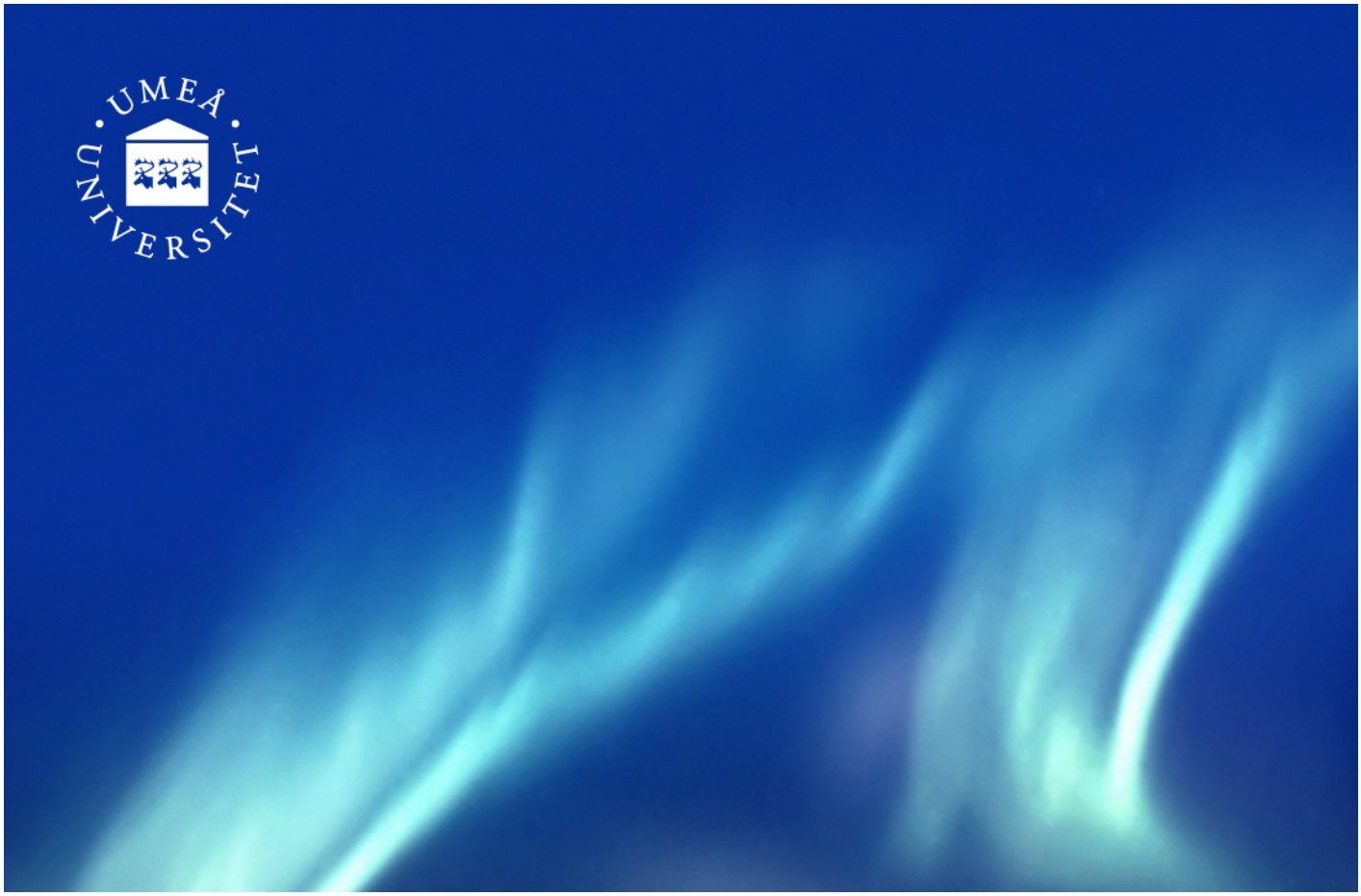

Parallel two-stage reduction to Hessenberg form using shared memory

Lars Karlsson and Bo Kågström

UMINF 10.14

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
Sweden

Parallel two-stage reduction to Hessenberg form using shared memory

L. Karlsson[a], B. Kågström[a]

[a] *Department of Computing Science and HPC2N, Umeå University, SE-901 87, Umeå, Sweden*

Abstract

We consider parallel reduction of a real matrix to Hessenberg form using orthogonal transformations. Standard Hessenberg reduction algorithms reduce the columns of the matrix from left to right in either a blocked or unblocked fashion. However, the standard blocked variant performs 20% of the computations as large matrix–vector multiplications. We show that a two-stage approach consisting of an intermediate reduction to r-Hessenberg form speeds up the reduction by avoiding matrix–vector multiplications. We describe and evaluate a new high-performance implementation of the two-stage approach that attains significant speedups over the one-stage approach on a dual quad-core machine. The key components are a dynamically scheduled implementation of the first stage and a blocked adaptively load-balanced implementation of the second stage.

Keywords: Hessenberg reduction, blocked algorithm, parallel computing, dynamic scheduling, high performance, multicore, memory hierarchies

1. Introduction

The Hessenberg decomposition $A = QHQ^T$ of a square matrix $A \in \mathbb{R}^{n\times n}$, where $H \in \mathbb{R}^{n\times n}$ is an upper Hessenberg matrix and $Q \in \mathbb{R}^{n\times n}$ is orthogonal, has many applications in numerical linear algebra [1]. Our focus is on large-scale applications that require blocked and parallel algorithms. For example, the Hessenberg QR algorithm, which computes a Schur form, takes as input a matrix in Hessenberg form [2]. Other examples include solvers for dense Sylvester-type matrix equations, which typically simplify the coefficient matrices to Schur and/or Hessenberg form [3, 4]. A third application is the solution of linear systems $(A - \sigma I)x = b$ for many shifts σ. By first reducing A to Hessenberg form, each system can be solved with only $\mathcal{O}(n^2)$ flops.

The aim of this paper is to describe and evaluate a new efficient Hessenberg reduction algorithm for shared-memory machines. The implementation assumes nearly uniform memory access, but the underlying algorithmic ideas are also applicable to machines with a strong NUMA component or distributed memory. The proposed algorithm consists of two stages, whereas the conventional approach goes directly to Hessenberg form. The first stage reduces a dense matrix to r-Hessenberg form, i.e., such that $A(i, j) = 0$ if $i > j + r$.

Email addresses: larsk@cs.umu.se (L. Karlsson), bokg@cs.umu.se (B. Kågström)

The second stage completes the Hessenberg reduction with a bulge-chasing procedure that annihilates the $r-1$ unwanted nonzero subdiagonals from the r-Hessenberg matrix.

The idea to do reduction in two stages is not new. Two-stage algorithms are effective for both symmetric band reduction [5] and Hessenberg-triangular reduction [6, 7].

The rest of the paper is organized as follows. We recall the conventional unblocked and blocked Hessenberg reduction algorithms in Section 2. We discuss some pros and cons of the two-stage approach in Section 3. We describe a new implementation of the first stage in Section 4. We outline the authors' recently published implementation of the second stage [8] in Section 5. We evaluate the performance of the two stages, both in isolation and combined, in Section 6. Finally, we summarize our findings and briefly review some related work in Section 7.

2. One-stage Hessenberg reduction

In both the blocked and the unblocked standard algorithms, the matrix is reduced from left to right using one Householder reflection per column. We recall the unblocked variant in Section 2.1 and the blocked variant in Section 2.2. They form the basis of several high-performance implementations of the Hessenberg reduction, e.g., see [9, 10].

2.1. Unblocked one-stage Hessenberg reduction

Algorithm 1 is a reformulation of the standard unblocked Hessenberg reduction algorithm. The j-loop iterates over the columns of the matrix and reduces them from left to right. A Householder reflection Q_j of order $n-j$ is constructed on line 2. A similarity transformation derived from Q_j is immediately applied to the matrix on lines 3–4. The input matrix A is reduced to Hessenberg form after $n-2$ iterations.

Algorithm 1 Unblocked one-stage Hessenberg reduction. Given a dense matrix $A \in \mathbb{R}^{n\times n}$, this algorithm overwrites A with $H = Q^T A Q$, where $H \in \mathbb{R}^{n\times n}$ is in upper Hessenberg form and $Q \in \mathbb{R}^{n\times n}$ is orthogonal.

1: **for** $j = 1{:}n-2$ **do**
2: Reduce $A(j+1{:}n, j)$ with a Householder reflection $Q_j = I - \tau v v^T$
3: $A(j+1{:}n, j{:}n) = Q_j^T \cdot A(j+1{:}n, j{:}n)$
4: $A(1{:}n, j+1{:}n) = A(1{:}n, j+1{:}n) \cdot Q_j$
5: **end for**

The flop count of Algorithm 1 is $\frac{10}{3}n^3$ plus lower order terms. Due to the special structure of a Householder reflection, its multiplication with a vector of length n requires only $4n$ flops compared to $2n^2$ flops for a dense matrix–vector multiplication. Approximately 60% of the flops are associated with updates on the right-hand side (line 4), and the remaining 40% are associated with updates on the left-hand side (line 3).

The performance of Algorithm 1 is limited on large matrices due to its exclusive use of matrix–vector multiplications. However, it is competitive for small matrices since it uses the least amount of flops of all the Hessenberg reduction algorithms that we discuss in this paper.

2.2. Blocked one-stage Hessenberg reduction

Most of the performance issues of Algorithm 1 are averted by applying the compact WY representation [11] as described in [9, 12]. Algorithm 2 states the resulting blocked algorithm. A benefit of using the compact WY representation is that 80% of the flops get associated with matrix–matrix multiplications. However, the remaining 20% flops remain as matrix–vector multiplications.

Algorithm 2 Blocked one-stage Hessenberg reduction. Given a dense matrix $A \in \mathbb{R}^{n\times n}$, this algorithm overwrites A with $H = Q^T AQ$, where $H \in \mathbb{R}^{n\times n}$ is in upper Hessenberg form and $Q \in \mathbb{R}^{n\times n}$ is orthogonal. The block size is b.

1: **for** $j_1 = 1{:}b{:}n-2$ **do**
2: $\quad\hat{b} = \min\{b, n-j_1-1\}$
3: $\quad\mathbf{i}_1 = 1{:}j_1; \quad \mathbf{i}_2 = j_1+1{:}n; \quad \mathbf{i}_3 = j_1+\hat{b}{:}n$
4: $\quad$Set $Y \in \mathbb{R}^{n\times\hat{b}}$, $V \in \mathbb{R}^{n\times\hat{b}}$, and $T \in \mathbb{R}^{\hat{b}\times\hat{b}}$ to zero.
5: $\quad$**for** $j = j_1{:}j_1+\hat{b}-1$ **do**
6: $\quad\quad\mathbf{i}_4 = j+1{:}n; \quad \mathbf{i}_5 = 1{:}j-j_1$
7: $\quad\quad A(\mathbf{i}_2, j) = A(\mathbf{i}_2, j) - Y(\mathbf{i}_2, \mathbf{i}_5)\cdot V(j, \mathbf{i}_5)^T$
8: $\quad\quad A(\mathbf{i}_2, j) = (I - V(\mathbf{i}_2, \mathbf{i}_5)\cdot T(\mathbf{i}_5, \mathbf{i}_5)\cdot V(\mathbf{i}_2, \mathbf{i}_5)^T)^T \cdot A(\mathbf{i}_2, j)$
9: $\quad\quad$Reduce $A(\mathbf{i}_4, j)$ with a Householder reflection $I - \tau vv^T$
10: $\quad\quad V(\mathbf{i}_4, j-j_1+1) = v; \quad T(j-j_1+1, j-j_1+1) = \tau$
11: $\quad\quad T(\mathbf{i}_5, j-j_1+1) = \tau \cdot V(\mathbf{i}_4, \mathbf{i}_5)^T \cdot v$
12: $\quad\quad Y(\mathbf{i}_2, j-j_1+1) = \tau \cdot A(\mathbf{i}_2, \mathbf{i}_4)\cdot v - Y(\mathbf{i}_2, \mathbf{i}_5)\cdot T(\mathbf{i}_5, j-j_1+1)$
13: $\quad\quad T(\mathbf{i}_5, j-j_1+1) = -T(\mathbf{i}_5, \mathbf{i}_5)\cdot T(\mathbf{i}_5, j-j_1+1)$
14: $\quad$**end for**
15: $\quad Y(\mathbf{i}_1, :) = A(\mathbf{i}_1, \mathbf{i}_2)\cdot V(\mathbf{i}_2, :)\cdot T$
16: $\quad A(\mathbf{i}_1, \mathbf{i}_2) = A(\mathbf{i}_1, \mathbf{i}_2) - Y(\mathbf{i}_1, :)\cdot V(\mathbf{i}_2, :)^T$
17: $\quad A(\mathbf{i}_2, \mathbf{i}_3) = A(\mathbf{i}_2, \mathbf{i}_3) - Y(\mathbf{i}_2, :)\cdot V(\mathbf{i}_3, :)^T$
18: $\quad A(\mathbf{i}_2, \mathbf{i}_3) = (I - V(\mathbf{i}_2, :)\cdot T\cdot V(\mathbf{i}_2, :)^T)^T \cdot A(\mathbf{i}_2, \mathbf{i}_3)$
19: **end for**

Algorithm 2 applies a sequence of orthogonal similarity transformations

$$A \leftarrow (I - VTV^T)^T A(I - VTV^T) \tag{1}$$

that eventually produces a matrix in upper Hessenberg form. The purpose of the inner loop (lines 5–14) is to construct a compact WY representation $I - VTV^T$ such that (1) reduces columns $j_1{:}j_1+\hat{b}-1$. The inner loop also computes the lower part of the matrix $Y = AVT$. The statements 15–18 in the outer loop completes the computation of Y and also updates the matrix A via

$$A \leftarrow (I - VTV^T)^T (A - YV^T). \tag{2}$$

The formulas that are used in the inner loop to compute the upper triangular matrix T are derived from a special case of equation (3), which expresses the product of two compact WY representations as a new one:

$$(I - V_1T_1V_1^T)(I - V_2T_2V_2^T) = I - \underbrace{\begin{bmatrix}V_1 & V_2\end{bmatrix}}_{V} \underbrace{\begin{bmatrix}T_1 & -T_1V_1^TV_2T_2 \\ 0 & T_2\end{bmatrix}}_{T} \underbrace{\begin{bmatrix}V_1 & V_2\end{bmatrix}^T}_{V^T}. \tag{3}$$

The lower part of the matrix Y is computed using the formula

$$\begin{aligned} Y &= \begin{bmatrix} Y_1 & Y_2 \end{bmatrix} = AVT \\ &= A \begin{bmatrix} V_1 & V_2 \end{bmatrix} \begin{bmatrix} T_1 & -T_1 V_1^T V_2 T_2 \\ 0 & T_2 \end{bmatrix} \\ &= \begin{bmatrix} AV_1T_1 & AV_2T_2 - AV_1T_1V_1^TV_2T_2 \end{bmatrix} \\ &= \begin{bmatrix} Y_1 & AV_2T_2 - Y_1V_1^TV_2T_2 \end{bmatrix}. \end{aligned} \tag{4}$$

Since the expression $V_1^T V_2 T_2$ appears in both (3) and (4), it is computed only once by Algorithm 2.

The flop count is similar to that of Algorithm 1 if we neglect lower order terms. However, the overhead of the blocked algorithm grows proportionally to b^3. The leading term of the flop count can be broken down into five primary contributions as follows. Line 12: $\frac{2}{3}n^3$ (20%), line 15: $\frac{1}{3}n^3$ (10%), line 16: $\frac{1}{3}n^3$ (10%), line 17: $\frac{2}{3}n^3$ (20%), and line 18: $\frac{4}{3}n^3$ (40%).

3. Two-stage Hessenberg reduction

Recall that the two-stage approach first reduces the matrix to r-Hessenberg form and then to upper Hessenberg form. The free parameter r can be chosen to maximize performance. As is explained in the coming sections, the performance of the first stage increases with r, whereas the performance of the second stage decreases with r. This trade-off leads to the conclusion that a modest r-value gives the best performance when the two stages are combined.

The one-stage approach requires $\frac{10}{3}n^3$ flops to compute H and an additional $\frac{4}{3}n^3$ flops to accumulate Q. The two-stage approach requires $\frac{10}{3}n^3$ flops for the first stage alone, and $2n^3$ flops for the second stage. Accumulation of the reflections from the first stage requires $\frac{4}{3}n^3$ flops, and the reflections from the second stage add $2n^3$ flops on top of that. In summary, the two-stage approach consumes 60% more flops than the one-stage approach when computing H, and it consumes 86% more flops when Q is also explicitly computed.

4. Stage 1: reduction from dense to r-Hessenberg form

It is straightforward to modify Algorithm 2 to produce an r-Hessenberg form using two levels of blocking. The inner block size is determined by the number of subdiagonals r. The outer block size b, however, does not influence the second stage and can hence be tuned independently. Two levels of blocking are necessary since r is relatively small.

4.1. Algorithm

Algorithm 3 generalizes Algorithm 2 to produce an r-Hessenberg form. In each iteration of the inner loop, Algorithm 3 reduces r columns by a recursive QR factorization [13, 14] (line 10). The matrix–vector multiplications Av in Algorithm 1 correspond to matrix–matrix multiplications AV, where V has r columns, in Algorithm 2.

A sequential implementation of Algorithm 3 that is linked to a multithreaded BLAS library would have suboptimal performance since many operations are small and hence

Algorithm 3 Blocked reduction to r-Hessenberg form. Given a dense matrix $A \in \mathbb{R}^{n\times n}$, this algorithm overwrites A with $H = Q^T AQ$, where $H \in \mathbb{R}^{n\times n}$ is in upper r-Hessenberg form and $Q \in \mathbb{R}^{n\times n}$ is orthogonal. The outer block size is b.

1: **for** $j_1 = 1{:}b{:}n-r-1$ **do**
2: $\quad \hat{b} = \min\{b, n-r-j_1\}$
3: $\quad$ Set $Y \in \mathbb{R}^{n\times\hat{b}}$, $V \in \mathbb{R}^{n\times\hat{b}}$, and $T \in \mathbb{R}^{\hat{b}\times\hat{b}}$ to zero
4: $\quad \mathbf{i}_1 = 1{:}j_1+r-1; \quad \mathbf{i}_2 = j_1+r{:}n; \quad \mathbf{i}_3 = j_1+\hat{b}{:}n$
$\quad$ *% Task $\mathcal{T}_1$ (lines 5–15)*
5: $\quad$ **for** $j_2 = j_1{:}r{:}j_1+\hat{b}-1$ **do**
6: $\quad\quad \hat{r} = \min\{r, j_1+\hat{b}-j_2\}$
7: $\quad\quad \mathbf{i}_4 = j_2+r{:}n; \quad \mathbf{i}_5 = 1{:}j_2-j_1; \quad \mathbf{i}_6 = j_2-j_1+1{:}j_2+\hat{r}-j_1; \quad \mathbf{i}_7 = j_2{:}j_2+\hat{r}-1$
$\quad\quad$ *% Task $\mathcal{T}_{1.1}$ (lines 8–12)*
8: $\quad\quad A(\mathbf{i}_2,\mathbf{i}_7) = A(\mathbf{i}_2,\mathbf{i}_7) - Y(\mathbf{i}_2,\mathbf{i}_5)\cdot V(\mathbf{i}_7,\mathbf{i}_5)^T$
9: $\quad\quad A(\mathbf{i}_2,\mathbf{i}_7) = (I - V(\mathbf{i}_2,\mathbf{i}_5)\cdot T(\mathbf{i}_5,\mathbf{i}_5)\cdot V(\mathbf{i}_2,\mathbf{i}_5)^T)^T \cdot A(\mathbf{i}_2,\mathbf{i}_7)$
10: $\quad\quad$ Reduce $A(\mathbf{i}_4,\mathbf{i}_7)$ with the QR-factorization $A(\mathbf{i}_4,\mathbf{i}_7) = (I - \hat{V}\hat{T}\hat{V}^T)R$
11: $\quad\quad V(\mathbf{i}_4,\mathbf{i}_6) = \hat{V}; \quad T(\mathbf{i}_6,\mathbf{i}_6) = \hat{T}$
12: $\quad\quad T(\mathbf{i}_5,\mathbf{i}_6) = V(\mathbf{i}_4,\mathbf{i}_5)^T \cdot V(\mathbf{i}_4,\mathbf{i}_6)\cdot T(\mathbf{i}_6,\mathbf{i}_6)$
$\quad\quad$ *% Task $\mathcal{T}_{1.2}$ (line 13)*
13: $\quad\quad Y(\mathbf{i}_2,\mathbf{i}_6) = A(\mathbf{i}_2,\mathbf{i}_4)\cdot V(\mathbf{i}_4,\mathbf{i}_6)\cdot T(\mathbf{i}_6,\mathbf{i}_6) - Y(\mathbf{i}_2,\mathbf{i}_5)\cdot T(\mathbf{i}_5,\mathbf{i}_6)$
$\quad\quad$ *% Task $\mathcal{T}_{1.3}$ (line 14)*
14: $\quad\quad T(\mathbf{i}_5,\mathbf{i}_6) = -T(\mathbf{i}_5,\mathbf{i}_5)\cdot T(\mathbf{i}_5,\mathbf{i}_6)$
15: $\quad$ **end for**
$\quad$ *% Task $\mathcal{T}_2$ (line 16)*
16: $\quad Y(\mathbf{i}_1,:) = A(\mathbf{i}_1,\mathbf{i}_2)\cdot V(\mathbf{i}_2,:)\cdot T$
$\quad$ *% Task $\mathcal{T}_3$ (line 17)*
17: $\quad A(\mathbf{i}_1,\mathbf{i}_2) = A(\mathbf{i}_1,\mathbf{i}_2) - Y(\mathbf{i}_1,:)\cdot V(\mathbf{i}_2,:)^T$
$\quad$ *% Task $\mathcal{T}_4$ (lines 18–19)*
18: $\quad A(\mathbf{i}_2,\mathbf{i}_3) = A(\mathbf{i}_2,\mathbf{i}_3) - Y(\mathbf{i}_2,:)\cdot V(\mathbf{i}_3,:)^T$
19: $\quad A(\mathbf{i}_2,\mathbf{i}_3) = (I - V(\mathbf{i}_2,:)\cdot T\cdot V(\mathbf{i}_2,:)^T)^T \cdot A(\mathbf{i}_2,\mathbf{i}_3)$
20: **end for**

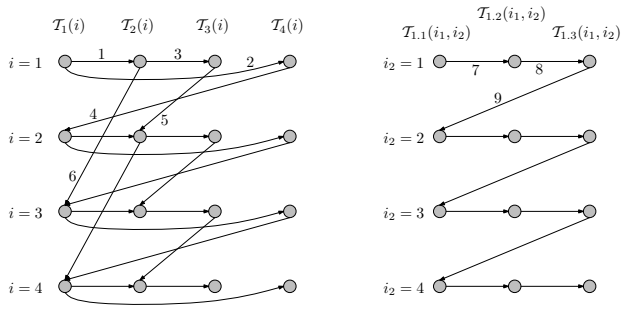

Figure 1: Partial task graphs for the outer (*left*) and inner (*right*) loops. The nodes represent task instances, and the edges represent precedence constraints. Only a few task instances, enough to display the pattern of constraints, are shown. The index i_1 is constant in the graph on the right.

do not parallelize well. One consequence is a substantial overhead due to idle processors/cores.

Our implementation of Algorithm 3 relies on dynamic scheduling of coarse-grained tasks that call sequential BLAS. The scheduler tries to overlap the sequential bottlenecks, e.g., the QR factorization, with delayed updates, e.g., lines 16–17.

The distribution of flops in Algorithm 3 is essentially the same as in Algorithm 2. In particular, the delayable updates on lines 16–17 account for 20% of the flops.

4.2. Tasks, atoms, and constraints

The comments in Algorithm 3 define four tasks within the outer loop, namely, tasks $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$, and $\mathcal{T}_4$. There is one instance of each task per iteration of the outer loop. We let $\mathcal{T}_a(i)$, where $a \in \{1, 2, 3, 4\}$, denote the i-th instance of task $\mathcal{T}_a$.

Precedence constraints are expressed using "$\prec$". For example, $\mathcal{T}_a(i) \prec \mathcal{T}_b(j)$ means that the instance $\mathcal{T}_a(i)$ must complete before the instance $\mathcal{T}_b(j)$ is allowed to start. Our implementation imposes the following constraints, which are also illustrated in Figure 1 (*left*).

1. $\mathcal{T}_1(i) \prec \mathcal{T}_2(i)$. Instance $\mathcal{T}_1(i)$ must complete the computation of V and T before instance $\mathcal{T}_2(i)$ can use them.
2. $\mathcal{T}_1(i) \prec \mathcal{T}_4(i)$. Instance $\mathcal{T}_1(i)$ must complete the computation of Y, V, and T before instance $\mathcal{T}_4(i)$ can use them.
3. $\mathcal{T}_2(i) \prec \mathcal{T}_3(i)$. Instance $\mathcal{T}_2(i)$ must complete the computation of Y before instance $\mathcal{T}_3(i)$ can use it.
4. $\mathcal{T}_4(i) \prec \mathcal{T}_1(i+1)$. Instance $\mathcal{T}_4(i)$ must complete the updates of the trailing matrix before instance $\mathcal{T}_1(i+1)$ can use it.
5. $\mathcal{T}_3(i) \prec \mathcal{T}_2(i+1)$. Instance $\mathcal{T}_3(i)$ must complete its use of the buffer for Y before instance $\mathcal{T}_2(i+1)$ overwrites it.
6. $\mathcal{T}_2(i) \prec \mathcal{T}_1(i+2)$. Instance $\mathcal{T}_2(i)$ must complete its use of one of the two buffers for T before instance $\mathcal{T}_1(i+2)$ overwrites it.

Constraints 1–4 are direct consequences of the data flow of the algorithm. Constraint 5 is a result of the use of a single buffer for Y. Constraint 6 is a result of the use of double

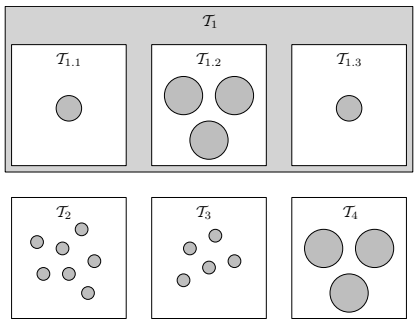

Figure 2: Decomposition of tasks into atoms. The granularity (fine to large) of the atoms are illustrated by discs of different size. Since the tasks $\mathcal{T}_{1.1}$ and $\mathcal{T}_{1.3}$ are sequential, each of them contains a single disc only.

buffers for T. Instance $\mathcal{T}_1(i+1)$ can be overlapped with the updates represented by the two instances $\mathcal{T}_2(i)$ and $\mathcal{T}_3(i)$.

The inner loop, i.e., task $\mathcal{T}_1$, is further partitioned into the three tasks $\mathcal{T}_{1.1}$, $\mathcal{T}_{1.2}$, and $\mathcal{T}_{1.3}$. The instances of these tasks are decsribed by two parameters. Specifically, the notation $\mathcal{T}_b(i_1, i_2)$, where $b \in \{1.1, 1.2, 1.3\}$, refers to the i_2-th instance of task $\mathcal{T}_b$ within the instance $\mathcal{T}_1(i_1)$.

The nested tasks are constrained by the following constraints, which are also illustrated in Figure 1 (*right*).

7. $\mathcal{T}_{1.1}(i_1, i_2) \prec \mathcal{T}_{1.2}(i_1, i_2)$. Instance $\mathcal{T}_{1.1}(i_1, i_2)$ must complete the reduction step before instance $\mathcal{T}_{1.2}(i_1, i_2)$ can use the new reflections.
8. $\mathcal{T}_{1.2}(i_1, i_2) \prec \mathcal{T}_{1.3}(i_1, i_2)$. Instance $\mathcal{T}_{1.2}(i_1, i_2)$ must complete its use of the buffer for T before instance $\mathcal{T}_{1.3}(i_1, i_2)$ overwrites it.
9. $\mathcal{T}_{1.3}(i_1, i_2) \prec \mathcal{T}_{1.1}(i_1, i_2 + 1)$. Instance $\mathcal{T}_{1.3}(i_1, i_2)$ must complete the computation of T before instance $\mathcal{T}_{1.1}(i_1, i_2 + 1)$ can use it.

Constraints 7–8 restrict the instances within one iteration, and constraint 9 forces one iteration to end before the next iteration begins. In other words, the instances in the inner loop must be executed in sequence.

The instances are too coarse to be scheduled directly to the threads. Each instance is therefore partitioned into one or more independent *atoms*, which are the units of sequential computation that the scheduler maps to threads. The constraints are enforced at the task level, while the computations are scheduled at the atom level.

The granularity of the atoms is an important consideration. Fine-grained atoms typically lead to tight schedules that have little idle time overhead. However, fine-grained atoms also lead to large amounts of scheduler overhead and slow kernels. Coarse-grained atoms, on the other hand, lead to less scheduler overhead and faster kernels, but can also lead to idle time overhead.

We use a mix of both fine- and coarse-grained atoms, as illustrated in Figure 2. Instances of the tasks $\mathcal{T}_{1.1}$ and $\mathcal{T}_{1.3}$ are executed sequentially and thus they each correspond to a single atom. Instances of the tasks $\mathcal{T}_{1.2}$ and $\mathcal{T}_4$ are decomposed into coarse-grained atoms, i.e., one atom per thread. Instances of the remaining tasks are decomposed into fine-grained atoms. The coarse-grained tasks $\mathcal{T}_{1.2}$ and $\mathcal{T}_4$ account for 80% of the flops in

Algorithm 3. Thus, the majority of the computations are done by the efficient kernels that are enabled by the coarse-grained atoms.

4.3. Scheduler of stage 1

A suitable scheduler should overlap the sequential reduction step with delayed updates while also avoiding excessive idle time overhead due to the presence of coarse-grained atoms.

Conceptually, the scheduler consists of a centralized priority queue that contains the atoms that are ready for execution. The threads fetch ready atoms from the queue one at a time. The tasks (and consequently their atoms) are prioritized from highest to lowest priority according to the priority list

$$(\mathcal{T}_{1.2},\ \mathcal{T}_4,\ \mathcal{T}_{1.1},\ \mathcal{T}_{1.3},\ \mathcal{T}_2,\ \mathcal{T}_3). \tag{5}$$

Specifically, if an atom belonging to task $\mathcal{T}_4$ is ready at the same time as an atom belonging to task $\mathcal{T}_2$, then the $\mathcal{T}_4$-atom is fetched first.

In reality, the scheduler uses implicit representations for both the priority queue and the task graph, in an attempt to reduce scheduler overhead. In particular, the entire state of the scheduler fits in a few cache lines. Due to the constraints on the tasks, there is at most one ready instance per task at any moment in time. To find the next ready atom, the scheduler examines each ready instance in the order specified by the list (5). The first free atom that is found in a ready instance is fetched, since it has the highest priority.

The scheduler avoids excessive idle time overhead by keeping the threads nearly synchronized in their execution of atoms that have long durations. Note that the two coarse-grained tasks, i.e., $\mathcal{T}_{1.2}$ and $\mathcal{T}_4$, have the highest priority in (5) and that both can not be ready at the same time. Thus, if a thread fetches a coarse-grained atom, then the other threads shortly thereafter fetch the sibling atoms. The lag depends on the duration of a fine-grained atom, which is relatively short by definition.

5. Stage 2: reduction from r-Hessenberg to Hessenberg form

The input to the second stage is an r-Hessenberg matrix $A \in \mathbb{R}^{n\times n}$. The aim is to complete the reduction to Hessenberg form. Algorithms for symmetric band reduction [15, 16, 17] can be adapted to nonsymmetric matrices, e.g., matrices in r-Hessenberg form.

This section covers some aspects of our recently published parallel algorithm [8]. It has a dynamic adaptive load-balancing scheme that partitions the load into coarse tasks and uses time measurements as feedback to further improve the load balance. One level of look-ahead enables the algorithm to hide a sequential bottleneck by overlapping it with delayed updates. Below, we focus on the aspects that are related to blocking and refer the reader to [8] for details on the parallelization.

Algorithm 4 reduces a matrix from r-Hessenberg to Hessenberg form using Householder reflections of order r. The columns are reduced from left to right. When column j is reduced, an $r \times r$ bulge with fill-in elements appears below the r-th subdiagonal in the block $A(j+r+1:j+2r, j+1:j+r)$. A sequence of reductions are applied to reduce the first column of each bulge that appears. This results in a chain of partially reduced

Algorithm 4 Given an upper r-Hessenberg matrix $A \in \mathbb{R}^{n\times n}$, this algorithm overwrites A with $H = Q^T AQ$, where $H \in \mathbb{R}^{n\times n}$ is in upper Hessenberg form and $Q \in \mathbb{R}^{n\times n}$ is orthogonal.

```
1:  for j = 1:n − 2 do
2:      k_1 = 1 + ⌊(n−j−2)/r⌋
3:      for k = 0:k_1 − 1 do
4:          ℓ = j + max{0, (k − 1)r + 1}
5:          i_1 = j + kr + 1:min{j + (k + 1)r, n}
6:          i_2 = ℓ:n
7:          i_3 = 1:min{j + (k + 2)r, n}
8:          Reduce A(i_1, ℓ) with a Householder reflection Q_k^j = I − τvv^T
9:          A(i_1, i_2) = (Q_k^j)^T A(i_1, i_2)
10:         A(i_3, i_1) = A(i_3, i_1)Q_k^j
11:     end for
12: end for
```

bulges that are spaced r elements apart along the r-th subdiagonal of A. The reduction of the $(j+1)$-st column can now begin, since the new bulges consume the old ones.

In symmetric band reduction, the bulge-chasing requires only $\mathcal{O}(n^2)$ flops and is therefore relatively cheap compared to the $\mathcal{O}(n^3)$ cost of accumulating the transformation matrix Q [15].

In the nonsymmetric case, however, the nonzero upper triangular part of A increases the flop count from $\mathcal{O}(n^2)$ to $2n^3$. Therefore, the low arithmetic intensity, i.e., the ratio of flops to bytes transferred to/from main memory, of Algorithm 4 makes it too slow to be practical.

A blocked algorithm must delay updates from several consecutive sweeps. A *sweep* consists of the computations in one complete execution of the inner loop of Algorithm 4 (lines 3–11). It is possible to obtain all of the reflections associated with q consecutive sweeps starting at column j_1, i.e., the reflections $Q_\star^j$ for $j = j_1 : j_1 + q - 1$, by updating entries only inside a band with bandwidth $\mathcal{O}(qr)$. Hence, the reflections are obtained using no more than $\mathcal{O}(q^2rn)$ flops, which is negligible compared to the $\mathcal{O}(qn^2)$ flops that are associated with the q sweeps.

Figure 3 illustrates which entries are necessary to update in order to construct the reflections $Q_\star^1$ associated with the first sweep. The bulges must be generated by applying some of the updates from the right-hand side to r rows (*regions 2–9*). Moreover, the first column of each bulge must be reduced from the left-hand side (*dark*). To also obtain the reflections from a second sweep, updates from the right-hand side must be applied to an additional r rows and the updates from the left-hand side must be applied to one additional column.

The remaining updates can be applied first from the right-hand side and then from the left-hand side. The aim is to rearrange the remaining updates from either side so that the arithmetic intensity is increased by a factor close to q. Consider the problem of applying the reflections $Q_\star^j$ for $j = j_1 : j_1 + q - 1$ from the left-hand side to a single block of c columns. Assume for simplicity that there are k_1 reflections in each of the sweeps. If the cache is initially empty and emptied explicitly after the updates, then at least

$$qk_1r + 2c(k_1r + q - 1) \tag{6}$$

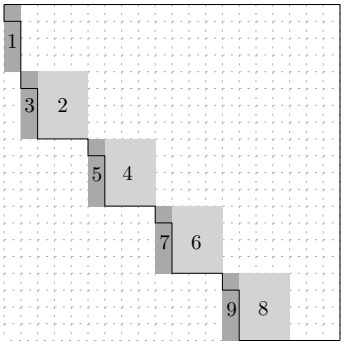

Figure 3: Entries updated from the left-hand side (*dark*) and the right-hand side (*light* + *dark*, excluding region 1) during the construction of $Q_\star^1$ for $n = 20$, $r = 4$. The numbers 1–9 show the order in which the updates are applied. The solid outline encloses the nonzeros that remain after the updates have been applied.

entries must be read from or written to main memory. The first term comes from reading the Householder vectors, and the second term comes from reading and writing the block column.

If there is no cache reuse across reflections, then at least

$$qk_1r + 2qk_1rc \tag{7}$$

entries must be read from or written to main memory. The first term comes from reading the Householder vectors, and the second term comes from reading and writing one block of rc entries per reflection.

Since the number of flops is the same for the best and worst cases, the ratio of the arithmetic intensity in the best case to that in the worst case approaches

$$q\left(\frac{1+2c}{q+2c}\right) \tag{8}$$

as k_1 tends to infinity. In particular, the improvement factor is close to q if $q \ll c$.

The minimum amount of cache memory that is required to attain the lower bound (6) on the amount of memory traffic depends on the order in which the updates are applied. Figure 4 (*left*) shows the 19 reflections that belong to the first $q = 4$ sweeps of an $n = 20$, $r = 4$ problem. The dashed edges illustrate the fact that Q_k^{j-1} and Q_{k+1}^{j-1} must precede Q_k^j. First, consider the order in Figure 4 (*left*). The reflections are applied one *sweep* at a time, and therefore $c(k_1r - 1)$ entries must be cached from one sweep to the next in order to attain the lower bound (6). Second, consider the order in Figure 4 (*right*). The reflections are applied one *bulge* at a time, and only $c(q-1)$ entries must be cached from one bulge to the next in order to attain the lower bound (6). Thus, the order in Figure 4 (*right*) is inherently more effective. Another reason to prefer this order is that it can be used in conjunction with the compact WY representation, as explained in [5, 15].

6. Performance evaluation

Numerical experiments was carried out on one node of the Akka system at HPC2N. Akka consists of 672 interconnected nodes with a total of 5376 processor cores. Each

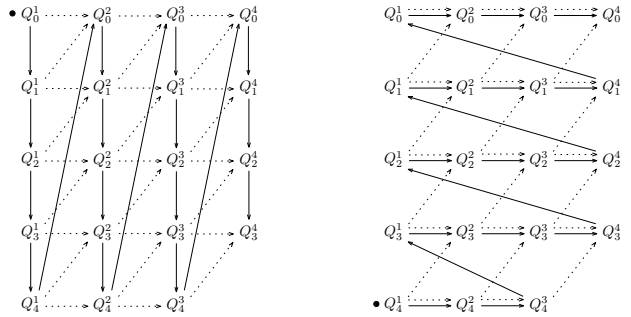

Figure 4: Two different ways to apply the updates. The order in which reflections are constructed (*left*) and the order in which they are applied (*right*). The sequence starts at the solid dot, •, and continues along the solid edges. The dashed edges illustrate the fact that Q_k^{j-1} and Q_{k+1}^{j-1} must precede Q_k^j.

node consists of two Intel Xeon L5420 quad-core processors with 12 MB of L2 cache. The processors run at 2.5 GHz and they share 16 GB of RAM. A node's theoretical double precision peak performance is $8 \cdot 2.5 \cdot 2 \cdot 2 = 80$ Gflop/s. The code was linked to the libraries GotoBLAS2 1.13, LAPACK 3.2.1, ScaLAPACK 1.8.0, BLACS 1.1, and OpenMPI 1.4.2. In order to reduce the impact of system noise, each problem configuration was run twice and we selected the shortest execution time.

We compare our implementation of the two-stage approach with the blocked one-stage Hessenberg reduction algorithms available in LAPACK and ScaLAPACK. The LAPACK code was linked against multithreaded BLAS and the ScaLAPACK code was linked against sequential BLAS.

The tuning parameters that are explicitly exposed to the user were set as follows. The multithreaded BLAS library and the two-stage implementation used eight threads, i.e., one thread per core. The ScaLAPACK code used a distribution block size of 64×64 and a 2×4 processor mesh.

The first stage of the two-stage implementation used $r = 12$, $b = 48$, and the fine-grained atoms consisted of 64 rows/columns each. The second stage used $q = 16$ consecutive sweeps (see [8]). The updates from the left-hand (right-hand) side were applied to blocks of 64 columns (32 rows).

The measured speedup with respect to LAPACK for various matrix sizes is reported in Figure 5. The ScaLAPACK implementation achieves its highest speedups (1.2–1.5) for matrices that almost fit in the L2 cache. For such small-sized problems, the two-stage implementation is considerably slower than the other implementations. The ScaLAPACK routine is somewhat slower than the LAPACK routine for large matrices. We do not have a good explanation for this behaviour.

The two-stage implementation obtains its highest speedups (close to 2.5) for large matrices. The computational kernels that apply the Householder reflections have a strong influence on the overall performance. The fine granularity of the computations would cause a significant per-call overhead if the kernels were based on the BLAS [8]. Therefore, the kernels have been implemented from scratch. Moreover, two versions of the kernels have been written in an attempt to estimate the effect of hardware-specific optimizations. The main version of the kernels is written in portable C with loops that are partially unrolled and then optimized by the compiler. The second version of the kernels is written

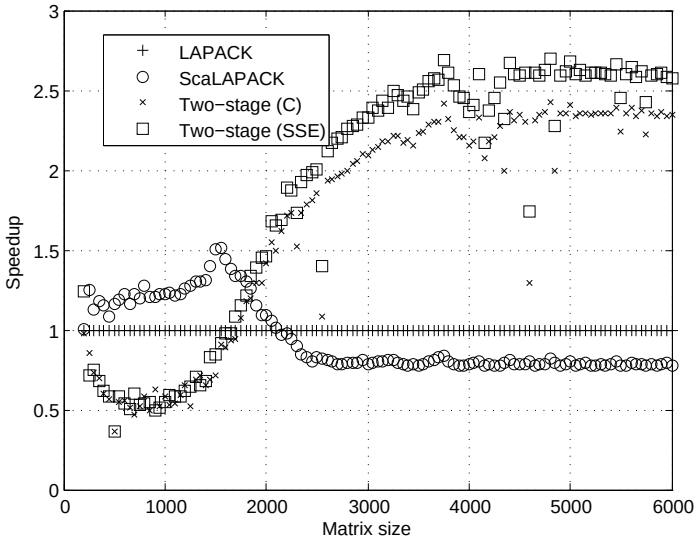

Figure 5: Speedup of four implementations of Hessenberg reduction compared to the LAPACK implementation.

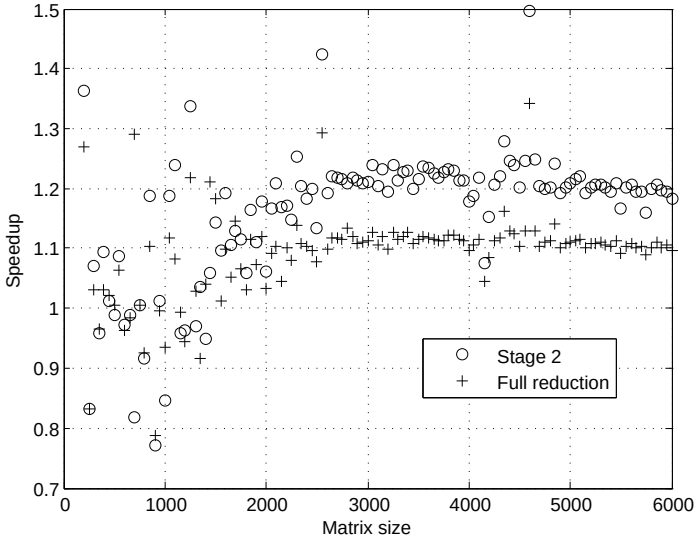

Figure 6: Speedup for the second stage and the entire Hessenberg reduction when using the SSE kernels in the second stage instead of the plain C kernels.

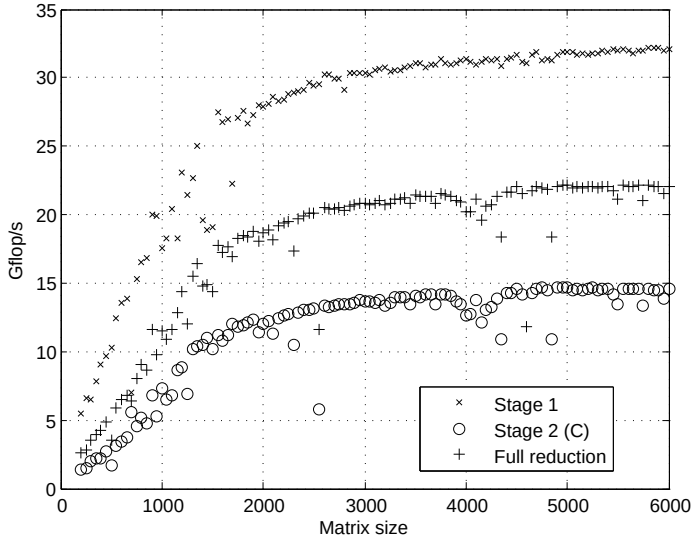

Figure 7: The performance of the two-stage implementation broken down into performance for the first stage, the second stage, and the full reduction.

in assembler and it uses SIMD instructions from the SSE instruction sets. According to Figure 6, the SSE kernels improve the performance of the second stage in isolation by around 20% and the full reduction by around 10%.

The performance of the individual stages as well as the full reduction are reported in Figure 7. The first stage peaks at around 30 Gflop/s, which is more than twice as fast as the second stage. The performance of the two stages combined peaks at around 22 Gflop/s, which corresponds to 28% of the theoretical peak.

7. Conclusions

The performance of the conventional blocked one-stage Hessenberg reduction algorithm is limited by large matrix–vector multiplications that account for 20% of the total number of flops.

We show that a two-stage approach to Hessenberg reduction can obtain high performance in both stages and thus overcome its 60% increase in the number of flops compared to the one-stage approach.

Experiments on a dual quad-core machine shows a performance of 22 Gflop/s, or 28% of the theoretical peak, for the complete Hessenberg reduction. This corresponds to a speedup of 2.5 over the one-stage implementation in LAPACK linked against multithreaded BLAS. The first stage peaks above 30 Gflop/s while the second stage peaks close to 15 Gflop/s.

The results indicate that an implementation for general matrix sizes should use the standard blocked algorithm for small matrices and the blocked two-stage algorithm for large matrices.

7.1. Related work

Recent developments related to the use of hybrid CPU+GPU systems have demonstrated that performing the matrix–vector multiplications as well as most of the matrix–matrix multiplications on a GPU can significantly boost the performance of blocked Hessenberg reduction [18]. Careful design of the algorithm in [18] ensures that the data transfers between the host and GPU memories do not outweigh the potential benefits. Specifically, the CPU and GPU need to be used in tandem instead of using the GPU as an accelerator at the BLAS layer.

So-called *tiled algorithms* or *algorithms-by-tiles* have recently received some attention. They typically use a combination of a blocked data layout with a data-flow driven scheduling of fine-grained tasks, each of which operates on a few small tiles/blocks of the matrix. A fast and scalable tiled algorithm for reduction to r-Hessenberg form, i.e., the first stage, is presented in [19]. The tiled algorithm incrementally reduces the tile rows of each tile column from left to right. The additional flops are compensated for by fine-grained parallelism and an effective memory access pattern.

The two-stage approach is analysed with respect to I/O efficiency in [20]. Sequential algorithms for both stages are proposed. The algorithm for the first stage is analogous to the one presented in this paper. The algorithm for the second stage is different from that in [8]. A chain of bulges are chased inside a sliding computational window. Updates to the off-diagonal entries are delayed and applied efficiently. However, the computational window becomes relatively large even for small increases in arithmetic intensity.

Acknowledgements

This research was conducted using the resources of the High Performance Computing Center North (HPC2N) and was supported by the *Swedish Research Council* (VR) under grant VR7062571 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128.

The work is also supported via eSSENCE, a strategic collaborative program in eScience funded by VR and the Swedish Government within the framework of national strategic research areas.

References

[1] C. F. Van Loan, Using the Hessenberg decomposition in control theory, in: Algorithms and Theory in Filtering and Control, Mathematical Programming Study No. 18, North-Holland, 1982.

[2] R. Granat, B. Kågström, D. Kressner, A novel parallel QR algorithm for hybrid distributed memory HPC systems, SIAM J. Sci. Comput. 32 (2010) 2345–2378.

[3] R. Granat, B. Kågström, Parallel solvers for Sylvester-type matrix equations with applications in condition estimation, part I: Theory and algorithms, ACM Trans. Math. Softw. 37 (2010) 1–32.

[4] R. Granat, B. Kågström, Algorithm 904: The SCASY library – parallel solvers for Sylvester-type matrix equations with applications in condition estimation, part II, ACM Trans. Math. Softw. 37 (2010) 1–4.

[5] C. Bischof, B. Lang, X. Sun, Parallel tridiagonalization through two-step band reduction, in: Scalable High-Performance Computing Conference, pp. 23–27.

[6] K. Dackland, B. Kågström, Blocked Algorithms and Software for Reduction of a Regular Matrix Pair to Generalized Schur Form, ACM Trans. Math. Software 25 (1999) 425–454.

[7] B. Kågström, D. Kressner, E. Quintana-Orti, G. Quintana-Orti, Blocked algorithms for the reduction to Hessenberg-triangular form revisited, BIT Numerical Mathematics 48 (2008) 563–584.

[8] L. Karlsson, B. Kågström, Efficient Reduction from Block Hessenberg Form to Hessenberg Form using Shared Memory, Technical Report UMINF 10.12, Department of Computing Science, SE-901 87, Umeå, Sweden, 2010. Submitted to PARA 2010.
[9] J. Choi, J. J. Dongarra, D. Walker, The Design of a Parallel Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal, and Bidiagonal Form, Technical Report UT-CS-95-275, University of Tennessee, Knoxville, TN, USA, 1995.
[10] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK User's Guide Third Edition, SIAM, Philadelphia, PA, USA, 1999.
[11] R. Schreiber, C. F. Van Loan, A storage-efficient WY representation for products of Householder transformations, SIAM J. Sci. Statist. Comput. 10 (1989) 53–57.
[12] J. Dongarra, S. Hammarling, D. Sorensen, Block reduction of matrices to condensed forms for eigenvalue computations, Journal of the ACM 27 (1989) 215–227.
[13] E. Elmroth, F. Gustavson, Applying recursion to serial and parallel QR factorization leads to better performance, IBM J. Research & Development 44 (2000) 605–624.
[14] E. Elmroth, F. Gustavson, I. Jonsson, B. Kågström, Recursive blocked algorithms and hybrid data structures for dense matrix library software, SIAM Review 46 (2004) 3–45.
[15] C. H. Bischof, B. Lang, X. Sun, A framework for symmetric band reduction, ACM Trans. Math. Software 26 (2000) 581–601.
[16] K. Murata, K. Horikoshi, A new method for the tridiagonalization of the symmetric band matrix, Information Processing in Japan 15 (1975) 108–112.
[17] H. R. Schwarz, Tridiagonalization of a symmetric band matrix, Numerische Mathematik 12 (1968) 231–241.
[18] S. Tomov, J. J. Dongarra, Accelerating the Reduction to Upper Hessenberg Form Through Hybrid GPU-Based Computing, Technical Report UT-CS-09-642, University of Tennessee Computer Science, 2009. Also as LAPACK Working Note 219.
[19] H. Ltaief, J. Kurzak, J. J. Dongarra, R. M. Badia, Scheduling two-sided transformations using tile algorithms on multicore architectures, Scientific Programming 18 (2010) 35–50.
[20] S. Mohanty, I/O Efficient Algorithms for Matrix Computations, Ph.D. thesis, Indian Institute of Technology Guwahati, 2009.
[21] G. Quintana-Ortí, R. A. van de Geijn, Improving the performance of reduction to Hessenberg form, ACM Trans. Math. Software 32 (2006) 180–194.
[22] H. Rutishauser, On Jacobi rotation patterns, in: Proc. Sympos. Appl. Math., volume XV, Amer. Math. Soc., Providence, R.I., 1963, pp. 219–239.
[23] M. W. Berry, J. J. Dongarra, Y. Kim, A parallel algorithm for the reduction of a nonsymmetric matrix to block upper-Hessenberg form, Parallel Computing 21 (1995) 1184–1200.