Parallel Two-Stage Hessenberg Reduction using Tile Algorithms for Multicore Architectures

Hatem Ltaief[1], Jakub Kurzak[1], and Jack Dongarra[1,2,3⋆]

[1] Department of Electrical Engineering and Computer Science,
University of Tennessee, Knoxville
[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory,
Oak Ridge, Tennessee
[3] School of Mathematics & School of Computer Science,
University of Manchester
{ltaief, kurzak, dongarra}@eecs.utk.edu

Abstract. This paper describes a parallel Hessenberg reduction in the context of multicore architectures using *tile algorithms*. The Hessenberg reduction is very often used as a pre-processing step in solving dense linear algebra problems, such as the standard eigenvalue problem. Although expensive, orthogonal transformations are accepted techniques and commonly used for this reduction because they guarantee stability, as opposed to elementarily transformations similar to what is used in Gaussian elimination. The state of the art, high performance dense linear algebra software libraries, i.e., LAPACK and ScaLAPACK, reduce the matrix to Hessenberg form through a one-stage process by using block Householder approach with compact WY representation. However, the main drawback of the *tile algorithms* approach for the Hessenberg reduction is that the full reduction can not be obtained similarly in a one-stage standard process. A two-stage approach has to be considered. The first stage corresponds to a redesign of the block Hessenberg matrix reduction, introduced by Dongarra *et. al* [12], to benefit from *tile algorithms* in the multicore environment. The second stage further reduces the matrix bandwidth to achieve the required Hessenberg form using a parallel bulge chasing procedure. On the one hand, by exploiting the concepts of *tile algorithms* in the multicore environment, the block Hessenberg reduction (first stage) achieves 72% of the DGEMM peak on a 12000×12000 matrix with 16 Intel Tigerton 2.4 GHz processors. On the other hand, the parallel bulge chasing procedure (second stage) is not appropriate for *tile algorithms* and therefore poorly performs on multicore architectures and slows down dramatically the overall algorithm.

1 Introduction

This paper describes a parallel Hessenberg Reduction (HR) in the context of multicore architectures using *tile algorithms*. The HR is very often used as a pre-

⋆ Research reported here was partially supported by the National Science Foundation and Microsoft Research.

processing step in solving dense linear algebra problems, such as the standard EigenValue Problem (EVP) [18]:

$$(A - \lambda I)\, x = 0,$$
$$with\ A \in \mathbb{R}^{n \times n},\ x \in \mathbb{C}^n,\ \lambda \in \mathbb{C}.$$

The need to solve EVPs emerges from various computational science disciplines including system and control theory, geophysics, molecular spectroscopy, particle physics, structure analysis, and so on. The basic idea is to transform the dense matrix A to an upper Hessenberg form H by applying successive transformations from the left (Q^T) as well as from the right (Q) as follows:

$$H = Q^T\, A\, Q,$$
$$A \in \mathbb{R}^{n \times n}\ ,\ Q \in \mathbb{R}^{n \times n}\ ,\ H \in \mathbb{R}^{n \times n}.$$

Although expensive, orthogonal transformations (i.e, Householder reflectors and Givens rotations) are accepted techniques and commonly used for this reduction because they guarantee stability, as opposed to elementarily transformations similar to what is used in Gaussian elimination [30].

Furthermore, in the last Top500 list from June 2009 [1], 99% of the fastest parallel systems in the world are based on multicores. This confronts the scientific software community with both a daunting challenge and a unique opportunity. The challenge arises from the disturbing mismatch between the design of systems based on this new chip architecture – hundreds of thousands of nodes, a million or more cores, reduced bandwidth and memory available to cores – and the components of the traditional software stack, such as numerical libraries, on which scientific applications have relied for their accuracy and performance. The manycore trend has even further exacerbated the problem, and it becomes judicious to efficiently develop existing or new numerical linear algebra algorithms suitable for such hardware. As discussed by Buttari *et al.* in [9], a combination of several parameters define the concept of *tile algorithms* and are essential to match the architecture associated with the cores: (1) Fine Granularity to reach a high level of parallelism and to fit the core small caches; (2) Asynchronicity to prevent any global barriers; (3) Block Data Layout (BDL), a high performance data representation to perform efficient memory access; and (4) Dynamic Data Driven Scheduler to ensure any queued tasks can immediately be processed as soon as all their data dependencies are satisfied. While bullets (1) and (3) represent important items for one-sided and two-sided transformations, (2) and (4) are even more critical for two-sided transformations because of the tremendous amount of tasks generated by such transformations. Indeed, as a comparison, the algorithmic complexity for the QR factorization used for least squares problems is $4/3\ n^3$ while it is $10/3\ n^3$ for the HR algorithm, with n the matrix size.

The state of the art, high performance dense linear algebra software libraries, i.e., LAPACK [7] (routine DGEHRD) and ScaLAPACK [10] (routine PDGEHRD), reduce the matrix to Hessenberg form through a one-stage process by using the block Householder approach with compact WY representation [26].

However, the main drawback of the *tile algorithms* approach for the HR is that the full reduction can not be obtained similarly in a one-stage standard process. A two-stage approach has to be considered. The first stage corresponds to a redesign of the block Hessenberg matrix reduction (BHR), introduced by Dongarra *et al.* [12], to benefit from *tile algorithms* in the multicore environment. The original dense matrix A is reduced to a block Hessenberg matrix H_b with b being the number of subdiagonals. Two versions of the BHR algorithms using *tile algorithms* for the first stage reduction are presented, the first one with Householder reflectors and the second one with Givens rotations. A short investigation on variants of Fast Givens rotations is also mentioned, but the work in this direction had to be resumed in favor of standard Givens rotations due to the lack of an efficient mechanism to accumulate the orthogonal transformations. The second stage annihilates those additional $b-1$ subdiagonals to achieve the required Hessenberg form by performing a parallel bulge chasing procedure using the parallel programming framework SMP Superscalar [3]. Unfortunately, this latter procedure does not run on top of *tile algorithms* and thus the obtained performance negates dramatically the overall performance of the two-stage approach on multicore architectures. It is also noteworthy to mention that this two-stage transformation process has already been studied in detail by Dackland *et. al* in [11] on single core, by Adlerborn *et. al* in [4] on distributed memory and more recently, by Kågström in [21] for the QZ algorithm (a slightly different problem, but still presents many similarities).

The reminder of this document is organized as follows: Section 2 recalls the standard HR algorithm and reviews the two orthogonal transformations based on Householder reflectors and Givens rotations. Section 3 describes the parallel implementation of the tile BHR algorithms for both orthogonal transformations as well as the bulge chasing procedure. Section 4 presents performance results of the two versions. Also, comparison tests are run on shared-memory architectures against the corresponding routines from LAPACK, ScaLAPACK and the vendor library MKL [2]. Finally, Section 5 summarizes the results of this paper and presents the ongoing work.

2 The Standard HR

In this section, we review the original HR algorithm as well as the orthogonal transformations based on Householder reflectors and Givens rotations. A short discussion on Fast Givens rotations is also included.

2.1 The Algorithm with Householder Reflectors

The standard HR algorithm based on Householder reflectors is written as in Algorithm 1. It takes as input the dense matrix A and gives as output the matrix in Hessenberg form. The reflectors v could be saved in the lower part of A for storage purposes and used later if necessary. The bulk of the computation is located in line 5 and in line 6 in which the reflectors are applied to A from

the left and then from the right, respectively. 4 flops are needed to annihilate one element of the matrix, which makes the total number of operations for such algorithm $10/3\ n^3$ (the lower order terms are neglected). It is obvious that Algorithm 1 is not efficient as described, especially for multicore architectures, not only because it is based on Level 2 BLAS, i.e., matrix-vector operations, but also because a single column is reduced at a time, which engenders a large stride access of memory. The whole idea is to transform this algorithm to work on small square chunks of data or tiles instead, in which elements are contiguous in memory (the processed data would then fit entirely in the cache) in order to improve data locality and cache reuse. The Householder reflectors can now be accumulated within the tiles and efficiently applied to the left and right of the matrix in the form of Level 3 BLAS, i.e., matrix-matrix operations.

Algorithm 1 Hessenberg Reduction with Householder reflectors

1: **for** $j\ =\ 1$ to $n-2$ **do**
2: $\quad x\ =\ A_{j+1:n,j}$
3: $\quad v\ =\ sign(x_1)\ ||x||_2\ e_1\ +\ x$
4: $\quad v\ =\ v\ /\ ||v||_2$
5: $\quad A_{j+1:n,j:n}\ =\ A_{j+1:n,j:n}\ -\ 2\ v\ (v^T\ A_{j+1:n,j:n})$
6: $\quad A_{1:n,j+1:n}\ =\ A_{1:n,j+1:n}\ -\ 2\ (A_{1:n,j+1:n}\ v)\ v^T$
7: **end for**

In the next section, we present a quite similar algorithm using standard Givens rotations.

2.2 The Algorithm with Standard Givens Rotations

The Givens rotation matrix is a rank-2 modification of the identity matrix and can be represented as follows:

$$g(i,j,\theta)\ =\ \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & & & s & \\ & & & 1 & & & \\ & & & & \ddots & & \\ & & & & & 1 & \\ & & -s & & & c & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{pmatrix}, \tag{1}$$

$$g\ \in\ \mathbb{R}^{n\times n},\ c = cos(\theta),\ s = sin(\theta),\ c^2\ +\ s^2\ = 1.$$

Let $x,\ y \in \mathbb{R}^n$. We have:

$$y \ = \ g(i,j,\theta)^T \ x \quad with \quad y_k = \begin{cases} c \times x_i - s \times x_j, & \text{if } k = i \\ s \times x_i + c \times x_j, & \text{if } k = j \\ x_k, & \text{otherwise.} \end{cases} \tag{2}$$

The multiplication $g(i,j,\theta)^T \ x$ is a counterclockwise rotation of x through an angle θ in the (i,j) plane and affects only the rows i and j. Therefore, if we want $y_j = 0$, then $c = \frac{x_i}{\sqrt{x_i^2+x_j^2}}$; $s = \frac{-x_j}{\sqrt{x_i^2+x_j^2}}$. In the same manner, this rotation can be applied from the right, i.e., $x \ g(j,i,\theta)$, and only columns j and i are involved. For the sake of simplicity, we omit θ in the formulation of the Givens rotations in the next algorithm.

The standard HR algorithm based on Givens rotations is written as in Algorithm 2. The cost to annihilate one element of the matrix with Givens rotations is 6 flops (Equation 2), which gives an overall operation count of $5n^3$, 50% more compared to the same reduction with Householder reflectors. Moreover, the predominance of vector-vector operations and the overhead of cache misses makes Givens rotations even less efficient. Thus, although their implementations are much simpler than Householder reflectors, Givens rotations are expected to poorly perform for the HR algorithm on multicore architectures. But, as seen later in Section 3, the algorithm can be expressed in term of tiles and can benefit from higher Level BLAS operations as well.

Algorithm 2 Hessenberg Reduction with Givens rotations

```
1: G ⟵ Id_n
2: for j = 1, 2 to n − 2 do
3:    for i = n, n − 1 to j + 2 do
4:       Build the local g(i − 1, i) such that A_{i,j} = 0
5:       Update A = g^T(i − 1, i) A
6:       Update A = A g(i, i − 1)
7:    end for
8: end for
```

The next section introduces the Fast Givens rotations, which are as expensive as Householder reflectors and still considered as orthogonal similarity, i.e., stable.

2.3 Discussion on Fast Givens Rotations

One disadvantage with standard Givens rotations is the two additional flops needed to annihilate one element of the matrix.

First introduced by Gentleman [16] and then by Hammarling [20] for the QR decomposition, Fast Givens rotations (FGs) are interesting because they only requires 4 flops to annihilate one element of the matrix. However, there are

major issues of arithmetic underflow or overflow and the proposed way to fix it actually creates overhead and may not improve performance, especially in the context of multicores.

Rath [25] applied FGs to the Jacobi method, the reduction to Hessenberg form and the QR algorithm for Hessenberg matrices. The novelty of his approach allows us to eliminate the computation of the square roots for the computation of cosine and sine. However, a close monitoring has to be done to avoid under/overflow and occasionally the matrix A has to be rescaled.

Anda and Park [6] introduced the self-scaling *chained* FGs, which delete the periodic rescaling that has been necessary to guard against under/overflow. While they focused only on the orthogonal one-sided transformations, their work could be easily extended to two-sided transformations. The idea basically is to decompose A as follows:

$$A = D\,Y, \text{ with } A,\ D \text{ and } Y\ \in\ \mathbb{R}^{n\times n},$$

with a diagonal matrix D and the scaled matrix Y. The goal is to dynamically scale the diagonal factor matrix D to be close to an identity matrix. For example, let us compute the new matrix $\hat{A}$ with the two bottom left elements located in a particular column annihilated:

$$\begin{aligned}\hat{A}^{(3)} &= D^{(3)}\ Y^{(3)}\\ &= D^{(3)}\ F^{(2)}\ Y^{(2)}\\ &= D^{(3)}\ F^{(2)}\ F^{(1)}\ Y^{(1)},\end{aligned}$$

with F representing the 2×2 FG matrix. F has this typical simplified form:

$$F = \begin{bmatrix}1 & 0\\ \beta & 1\end{bmatrix} \times \begin{bmatrix}1 & \alpha\\ 0 & 1\end{bmatrix},\ \alpha \text{ and } \beta\ \in\ \mathbb{R}, \tag{3}$$

and can be expressed up to eight different forms to ensure the diagonal elements of the matrix D stay within an absolute bound after any rotations. So, in the general case, we have:

$$\hat{A}^{(k+1)} = D^{(k+1)}\ F^{(k)}\ F^{(k-1)}\ \ldots\ F^{(1)}\ Y^{(1)}. \tag{4}$$

Let y_k be the k^{th} row of the matrix Y and $\hat{y}_k$ the corresponding transformed k^{th} row of the matrix Y. By plugging one of the particular form of F expressed from Equation (3) into Equation (4), we end up with the following equation:

$$\begin{aligned}\begin{bmatrix}\hat{y}_i\\ \hat{y}_j\end{bmatrix} &= F\ \begin{bmatrix}y_i\\ y_j\end{bmatrix}\\ &= \begin{bmatrix}1 & 0\\ \beta & 1\end{bmatrix}\ \begin{bmatrix}1 & \alpha\\ 0 & 1\end{bmatrix}\ \begin{bmatrix}y_i\\ y_j\end{bmatrix}\\ &= \begin{bmatrix}y_i{+}\alpha{\times}y_j\\ y_j{+}\beta{\times}(y_i + \alpha y_j)\end{bmatrix}.\end{aligned} \tag{5}$$

Achieving of 4 flops per zeroed element is done by *chaining* the components of the vectors $\hat{y}_i$ and $\hat{y}_j$, i.e., the components of the vector $\hat{y}_i$ have to be computed first and then their values are used to obtain the final result for the components of the vector $\hat{y}_j$. As opposed to the standard Givens rotations, where the updates are performed concurrently, the updates of the components with FGs are now serialized. But this is not really a limitation since the component pairs are contiguous in memory and each one would be loaded at once to the cache and then processed serially.

However, *chained* FGs present one major drawback. A bookkeeping procedure has to be established on the fly to save each specific form of the chosen matrices F during the reduction to guard against under/overflow that may occur in the diagonal elements of D. Unfortunately, this presents a major complexity for implementing an automatic mechanism to efficiently aggregate the small FG matrices within a tile in order to produce Level 3 BLAS operations during the left and right updates. Thus, even though more expensive, standard Givens rotations are preferred to FGs in order to efficiently generate Level 3 BLAS operations.

In the next section, we explain the modifications applied in Algorithms 1 and 2 to achieve high performance on multicore architectures using the concepts of *tile algorithms*.

3 The Parallel Two-Stage Hessenberg Reduction

In this section, we describe the two-stage approach to reduce a general matrix to Hessenberg form using *tile algorithms*. The idea is to decompose the original matrix into tiles, in which elements are contiguous in memory. The matrix elements are now annihilated by square tiles. The first subsection explains why a two-stage approach is mandatory when using *tile algorithms*. Then, we present two parallel implementations of the BHR based on Householder reflectors and Givens rotations. The parallel bulge chasing procedure as well as the overall algorithmic complexity are detailed in the following subsection. Finally, two dynamic data driven schedulers to process the kernels in parallel are outlined.

3.1 Limitations of the *Tile Algorithms* Approach for HR

The concept of *tile algorithms* is very suitable for one-sided methods (i.e., Cholesky, LU and QR/LQ). Indeed, the transformations are only applied to the matrix from one side. With the two-sided methods, the right transformation needs to preserve the reduction achieved by the left transformation. In other words, the right transformation should not destroy the zeroed structure by creating fill-in elements. That is why the only way to keep intact the obtained structure is to perform a shift of a tile in the adequate direction. In our BHR algorithm, the reduction is shifted one tile bottom from the top-left corner of the matrix. Therefore, the reduced matrix then has an upper block Hessenberg shape (first stage) and needs to be further reduced (second stage) to get the required Hessenberg form.

3.2 First Stage: The Block Hessenberg Reduction

In this stage, the BHR can be achieved through the use of fast orthogonal transformation kernels, based on Householder reflectors and Givens rotations, as described below. Furthermore, by using updating factorization techniques as suggested in [18, 28], the kernels for both implementations can be applied to tiles of the original matrix. Using updating techniques to tile the algorithms was first proposed by Yip [31] for LU to improve the efficiency of out-of-core solvers, and were recently reintroduced in [19, 23] for LU and QR, once more in the out-of-core context.

Householder Reflectors There are four kernels to perform the tile BHR kernels based on Householder reflectors. Let A be a matrix composed by $nt \times nt$ tiles of size $b \times b$. Let $A_{i,j}$ represent the tile located at the row index i and the column index j.

- **CORE_DGEQRT**: this kernel performs the *QR* blocked factorization of a subdiagonal tile $A_{k,k-1}$ of the input matrix. It produces an upper triangular matrix $R_{k,k-1}$, a unit lower triangular matrix $V_{k,k-1}$ containing the Householder reflectors and an upper triangular matrix $T_{k,k-1}$ as defined by the WY technique [26] for accumulating the transformations. $R_{k,k-1}$ and $V_{k,k-1}$ are written on the memory area used for $A_{k,k-1}$ while an extra work space is needed to store $T_{k,k-1}$. The upper triangular matrix $R_{k,k-1}$, called *reference tile*, is eventually used to annihilate the subsequent tiles located below, on the same panel.
- **CORE_DTSQRT**: this kernel performs the *QR* blocked factorization of a matrix built by coupling the reference tile $R_{k,k-1}$ that is produced by CORE_DGEQRT with a tile below the diagonal $A_{i,k-1}$. It produces an updated $R_{k,k-1}$ factor, $V_{i,k-1}$ matrix containing the Householder reflectors and the matrix $T_{i,k-1}$ resulting from accumulating the reflectors $V_{i,k-1}$.
- **CORE_DORMQR**: this kernel is used to apply the transformations computed by CORE_DGEQRT $(V_{k,k-1}, T_{k,k-1})$ to the tile row $A_{k,k:nt}$ (left updates) and the tile column $A_{1:nt,k}$ (right updates).
- **CORE_DTSSSMQR**: this kernel applies the reflectors $V_{i,k-1}$ and the matrix $T_{i,k-1}$ computed by CORE_DTSQRT to two tile rows $A_{k,k:nt}$ and $A_{i,k:nt}$ (left updates), and two tile columns $A_{1:nt,k}$ and $A_{1:nt,i}$ (right updates).

Compared to the tile *QR* kernels used by Buttari *et. al* in [9], the right variants for CORE_DORMQR and CORE_DTSSSMQR have been developed. The other kernels are exactly the same as [9]. The tile BHR algorithm with Householder reflectors then appears as in Algorithm 3. Figure 1 shows the BHR algorithm applied on a matrix with nt=5 tiles in each direction. The dark gray tile is the processed tile at the current step using as input dependency the black tile, the white tiles are the tiles zeroed so far, the bright gray tiles are those which still need to be processed and the striped tile represents the final data tile. For example, in Figure 1(a), a subdiagonal tile (in dark gray) of the first panel is reduced

Algorithm 3 Tile BHR Algorithm with Householder reflectors.

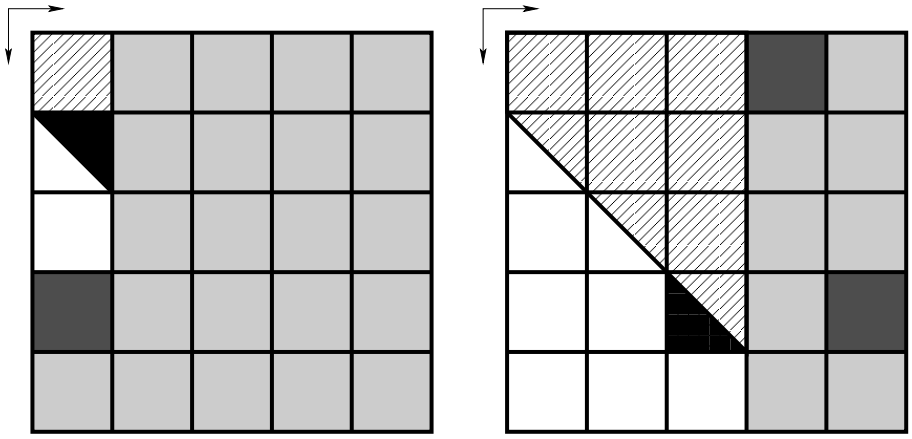
1: **for** $k = 2$ to nt **do**
2: $R_{k,k-1}, V_{k,k-1}, T_{k,k-1} \leftarrow$ CORE_DGEQRT($A_{k,k-1}$)
3: **for** $j = k$ to nt **do**
4: $A_{k,j} \leftarrow$ CORE_DORMQR(*left*, $V_{k,k-1}, T_{k,k-1}, A_{k,j}$)
5: **end for**
6: **for** $j = 1$ to nt **do**
7: $A_{j,k} \leftarrow$ CORE_DORMQR(*right*, $V_{k,k-1}, T_{k,k-1}, A_{j,k}$)
8: **end for**
9: **for** $i = k+1$ to nt **do**
10: $R_{k,k-1}, V_{i,k-1}, T_{i,k-1} \leftarrow$ CORE_DTSQRT($R_{k,k-1}, A_{i,k-1}$)
11: **for** $j = k$ to nt **do**
12: $A_{k,j}, A_{i,j} \leftarrow$ CORE_DTSSSMQR(*left*, $V_{i,k-1}, T_{i,k-1}, A_{k,j}, A_{i,j}$)
13: **end for**
14: **for** $j = 1$ to nt **do**
15: $A_{j,k}, A_{j,i} \leftarrow$ CORE_DTSSSMQR(*right*, $V_{i,k-1}, T_{i,k-1}, A_{j,k}, A_{j,i}$)
16: **end for**
17: **end for**
18: **end for**

using the upper structure of the reference tile (in black). This operation is done by the kernel CORE_DTSQRT($R_{2,1}, A_{4,1}, T_{4,1},$). In Figure 1(b), the reference tile (in black) of the third panel is reduced and its corresponding updates are applied, e.g., the top dark gray tile is CORE_DORMQR(*right*, $V_{4,3}, T_{4,3}, A_{1,4}$) while the bottom one is CORE_DORMQR(*left*, $V_{4,3}, T_{4,3}, A_{4,5}$). These update kernels are exclusively performed in Level 3 BLAS thanks to the WY technique for accumulating the transformations. Moreover, by working on small tiles with Block Data Layout (BDL), the elements are stored contiguous in memory and thus the access pattern to memory is more regular, which makes these kernels achieving high performance.

In the following part, we present a similar BHR approach based exclusively on Givens rotations.

Givens Rotations The kernels of the BHR algorithm with Givens rotations are much simpler than those with Householder reflectors. Except for the factorization kernels, all are straight calls to the BLAS library. The skeleton of Algorithm 4 with Givens rotations is exactly the same as Algorithm 3 with Householder reflectors. This can be proved by analyzing the dependencies between the kernels in the directed acyclic graph (DAG) generated by both algorithms. The nodes represent tasks, either panel factorization or update of a block-row or a block-column, and edges represent dependencies among them. Both generated DAGs are identical.

– **CORE_DGEGRG**: this kernel annihilates a subdiagonal tile $A_{k,k-1}$ of the input matrix with Givens rotations. It produces an upper triangular matrix $R_{k,k-1}$ and a dense Givens rotation matrix $G_{k,k-1}$ of size $b \times b$ resulting from



(a) BHR: Reduction Step 1. (b) BHR: Reduction Step 3.

Fig. 1. BHR algorithm applied on a tiled Matrix with nt= 5.

the aggregation of the local Givens rotation transformation matrices, in order to generate matrix-matrix operations during the left and the right updates. These explicit accumulations are performed using a recursive formula, as explained by Gill *et. al* in [17]. The components c and s of the local Givens rotation matrix can also be saved in a single element (see Stewart in [27]) and stored in the lower zeroes part of the tile $A_{k,k-1}$. $R_{k,k-1}$ is written on the memory area used for $A_{k,k-1}$, while an extra work space is needed to store the matrix $G_{k,k-1}$. The upper triangular reference tile $R_{k,k-1}$ is eventually used to annihilate the subsequent tiles located below, on the same panel.

- **CORE_DTSGRG**: this kernel eliminates a tile $A_{i,k-1}$ using the reference tile $R_{k,k-1}$ that is produces by CORE_DGEGRG. It yields an updated $R_{k,k-1}$ factor and a Givens rotation matrix $G_{i,k-1}$ of size $2b \times 2b$ with a shape depicted in Figure 2, resulting from accumulating the local Givens rotation matrices.
- **CORE_DORMGR**: this kernel is in fact a single call to DGEMM BLAS routine. It applies the rotations $G_{k,k-1}$ computed by CORE_DGEGRG to the tile row $A_{k,k:nt}$ (left updates) and the tile column $A_{1:nt,k}$ (right updates).
- **CORE_DTSSSMGR**: this kernel corresponds to two successive calls to DTRMM and DGEMM, i.e., a call to DTRMM for the lower triangular structure followed by a call to DGEMM for the upper square structure (see Figure 2). It applies the rotations $G_{i,k-1}$ computed by CORE_DTSQRT to two tile rows $A_{k,k:nt}$ and $A_{i,k:nt}$ (left updates), and two tile columns $A_{1:nt,k}$ and $A_{1:nt,i}$ (right updates).

The tile BHR algorithm with Givens rotations then appears as in Algorithm 4. Figure 3 and 4 illustrate step-by-step the sequential execution of Algorithms 3 and 4, with Householder reflectors and Givens rotations in order to eliminate the first panel of a tile width. More specifically, in Figure 3, the first matrix

Algorithm 4 Tile BHR Algorithm with Givens rotations.

1: **for** $k = 2$ to nt **do**
2: $R_{k,k-1}, G_{k,k-1} \leftarrow$ CORE_DGEGRG($A_{k,k-1}$)
3: **for** $j = k$ to nt **do**
4: $A_{k,j} \leftarrow$ CORE_DORMGR(*left*, $G_{k,k-1}, A_{k,j}$)
5: **end for**
6: **for** $j = 1$ to nt **do**
7: $A_{j,k} \leftarrow$ CORE_DORMGR(*right*, $G_{k,k-1}, A_{j,k}$)
8: **end for**
9: **for** $i = k+1$ to nt **do**
10: $R_{k,k-1}, G_{i,k-1} \leftarrow$ CORE_DTSGRG($R_{k,k-1}, A_{i,k-1}$)
11: **for** $j = k$ to nt **do**
12: $A_{k,j}, A_{i,j} \leftarrow$ CORE_DTSSSMGR(*left*, $G_{i,k-1}, A_{k,j}, A_{i,j}$)
13: **end for**
14: **for** $j = 1$ to nt **do**
15: $A_{j,k}, A_{j,i} \leftarrow$ CORE_DTSSSMGR(*right*, $G_{i,k-1}, A_{j,k}, A_{j,i}$)
16: **end for**
17: **end for**
18: **end for**

row corresponds to the factorization of the reference tile with its associated left updates. The second matrix row shows the factorization of the subdiagonal tile (using the reference tile) and its associated left updates, and so on. In Figure 4, the first matrix column shows the right updates relative to the factorization of the reference tile. The second matrix column represents the right updates relative to the factorization of the subdiagonal tile and so on. It appears necessary then to efficiently schedule the kernels to get high performance in parallel. In particular, the left and right updates can actually be interleaved, i.e., the right updates do not need to wait for the left updates to completely finish to be triggered. Indeed, the scheduling of these update kernels can result to an out-of-order execution, as seen in Section 3.5.

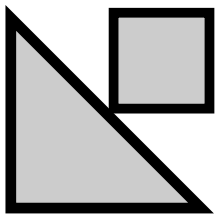


Fig. 2. Givens rotation matrix produced by CORE_DTSQRG used during the update procedures.

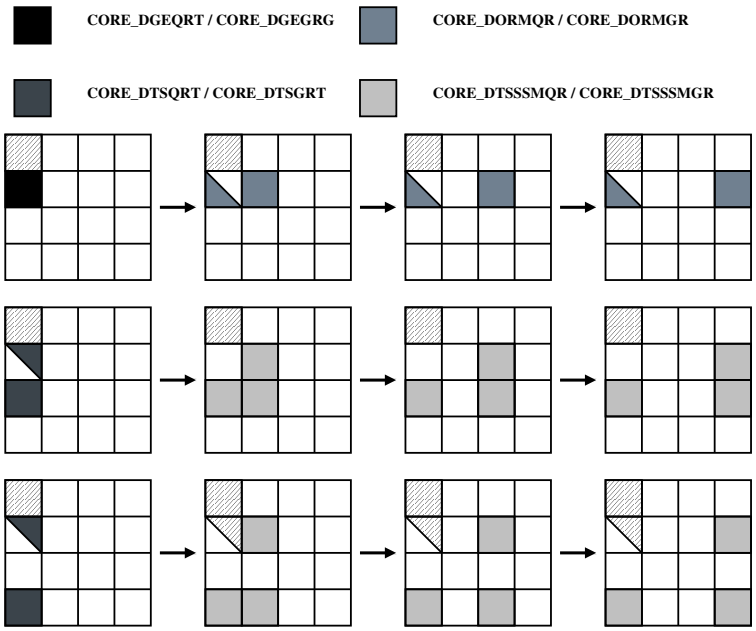


Fig. 3. Scheduling of the Left Orthogonal Transformation.

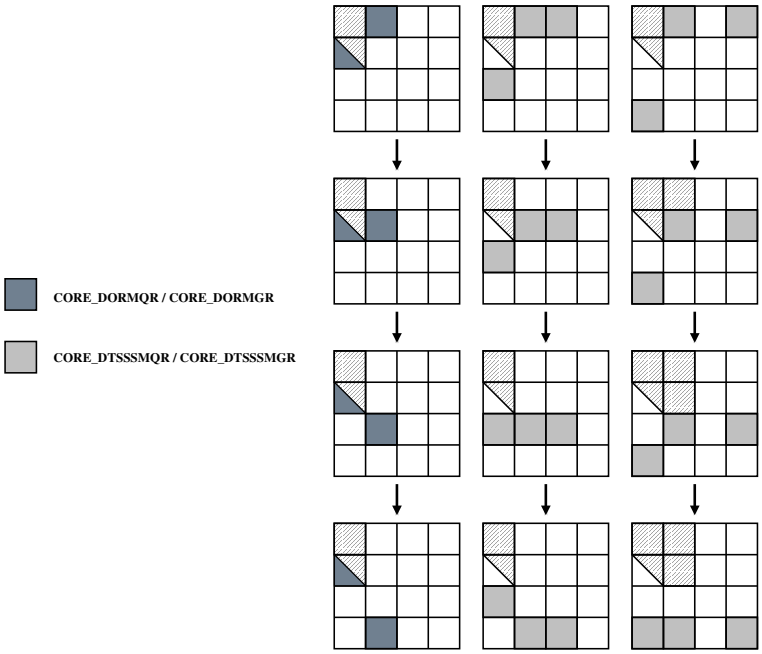


Fig. 4. Scheduling of the Right Orthogonal Transformation.

In the following subsection, we present the second stage of the HR, which further reduces the matrix to the Hessenberg form using a parallel bulge chasing procedure.

3.3 Second Stage: The Parallel Bulge Chasing Procedure

In the second stage, the block Hessenberg matrix H_b is further reduced to acquire the full Hessenberg form by implementing a bulge chasing procedure. This well-known procedure annihilates the nonzero lower $b-1$ subdiagonals. Typically, the elimination of a single element of the first subdiagonal will create a 2×2 bulge at the top of the matrix. This bulge needs to be chased down to the end of the matrix by successively applying orthogonal transformations (i.e., Householder reflectors or Givens rotations) from the left and right side of the matrix. However, an element-wise elimination is not efficient because it involves only Level 1 BLAS. A blocked bulge chasing procedure is rather preferred in which an entire column is first reduced. The orthogonal transformations are then accumulated and the left and right updates are delayed. Then, the corresponding updates are applied from the left and right side of the matrix in order to chase the $b \times b$ bulges down to the end of the matrix. This chasing procedure is described in Figure 5 for the first supersweep on a matrix with $nt = 4$ and $b = 4$. The red and green rectangles correspond to the left and right updates respectively, i.e., CORE_DORMQR or CORE_DORMGR, depending on whether the procedure is performed with Householder reflectors or Givens rotations. To limit the fill-in in the unreduced part of H_b, the $b \times b$ bulge needs to be chased down before any subsequent orthogonal transformations are applied on top of it. The blocked bulge chasing procedure is more efficient because it is mainly based on Level 3 BLAS. This technique is also described by Dackland *et. al* in [11] and by Adlerborn *et. al* in [4] for the QZ algorithm. To parallelize this procedure, we used the parallel programming framework SMP Superscalar (SMPSs) [3]. This framework allows us to perform pipelining strategy between each column annihilation or *supersweep* and ensures that the left and right updates are applied in the correct order (i.e., dependencies are not violated). More details about this framework are provided in the Section 3.5.

However, the blocked bulge chasing technique presents three major drawbacks:

- The number of floating-point operations engendered is significantly higher and is at least half the number of floating-point operations required to achieve the full reduction, i.e. $10/3\ n^3$, depending on the tile size b.
- If the eigenvectors are also needed by the user, an n^3 term will be added to the operation count of the back transformation procedure.
- And most of all, we can not even trade off the additional floating-point operations for a better parallel implementation on multicore architectures because this technique can not be expressed into *tile algorithms*. Between each supersweep, a shift down of a single element is performed (and not a shift of tile), which generates a mismatch with the block data layout (BDL) used for

tile algorithms. Indeed, a $b \times b$ bulge created during a supersweep may reside within two different tiles. For example, the right updates (green rectangle) in Figure 5(a) create a $b \times b$ block, which is shared by two different tiles. In the same way, by chasing down the bulges, most of the updates carried during the supersweep are applied to two different tiles at the same time. This would require a significant change in the actual kernel implementation and would introduce a major complexity in the scheduling scheme to detect overlapping regions. Therefore, the parallel blocked bulge chasing technique has been implemented such that it runs on top of the LAPACK data layout instead (column-major).

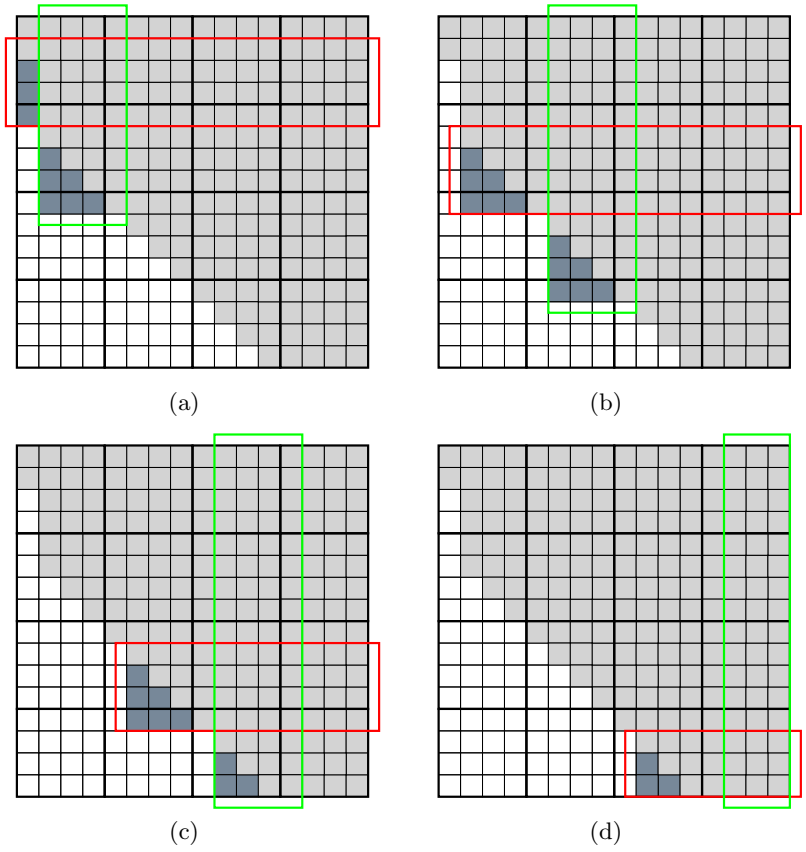


Fig. 5. Chasing the bulges during the first supersweep.

In the next subsection, the number of floating-point operations required for the entire two-stage approach is presented with Householder reflectors and Givens rotations.

3.4 Algorithmic Complexity

If an unblocked algorithm is used with Householder reflectors (see Algorithm 1), the algorithmic complexity for the BHR is $10/3\ n\ (n-b)\ (n-b)$, with b being the tile size. So, compared to the full HR complexity, i.e., $10/3\ n^3$, the BHR algorithm with Householder reflectors is performing $O(n^2\ b)$ fewer flops. With Givens rotations (see Algorithm 2), the algorithmic complexity of the corresponding unblocked algorithm is $5\ n\ (n-b)\ (n-b)$, i.e., 50% more compared to the one with Householder reflectors.

In the *tile algorithm* with Householder reflectors presented in Algorithm 3, we recall the four kernels and give their complexity:

- **CORE_DGEQRT**: $4/3b^3$ to perform the factorization of the reference tile $A_{k,k-1}$ and $2/3b^3$ for computing $T_{k,k-1}$.
- **CORE_DORMQR**: since $V_{k,k-1}$ and $T_{k,k-1}$ are triangular, $3b^3$ floating-point operations are performed in this kernel.
- **CORE_DTSQRT**: $2b^3$ to perform the factorization of the subdiagonal tile $A_{i,k-1}$ using the reference tile $A_{k,k-1}$ and $2/3b^3$ for computing $T_{i,k-1}$, which overall gives $10/3b^3$ floating-point operations.
- **CORE_DTSSSMQR**: by exploiting the structure of $V_{i,k-1}$ and $T_{i,k-1}$, $5b^3$ floating-point operations are needed by this kernel.

More details can be found in [8]. The total number of floating-point operations for the BHR with Householder reflectors is then:

$$\begin{aligned}\sum_{k=2}^{nt}(2b^3 + 3(nt-k)b^3 + \frac{10}{3}(nt-k)b^3 \\ + 5(nt-k)^2b^3 + 3b^3nt + 5nt(nt-k+1)b^3) \\ \simeq \frac{5}{3}nt^3b^3 + \frac{5}{2}nt^3b^3 \\ = \frac{5}{3}n^3 + \frac{5}{2}n^3 \\ = \frac{25}{6}n^3,\end{aligned} \tag{6}$$

which is 25% higher than the unblocked algorithm for the same reduction. Indeed, the cost of these updating techniques is an increase in the operation count for the BHR reduction. However, as suggested in [13–15], by setting up inner-blocking within the tiles during the panel factorizations as well as the trailing submatrix updates (i.e., left and right), CORE_DGEQRT-CORE_DTSQRT kernels and CORE_DORMQR-CORE_DTSSSMQR kernels respectively, those extra flops become negligible provided $s \ll b$, with s being the inner-blocking size. The inner-blocking size trades off actual memory load with those extra-flops. This blocking approach has also been described in [19, 24]. To understand how this cuts the operation count of the BHR algorithm, it is important to note that the CORE_DGEQRT, CORE_DORMQR and CORE_DTSQRT kernels only account for lower order terms in the total operation count for the tile block Hessenberg algorithm (see Equation 6). It is, thus, possible to ignore these terms and

derive the operation count for the tile block Hessenberg algorithm as the sum of the cost of all the CORE_DTSSSMQR kernels. The sequential performance of this most compute intensive kernel can be found in [5][4]. The $T_{i,k-1}$ generated by CORE_DTSQRT and used by CORE_DTSSSMQR are not upper triangular anymore but becomes upper-triangular by block thanks to inner-blocking. The cost of a single CORE_DTSSSMQR call drops down, and by ignoring the lower order terms, it is now $4b^3 + sb^2$. The total cost of the tile block Hessenberg algorithm with internal blocking is then:

$$\begin{aligned}\sum_{k=2}^{nt}((4b^3+sb^2)(nt-k)^2 + nt(nt-k+1)(4b^3+sb^2)) \qquad (7)\\ \simeq (4b^3+sb^2)(\frac{1}{3}nt^3+\frac{1}{2}nt^3)\\ = (1+\frac{s}{4b})(\frac{4}{3}n^3+2n^3).\end{aligned}$$

The operation count for the tile block Hessenberg algorithm with internal blocking is bigger than that of the unblocked algorithm only by the factor $(1 + \frac{s}{4b})$, which is negligible, provided that $s \ll b$. Note that, in the case where $s = b$, the tile block Hessenberg algorithm performs 25% more floating-point operations than the unblocked algorithm, as stated before. So, by adding the complexity of this first stage with the one of the bulge chasing procedure, we end up with a total of at least $5n^3$ floating-point operations for the full Hessenberg reduction, i.e., at least 50% higher than the one required by the unblocked algorithm.

In the *tile algorithm* with Givens rotations presented in Algorithm 4, we recall the four kernels and give their complexity:

- **CORE_DGEGRG**: $2b^3$ to perform the factorization of $A_{k,k-1}$ and $3b^3$ for computing the explicit accumulation of the Givens rotations, i.e., $G_{k,k-1}$.
- **CORE_DORMGR**: this kernel consists of a single call to DGEMM, i.e., it multiplies $A_{k,j}$ by $G_{k,k-1}$ of size b for $j \in k..nt$, where $2b^3$ floating-point operations are thus performed in this kernel.
- **CORE_DTSGRG**: $3b^3$ to perform the factorization of the subdiagonal tile $A_{i,k-1}$ using the reference tile $A_{k,k-1}$ and $6b^3$ for computing $G_{i,k-1}$ of size $2b \times 2b$.
- **CORE_DTSSSMGR**: this kernel calls successively DTRMM on $2b$ matrix size and DGEMM on b matrix size and therefore $2b^3 + 4b^3 = 6b^3$ floating-point operations are needed by this kernel.

The panel factorization kernels, i.e., CORE_DGEGRG and CORE_DTSGRG, are expensive due the explicit accumulation of the Givens matrices compared to those with Householder reflectors, i.e., CORE_DGEQRT and CORE_DTSQRT.

[4] Note that the name of the kernel has changed since, from dssrfb to CORE_DTSSSMQR.

The total number of floating-point operations for the BHR with Givens rotations is then:

$$\begin{aligned}\sum_{k=2}^{nt}(5b^3 + 2(nt-k)b^3 + 9b^3(nt-k) \\ + 6(nt-k)^2b^3 + 2b^3nt + 6nt(nt-k+1)b^3) \\ \simeq 5b^3nt^3 + \frac{11}{2}b^3nt^2 \\ = 5n^3 + \frac{11}{2}n^2b,\end{aligned} \tag{8}$$

which is 50% higher than the unblocked algorithm for the same reduction, assuming $b \ll n$. The authors are not aware of any existing techniques for Givens rotations that could decrease the amount of the extra-flops, as has been done with Householder reflectors. Finally, by adding the complexity of this first stage with the one of the bulge chasing procedure, we end up with a total of at least $15/2n^3$ floating-point operations for the full Hessenberg reduction, i.e., at least 125% higher than the one required by the unblocked algorithm.

In the following part, we present two dynamic data driven execution schedulers that ensure the small tasks (or kernels) generated by Algorithms 3 and 4 are processed as soon as their respective dependencies are satisfied.

3.5 Dynamic Data Driven Execution

Hand-Coded Dynamic Data Driven Scheduler A dynamic scheduling scheme similar to [9] has been extended for the two-sided orthogonal transformations. The dynamic scheduler maintains a central progress table, which is accessed in the critical section of the code and protected with mutual exclusion primitives (POSIX mutexes in this case). Each thread scans the table to fetch one task at a time for execution. As long as there are tasks with all dependencies satisfied, the scheduler will provide them to the requesting threads and will allow an out-of-order execution. The scheduler does not attempt to exploit data reuse between tasks. The centralized nature of the scheduler is inherently non-scalable with the number of threads. Also, the need for scanning a potentially large table window, in order to find work, is inherently non-scalable with the problem size. However, this organization does not cause performance problems for the numbers of threads, problem sizes and task granularities investigated in this paper. A Directed Acyclic Graph (DAG) can be used to represent the data flow between the nodes/kernels. While the DAG is quite easy to draw for a small number of tiles, it becomes very complex when the number of tiles increases and it is even more difficult to process than the one created by the one-sided orthogonal transformations. Indeed, the right updates impose robust constraints on the scheduler by filling up the DAG with multiple additional edges.

Figure 6 shows a complete tracing of the dynamic data driven scheduler using eight cores, performing in that particular experiment the BHR with House-

holder reflectors (the same figure could be generated with Givens rotations as well) with a matrix size $n = 2000$ and a tile size $b = 200$. The four different kernels are clearly identified with their colors. The red color represents the CORE_DGEQRT kernel, the green color is the CORE_DTSQRT kernel, the blue and violet colors stand for the CORE_DORMQR kernel for the left and right transformations, respectively, and finally, the yellow and pink colors correspond to the CORE_DTSSSMQR kernel for the the left and right orthogonal transformations, respectively. From this figure, the first task scheduled is obviously the CORE_DGEQRT kernel, which allows us to eventually generate many tasks for the other cores. This particular task is indeed part of the critical path and needs to be scheduled at a high priority level. As a matter of fact, the next CORE_DGEQRT of the second panel factorization is scheduled although the updates of the first panel are still being computed. Also, it is interesting to see what the tasks are that can run concurrently without violating the dependencies. For example, the CORE_DORMQR kernels, whether it is a left or right transformation, can be run independently from each other, except in the case of updating the tile $A_{k,k}$, in which the left transformation has to advance the right transformation. The CORE_DTSQRT kernels that do the panel factorization are sequential, i.e., they are called successively one after the other, but at the same time, they run concurrently with the CORE_DORMQR kernels. The CORE_DTSSSMQR kernels, represented by the yellow and pink colors, are clearly interleaved, which demonstrates the out-of-order execution of the DAG. Furthermore, Figure 6 shows that all the idle times, which represent the major scalability limit of the fork-join approach, can be removed thanks to the very low synchronization requirements of the graph driven execution. The idle time present at the end of the execution trace is due to the limited amount of parallelism among the very last tasks of the DAG. This is characteristic of the load imbalance in linear algebra algorithms. The graph driven execution also provides some degree of adaptivity since tasks are scheduled to threads depending on the availability of execution units.

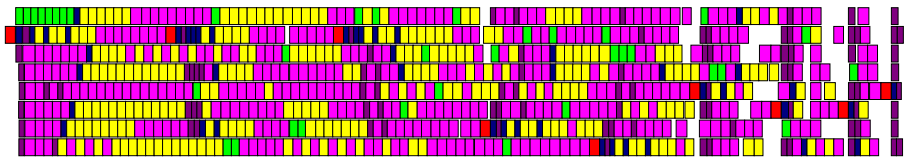


Fig. 6. Tracing of Dynamic Data Driven Execution with 8 cores. *Red*: CORE_DGEQRT; *Green*: CORE_DTSQRT; *Blue*, *Violet*: CORE_DORMQR, left and right updates resp.; *Yellow*, *Pink*: CORE_DTSSSMQR, left and right updates resp.

SMP Superscalar Framework SMP Superscalar (SMPSs) [3] is a parallel programming framework developed at the Barcelona Supercomputer Center

(Centro Nacional de Supercomputación). SMPSs is a dynamic scheduler implementation that addresses the automatic exploitation of the functional parallelism of a sequential program in multicore and symmetric multiprocessor environments. SMPSs allows the programmers to write sequential applications, and the framework is able to exploit the existing concurrency and to use the different processors by means of an automatic parallelization at execution time. The programmer is responsible for identifying parallel tasks, which have to be side-effect-free (atomic) functions. However, the user is not responsible for exposing the structure of the task graph. The task graph is built automatically, based on the information of task parameters and their directionality. Construction of the DAG does, however, introduce overhead. The programming environment consists of a source-to-source compiler and a supporting runtime library. The compiler translates C code with pragma annotations to standard C99 code with calls to the supporting runtime library and compiles it using the platform native compiler. The runtime takes care of scheduling the tasks and handling the associated data. The SMPSs scheduler attempts to exploit locality by scheduling dependent tasks to the same thread, such that output data is reused immediately. Also, in order to reduce dependencies, SMPSs runtime is capable of renaming data, leaving only the true dependencies. At runtime the main thread creates worker threads, as many as necessary to fully utilize the system, and starts constructing the task graph (populating its ready list). Each worker thread maintains its own ready list and populates it while executing tasks. A thread consumes tasks from its own ready list in LIFO order. If that list is empty, the thread consumes tasks from the main ready list in FIFO order, and if that list is empty, the thread steals tasks from the ready lists of other threads in FIFO order.

SMPSs is in particular very useful when dealing with sequential algorithms that may not be that obvious to implement in parallel, especially those which can not be expressed in friendly data layout, i.e., tiles / BDL. Indeed, the sequential bulge chasing procedure is one example, because the detection of overlapping regions during supersweeps is very challenging and critical to ensure efficient scheduling and, at the same time, numerical correctness. Therefore, the parallel implementation of the bulge chasing procedure presented in this paper relies totally on SMPSs.

In the next section, we present the experimental results comparing our two BHR implementations (i.e., with Householder reflectors and Givens rotations) associated with the parallel bulge chasing procedure (only Householder reflectors are considered for that procedure) against the state of the art library, i.e., LAPACK [7], ScaLAPACK [10] and MKL 10.1 [2].

4 Experimental Results

4.1 Experimental Environment

The experiments have been performed on a quad-socket quad-core machine based on an Intel Xeon EMT64 E7340 processor operating at 2.39 GHz. The theoretical peak is equal to 9.6 Gflop/s/ per core or 153.2 Gflop/s for the whole node,

composed of 16 cores. There are two levels of cache. The level-1 cache, local to the core, is divided into 32 kB of instruction cache and 32 kB of data cache. Each quad-core processor being actually composed of two dual-core Core2 architectures, the level-2 cache has 2×4 MB per socket (each dual-core shares 4 MB). The effective bus speed is 1066 MHz per socket leading to a bandwidth of 8.5 GB/s (per socket). The machine is running Linux 2.6.25 and provides Intel Compilers 11.0 together with the MKL 10.1 vendor library. All the experiments presented below focus on asymptotic performance and have been conducted on the maximum amount of cores available on the machine, i.e., 16 cores.

4.2 Tuning

The performance of *tile algorithms* strongly depends on tunable execution parameters of the outer and the inner blocking sizes [5]. The outer block size b trades off parallelization granularity and scheduling flexibility with single core utilization, while the inner block size s trades off memory load with extra-flops due to redundant calculations. Manual tuning based on empirical data has been performed to determine the optimal tile / inner blocking size $(b_{Hbhr}; s_{Hbhr})$ for the BHR algorithm with Householder reflectors and only the tile size b_{Gbhr} for the BHR algorithm with Givens rotations. The same procedure has been repeated for the Full Hessenberg Reduction (FHR) with the parallel bulge chasing procedure. More precisely, only the most compute intensive kernels, i.e., CORE_DTSSSMQR and CORE_DTSSSMGR, have been tuned for different matrix sizes. The tile size for each implementation and different matrix sizes which gives the best performance has been selected as optimal. Therefore, for each matrix size, we actually get different optimal tile sizes (i.e., $b_{Hbhr} \neq b_{Gbhr}$) between both algorithms. For example, for the first stage, the couples $(b_{Hbhr} = 200; s_{Hbhr} = 40)$ and $b_{Gbhr} = 140$ were chosen with Householder reflectors and Givens rotations respectively, to get the asymptotic performance ($n = 12000$). For the second stage, we have considered the parallel bulge chasing procedure only with Householder reflectors, although the same experiments could be done with Givens rotations. SMPSs parallel programming framework has been used to schedule the two-stage approach kernels. The optimal block sizes for the first stage $(b_{Hbhr} = 200; s_{Hbhr} = 40)$ are obviously not optimal for the second stage when looking at asymptotic performance. Clearly, the parallel bulge chasing is the bottleneck in this two-stage approach as mentioned in Section 3.3. So, in order to try to get some parallel performance of the overall algorithm, one should definitely select a different couple $(b_{Hfhr}; s_{Hfhr})$ so that at the end of the first stage, the reduced matrix has a smaller bandwidth. To get the asymptotic performance for the two-stage approach (BHR algorithm followed by the bulge chasing procedure) with Householder reflectors, the selected optimal block size was $(b_{Hfhr} = 100; s_{Hfhr} = 20)$. Finally, the block sizes for LAPACK and ScaLAPACK have also been manually tuned, $b = 32$ and $b = 64$ respectively. The MKL library is a highly optimized library tuned by the vendor.

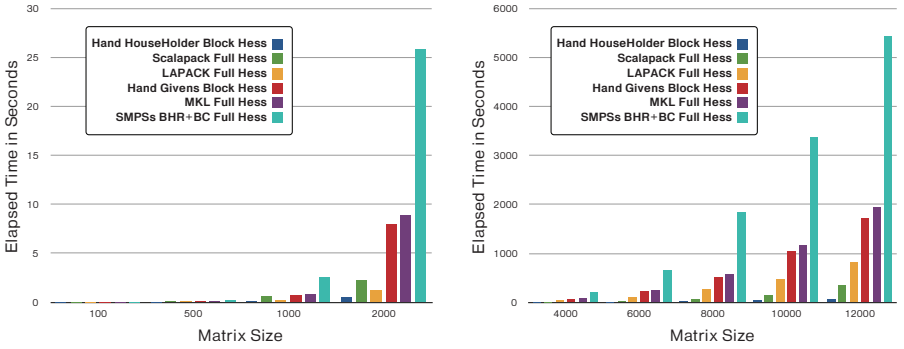
4.3 The BHR Performance Comparisons

The number of floating-point operations used as a reference to compute the performance of both BHR implementations is $10/3\ n\ (n-b)\ (n-b)$. The hand coded dynamic data driven scheduler has been used to run both BHR algorithms in parallel. Figure 7 shows the execution time in seconds for small and large matrix sizes on 16 cores. For a 12000×12000 problem size, the BHR algorithm with Householder reflectors roughly runs 30 times faster than MKL and LAPACK, 15 times faster than ScaLAPACK and, finally, 6 times faster than the BHR implementation with Givens rotation. The authors understand that it may not be a fair comparison against those latter libraries, since the reduction is completely achieved in that case. Indeed, as mentioned in Section 3.4, the remaining reduction from block Hessenberg to full Hessenberg is very expensive. The purpose of showing such performance curves is only to give a rough idea, in terms of elapsed time and performance, of the whole reduction process.

Figure 8 presents the parallel performance in Gflop/s of both BHR algorithms. The BHR algorithm with Householder reflectors scales quite well while the matrix size increases, reaching 95 Gflop/s. It runs at 61% of the system theoretical peak and 72% of the DGEMM peak. The BHR algorithm with Givens rotations is reaching around 30 Gflop/s at $n = 8000$ and starting to decrease for larger matrix sizes. There are mainly three limitations that explain the deficiency of the BHR algorithm with Givens rotations compared to the BHR with Householder reflectors:

- First, the BHR with Givens rotations itself performs 50% more flops compared to the BHR with Householder reflectors. To our knowledge, there exist no inner blocking techniques for Givens rotations, which could decrease the amount of the extra-flops.
- Second, although they are not the most called, the kernels that compute the panel factorization (i.e., CORE_DGEGRG and CORE_DTSGRG) and that explicitly build the orthogonal transformation matrices G are not optimized because they do not rely on high Level BLAS.
- And third, the update kernels (i.e., CORE_DORMGR and CORE_DTSSSMGR) are not performing their operations in-place. Although these kernels are mostly straight calls to Level 3 BLAS, they still need workspaces to compute the corresponding operations and to copy the results back to the right place.

Another point that adds to the above list is the fact that the optimal tile sizes b_{Hbhr} and b_{Gbhr} for a given matrix size are not the same. The selected optimal tile size of the BHR with Householder reflectors b_{Hbhr} for each matrix size is always larger than the one selected for the BHR with Givens rotations. Therefore, the reduced matrices from both algorithms at the end of the first stage do not have the same bandwidth. So, the BHR algorithm with Householder reflectors is actually performing $O((n - b_{Gbhr})^2(b_{Hbhr} - b_{Gbhr}))$ fewer flops.



(a) Small Data Size. (b) Large Data Size.

Fig. 7. Elapsed time in seconds for the Block Hessenberg Reduction on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS 10.1.

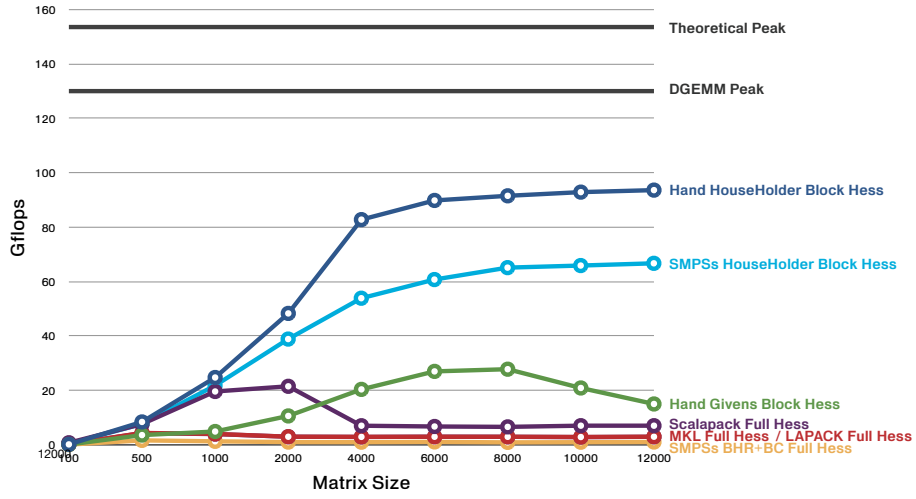


Fig. 8. Parallel Performance Comparisons on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS 10.1.

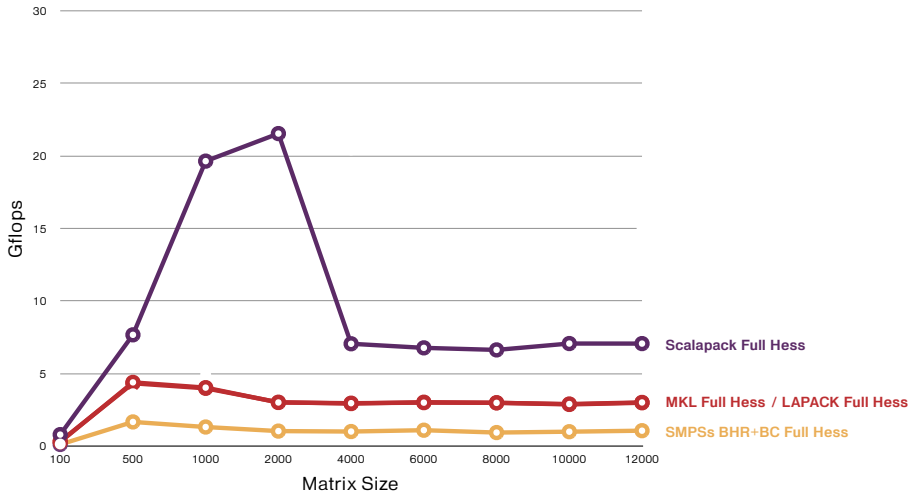


Fig. 9. Zoom-in.

4.4 The FHR Performance Comparisons

Our two-stage Full Hessenberg Reduction (FHR) approach is again composed by the BHR algorithm (first stage) followed by a parallel bulge chasing procedure (second stage). The approach presented here is only based on Householder reflectors. The task scheduling of the whole algorithm relies on SMPSs. The optimal tile sizes of the BHR algorithm for each matrix size have been selected such that the FHR achieves the best performance. Those tile sizes are obviously smaller than those selected when only the first stage is performed. The goal is to try to improve as much as possible the second stage by diminishing the bandwidth. It is a trade-off because now the first stage does not perform very well with smaller tile sizes. By looking at the two blue curves from Figure 8, we clearly see this behavior. The BHR represented by the dark blue curve is optimized for the first stage (large tile sizes) while the BHR represented by the light blue curve is optimized for the entire two-stage approach (small tile sizes). Now, by looking at Figure 9 which zooms into the FHR curves, we clearly see how the second stage negates the overall performance of the two-stage approach for the reasons already explained in Section 3.3. This two-stage approach is also handicapped by the cost of the translation from the original BDL of the first stage to the standard LAPACK layout of the second stage, which takes approximately less than 10% of the elapsed time for large matrix sizes. The authors are also surprised to see the same curves for the FHR of LAPACK and MKL, probably because they have a very similar implementation.

Note that the FHR of ScaLAPACK is twice as fast as the FHR of MKL and LAPACK thanks to the two-dimensional block cyclic distribution.

5 Summary and Future Work

The emergence and continuing use of multicore architectures require changes in the existing software and sometimes even a complete redesign of the established algorithms. This mismatch has been clearly identified by Agullo *et al.* in [5] for one-sided factorizations (Cholesky, QR and LU) on multicore architectures with the state of the art linear algebra libraries. Ltaief *et al.* [22] and this current paper confirm that this mismatch is even more critical for the two-sided transformations.

However, by exploiting the concepts of *tile algorithms* in the multicore environment, i.e., high level of parallelism with fine granularity and high performance data representation combined with a dynamic data driven execution, the BHR algorithm with Householder reflectors (first stage) achieves 72% (95 Gflop/s) of the DGEMM peak on a 12000×12000 matrix size with 16 Intel Tigerton 2.4 GHz cores. This algorithm performs most of the operations in Level 3 BLAS and considerably surpasses in performance the BHR algorithm with Givens rotations. Unfortunately, the FHR algorithm can not achieve any comparable performance, and this is especially due to the inefficiency on multicore architectures of the parallel bulge chasing procedure. This procedure may indeed not suitable for multicores because of the conflict it engenders with the data layout, i.e., standard layout as in LAPACK versus BDL / tile layout.

The purpose of this paper is to show that going from the dense matrix to the block Hessenberg form is really an efficient process on multicore architectures thanks to *tile algorithms*. This block structure might be considered at some point to solve the full non symmetric EVP, either as a pre-processing to go to Hessenberg form or maybe to go directly from block Hessenberg to Schur form. The efficient reduction to full Hessenberg is still an open research issue on multicore architectures. It is also noteworthy to mention another approach followed recently by one of the author to achieve the full reduction using hybrid computations [29], where the matrix-vector product occurring in the panel factorization is off-loaded to the GPU, benefiting from the high bandwidth that such hardware offers.

Finally, this work can be extended to the rest of the family of the two-sided orthogonal transformations, especially the block tri-diagonalization in which the bulge chasing procedure will definitely require fewer flops. The authors are also looking at previous algorithms such as Jacobi methods as well as the potential use of matrix sign functions to achieve the full reduction in the context of *tile algorithms*.

6 Acknowledgment

The authors thank Julien Langou, Alfredo Buttari and the PLASMA team as well as the two anonymous reviewers for their insightful comments, which greatly helped to improve the quality of this article.

References

1. http://www.top500.org.
2. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.
3. SMP Superscalar. http://www.bsc.es/ → Computer Sciences → Programming Models → SMP Superscalar.
4. B. Adlerborn, K. Dackland, and B. Kågström. Parallel two-stage reduction of a regular matrix pair to hessenberg-triangular form. In T. Sørevik, F. Manne, R. Moe, and A. H. Gebremedhin, editors, *PARA*, volume 1947 of *Lecture Notes in Computer Science*, pages 92–102. Springer, 2000.
5. E. Agullo, B. Hadri, H. Ltaief, and J. J. Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. *University of Tennessee Computer Science Technical Report (also LAPACK Working Note 217), Accepted for publication at SuperComputing 2009*.
6. A. A. Anda and H. Park. Fast plane rotations with dynamic scaling. *SIAM J. Matrix Anal. Appl.*, 15:162–174, 1994.
7. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, third edition, 1999.
8. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled qr factorization for multicore architectures. *Concurr. Comput. : Pract. Exper.*, 20(13):1573–1590, 2008.
9. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.
10. J. Choi, J. Demmel, I. Dhillon, J. Dongarra, Ostrouchov, S., A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK, a portable linear algebra library for distributed memory computers-design issues and performance. *Computer Physics Communications*, 97(1-2):1–15, 1996.
11. 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(4):425–454, 1999.
12. J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1–2):215–227, Sept. 1989. (LAPACK Working Note #2).
13. E. Elmroth and F. G. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. In *Applied Parallel Computing, Large Scale Scientific and Industrial Problems, 4th International Workshop, PARA'98*, Umeå, Sweden, June 14-17 1998. Lecture Notes in Computer Science 1541:120-128.
14. E. Elmroth and F. G. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. & Dev.*, 44(4):605–624, 2000.

15. E. Elmroth and F. G. Gustavson. High-performance library software for QR factorization. In *Applied Parallel Computing, New Paradigms for HPC in Industry and Academia, 5th International Workshop, PARA 2000*, Bergen, Norway, June 18-20 2000. Lecture Notes in Computer Science 1947:53-63. http://dx.doi.org/10.1007/3-540-70734-4_9.
16. W. M. Gentleman. Least squares computations by Givens transformations without square roots. *J. Inst. Math. Appl.*, 12:329–336, 1973.
17. P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders. Methods for modifying matrix factorizations. *Math. Comp.*, 28:505–535, 1974.
18. G. H. Golub and C. F. Van Loan. *Matrix Computation*. John Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.
19. B. C. Gunter and R. A. van de Geijn. Parallel out-of-core computation and updating of the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, Mar. 2005.
20. S. Hammarling. A note on modifications to the Givens plane rotations. *J. Inst. Maths Applics*, 13:215–218, 1974.
21. B. Kågström, D. Kressner, E. Quintana-Orti, and G. Quintana-Orti. Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisited. Technical report, 2008.
22. H. Ltaief, J. Kurzak, and J. Dongarra. Parallel band two-sided matrix bidiagonalization for multicore architectures. *University of Tennessee Computer Science Technical Report, UT-CS-08-631 (also LAPACK Working Note 209), Accepted in IEEE Transactions on Parallel and Distributed Systems*.
23. E. S. Quintana-Ortí and R. A. van de Geijn. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software*, 35(2), July 2008.
24. G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. A. van de Geijn, and F. G. Van Zee. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *PDP*, pages 301–310. IEEE Computer Society, 2008.
25. W. Rath. Fast Givens rotations for orthogonal similarity transformations. *Numer. Math.*, 40:47–56, 1982.
26. R. Schreiber and C. Van Loan. A storage efficient WY representation for products of householder transformations. *SIAM J. Sci. Statist. Comput.*, 10:53–57, 1989.
27. G. W. Stewart. The economical storage of plane rotations. *Numerische Mathematik*, 25:137–138, 1976.
28. G. W. Stewart. *Matrix Algorithms Volume I: Matrix Decompositions*. SIAM, Philadelphia, 1998.
29. S. Tomov and J. Dongarra. Accelerating the reduction to upper hessenberg form through hybrid gpu-based computing. *University of Tennessee Computer Science Technical Report (also LAPACK Working Note 219), Submitted to Journal of Parallel Computing*.
30. L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.
31. E. L. Yip. Fortran subroutines for out-of-core solutions of large complex linear systems. *Technical Report CR-159142, NASA*, November 1979.