$$A = \left[\begin{array}{cc|cc|cc|cc} R & X & X & X & X & X & X & X \\ 0 & X & X & X & X & X & X & X \\ \hline 0 & X & X & X & X & X & X & X \\ 0 & -I & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & -I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -I & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & X & X & 0 & 0 \\ 0 & 0 & 0 & 0 & X & X & 0 & 0 \end{array}\right], \qquad B = \left[\begin{array}{cc|cc|cc|cc} 0 & X & X & 0 & 0 & 0 & 0 & 0 \\ 0 & X & X & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & X & X & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & I & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & I & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & I \end{array}\right],$$

where X denotes a dense matrix of appropriate size. This deflates $n - r_d$ infinite eigenvalues at the top left corner. What remains after removing the first $n - r_d$ rows and columns is a reduced problem with the structure

$$A = \left[\begin{array}{c|c|c|c} \check{P}_3 & \check{P}_2 & \check{P}_1 & X \\ \hline -I & 0 & 0 & 0 \\ \hline 0 & -I & 0 & 0 \\ \hline 0 & 0 & \check{P}_0 & 0 \end{array}\right], \qquad B = \left[\begin{array}{c|c|c|c} \check{P}_4 & 0 & 0 & 0 \\ \hline 0 & I & 0 & 0 \\ \hline 0 & 0 & I & 0 \\ \hline 0 & 0 & 0 & I \end{array}\right]. \tag{A.6}$$

The sizes of the blocks in (A.6) are the same as the blocks in (A.4), except for the colored blocks which have shrunk. The red blocks have size $r_d \times r_d$ and one dimension of each blue block has reduced from n to r_d. It is straightforward to verify that the block sizes agree with the output specification (A.2). This completes our description of the second step of the deflation procedure.

A.4. The extreme case $d = 2$. The extreme case $d = 2$ requires a little bit of additional care. The first step (Section A.2) of the deflation procedure can be applied the same as before. The output from the first step takes for $d = 2$ the form

$$A = \left[\begin{array}{c|c} P_1 & -U_1 \\ \hline \widehat{P}_0 & 0 \end{array}\right], \quad B = \left[\begin{array}{c|c} P_2 & 0 \\ \hline 0 & I \end{array}\right],$$

where the last block row and column have r_0 rows and columns. What concerns us is what transpires in the second step after having compressed the columns of P_2. According to Section A.3 we should next compute a QR factorization of the top left 3×1 block submatrix of A. But for $d = 2$ the third block in that submatrix is not $-I$ as in (A.5) but instead a dense block $\widehat{P}_{0,1}$ of size $r_0 \times (n - r_2)$. For $d > 2$ that block would have been equal to $-I_{n-r_d}$. What is important for the effect of the QR factorization on the sparsity structure is the number of leading nonzero rows in that 3×1 block submatrix. Special care for $d = 2$ is therefore needed if $r_0 > n - r_2$. In that case the number of leading nonzero rows is greater than for the case $d > 2$ and thus a direct application of the second step would lead to more fill. To avoid this problem we first reduce the dense (tall-and-skinny) $\widehat{P}_{0,1}$ block to upper triangular form, which in itself has no effect on the generic sparsity structure. But it crucially reduces the number of nonzero leading rows in $\widetilde{P}_{0,1}$ from r_0 to the desired $n - r_2$ and thereby overcomes the problem.

A.5. Algorithm. Algorithm 5 summarizes the discussion above by formalizing the deflation procedure.

Algorithm 5: Deflation procedure. Deflates zero and infinite eigenvalues induced by rank deficiencies in P_0 and/or P_d.

Construct the matrix pair (A, B) as in (A.3);
Compute the ranks r_0 and r_d of P_0 and P_d, respectively;
Construct a unitary matrix U_0 such that $U_0^* P_0 = \begin{bmatrix} \widetilde{P}_0 \\ 0 \end{bmatrix}$ and $\widetilde{P}_0$ has full rank;
Construct a unitary matrix U_d such that $P_d U_d = \begin{bmatrix} 0 & \widetilde{P}_d \end{bmatrix}$ and $\widetilde{P}_d$ has full rank;
Overwrite the $(d, d-1)$ block of A with $U_0^* P_0$;
Overwrite the top right corner of A with U_0;
Deflate $n - r_0$ zero eigenvalues from the bottom right corner and remove the last $n - r_0$ rows and columns from A and B;
Overwrite the $(1, 1)$ block of B with $P_d U_d$;
for $k \leftarrow 1, 2, \ldots d-1$ **do**
    Overwrite the $(1, k)$ block of A with $P_{d-k} U_d$;
Overwrite the $(d, d-1)$ block of A with $(U_0^* P_0) U_d$;
if $d = 2$ *and* $r_0 > n - r_2$ **then**
    Let X denote the submatrix $A(n+1\colon n+r_0, 1\colon n-r_2)$;
    Compute a QR factorization $X = QR$;
    Update A by pre-multiplying rows $n+1\colon n+r_0$ by Q^*;
    Update A by post-multiplying columns $n+1\colon n+r_0$ by Q;
Let X denote the submatrix $A(1\colon 2n-r_d, 1\colon n-r_d)$;
Compute a QR factorization $X = QR$;
Update A and B by pre-multiplying rows $1\colon 2n-r_d$ by Q^*;
Deflate $n - r_d$ infinite eigenvalues at the top left corner and remove the first $n - r_d$ rows and columns from A and B;

[1] E. N. Antoniou and S. Vologiannidis, *A new family of companion forms of polynomial matrices*, Electron. J. Linear Algebra, 11 (2004), pp. 78–87.

[2] K. Dackland and B. Kågström, *Blocked algorithms and software for reduction of a regular matrix pair to generalized Schur form*, ACM Trans. Math. Software, 25 (1999), pp. 425–545.

[3] F. De Terán and F. Tisseur, *Backward error and conditioning of Fiedler companion linearizations*. In preparation.

[4] I. Gohberg, P. Lancaster, and L. Rodman, *Matrix Polynomials*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009. Unabridged republication of book first published by Academic Press in 1982.

[5] S. Hammarling, C. J. Munro, and F. Tisseur, *An algorithm for the complete solution of quadratic eigenvalue problems*, ACM Trans. Math. Software, 39 (2013), pp. 18:1–18:19.

[6] N. J. Higham, R.-C. Li, and F. Tisseur, *Backward error of polynomial eigenproblems solved by linearization*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 1218–1241.

[7] N. J. Higham, D. S. Mackey, and F. Tisseur, *The conditioning of linearizations of matrix polynomials*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 1005–1028.

[8] B. Kågström and D. Kressner, *Multishift variants of the QZ algorithm with agressive early deflation*, SIAM J. Matrix Anal. Appl., 29 (2006), pp. 199–227.

[9] B. Kågström, D. Kressner, E. S. Quintana-Orti, and G. Quintana-Orti, *Blocked algorithms for the reduction to Hessenberg-triangular form revisited*, BIT, 48 (2008), pp. 563–584.

[10] L. Karlsson and B. Kågström, *Efficient reduction from block Hessenberg form to Hessenberg form using shared memory*, in Applied Parallel and Scientific Computing, K. Jónasson, ed., vol. 7134 of LNCS, Springer Berlin Heidelberg, 2012, pp. 258–268.

[11] B. Lang, *Parallel reduction of banded matrices to bidiagonal form*, Parallel Comput., 22 (1996), pp. 1–18.
[12] ———, *Using level 3 BLAS in rotation-based algorithms*, SIAM J. Sci. Comput., 19 (1998), pp. 626–634.
[13] D. S. Mackey, *The continuing influence of Fiedler's work on companion matrices*, Linear Algebra Appl., 439 (2013), pp. 810–817.
[14] D. S. Mackey, N. Mackey, C. Mehl, and V. Mehrmann, *Vector spaces of linearizations for matrix polynomials*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 971–1004.
[15] D. S. Watkins, *Performance of the QZ algorithm in the presence of infinite eigenvalues*, SIAM J. Matrix Anal. Appl., 22 (2000), pp. 364–375.