Parallel Block Hessenberg Reduction using Algorithms-By-Tiles for Multicore Architectures Revisited

Ltaief, Hatem and Kurzak, Jakub and Dongarra, Jack

2009

MIMS EPrint: **2009.6**

ISSN 1749-9097

Parallel Block Hessenberg Reduction using Algorithms-By-Tiles for Multicore Architectures Revisited *LAPACK Working Note #208*

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. The objective of this paper is to extend and redesign the block matrix reduction applied for the family of two-sided factorizations, introduced by Dongarra *et al.* [9], to the context of multicore architectures using *algorithms-by-tiles*. In particular, the Block 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 commonly used for this reduction because they guarantee stability, as opposed to Gaussian Elimination. Two versions of the Block Hessenberg Reduction are presented in this paper, 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. Furthermore, in the last Top500 list from June 2008, 98% of the fastest parallel systems in the world are based on multicores. The emerging petascale systems consisting of hundreds of thousands of cores have exacerbated the problem even more and it becomes judicious to efficiently integrate existing or new numerical linear algebra algorithms suitable for such hardwares. By exploiting the concepts of *algorithms-by-tiles* 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 Block Hessenberg Reduction presented here achieves 72% of the DGEMM peak on a 12000×12000 matrix with 16 Intel Tigerton 2.4 GHz processors.

1 Introduction

The objective of this paper is to extend and redesign the block matrix reduction applied for the family of two-sided factorizations, e.g., Hessenberg, Tridiagonalization, Bi-diagonalization, introduced by Dongarra *et al.* [9], to the

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

context of multicore architectures using *algorithms-by-tiles*. In particular, Hessenberg Reduction (HR) is very often used as a pre-processing step in solving dense linear algebra problems, such as the standard EigenValue Problem (EVP) [12]:

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

The need to solve EVPs emerges from various computational science disciplines e.g., structural engineering, electronic structure calculations, computational fluid dynamics, and also, in information technology e.g., search engines rank websites [14]. The basic idea is to transform the dense matrix A to an upper Hessenberg form H by applying successive transformations from the left (Q) as well as from the right (Q^T) as follows:

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

In this paper, we will look only at the first stage of HR, which goes from the original dense matrix A to a block Hessenberg matrix H_b with b being the number of sub-diagonals. The second stage, which annihilates those additional b sub-diagonals, is not examined in this paper but will appear in a companion one. This *two-stage* transformation process is also explained by Kagstrom *et. al* in [15] for the QZ algorithm. 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 [22]. Two versions of the Block Hessenberg Reduction (BHR) algorithms 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 because of a predictable bad impact on parallel performance. The main reasons will be pointed out later in the paper.

Furthermore, in the last Top500 list from June 2008 [1], 98% of the fastest parallel systems in the world are based on multicores. The emerging petascale systems consisting of hundreds of thousands of cores have exacerbated the problem even more and it becomes judicious to efficiently integrate existing or new numerical linear algebra algorithms suitable for such hardwares. As discussed by Buttari *et al.* in [7], a combination of several parameters define the concept of *algorithms-by-tiles* 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 enqueued 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. On the other hand, previous work done by Kurzak *et. al* show how the characteristics of tiled algorithms perfectly match the architectural features of the high performance Cell Broadband Engine processor [16, 17].

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 implementation of the parallel tiled BHR algorithm. Section 4 presents performance results of the two versions. Also, comparison tests are run on shared-memory architectures against the state of the art, high performance dense linear algebra software libraries, LAPACK [4] and ScaLAPACK [8]. Section 5 gives a detailed overview of previous projects in this area. Finally, section 6 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 follows:

Algorithm 1 Hessenberg Reduction with Householder reflectors

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

Algorithm 1 takes as input the dense matrix A and gives as output the matrix in Hessenberg form. The reflectors v_j 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 because it is based on matrix-vector operations. Also, a single entire 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 tiles instead in order to improve data locality and cache reuse and to generate as much as possible matrix-matrix operations .

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:

$$g(i,j,\theta) \;=\; \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & \text{——} & \text{——} & s & \\ & & | & 1 & & | & \\ & & | & \ddots & & | & \\ & & | & & 1 & | & \\ & & -s & \text{——} & \text{——} & c & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{pmatrix}, \qquad (1)$$

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

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

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

The multiplication $g(i,j,\theta)^T \times x$ 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., $x \times g(j,i,\theta)$, 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. And this does not include the cost of accumulating the *local* Givens matrices in line 5. Thus,

Algorithm 2 Hessenberg Reduction with Givens rotations

```
1: $G \Leftarrow Id_n$
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:     Accumulate $G = g(i-1,i) \times G$
6:     Update $A = g^T(i-1,i) \times A$
7:     Update $A = A \times g(i,i-1)$
8:   end for
9: end for
```

although their implementations are much simpler than Householder reflectors, Givens rotations are expected to poorly perform for the HR algorithm, if used as in algorithm 2, due to the predominance of vector-vector operations and to the overhead of cache misses. But, as seen later in section 3, some of those limitations can be overcome, however the whole algorithm can still benefit from the concepts of *algorithms-by-tiles*.

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

2.3 Discussion on Fast Givens Rotations

The main problem with standard Givens rotations is the two additional flops needed to annihilate one element of the matrix.

First introduced by Gentleman [10] and then by Hammarling [13] for the QR-decomposition, FGRs are interesting because it 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 [19] applied FGRs 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 [3] introduced the Self-Scaling Chained FGRs, 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:

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

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 three bottom left elements 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 FGR matrix. F has this typical simplified form:

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

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

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

A bookkeeping procedure has to be established to save each specific form of the chosen matrices F during the reduction. Unfortunately, this prevents an automatic construction of the global matrix G as implemented in algorithm 2 (line 5) for the standard HR based on Givens rotations. Therefore, this presents a major complexity for efficiently accumulating FGR matrices.

On the other hand, the achievement of 4 flops per zeroed element is done by using *chained* FGRs. 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 equation (3) into equation (4), we end up with the following equation:

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

The component $\hat{y}_i$ has to be computed first and then its value is used to obtain the final result for the component $\hat{y}_j$. While this is suitable for vector processors, it is expected to perform poorly in cache-based architectures due to memory overheads. Thus, even though more expensive, standard Givens rotations are preferred to FGRs.

In the next section, we explain the modifications applied in algorithms 1 and 2 to achieve high performance using the concepts of *algorithms-by-tiles*.

3 The Parallel Tile BHR Algorithm

We present two parallel implementations of the BHR based on Householder reflectors and Givens rotations. 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 is doing $O(n^2\ b)$ less flops, which is a negligible expense of the overall HR algorithm cost. By using updating factorization techniques as suggested in [12, 5], the kernels for both implementations can be applied to square blocks or tiles of the original matrix. Also, thanks to fine granularity and BDL, the memory access management is significantly ameliorated.

3.1 Description of the Fast Orthogonal Transformation Kernels

Householder reflectors

The tiled BHR kernels based on Householder reflectors are identical to the ones used by Buttari *et. al* in [7] for the QR factorization. Basically, DGEQRT is used to do a QR blocked factorization using the WY technique for efficiently accumulating the Householder reflectors [20]. The DLARFB kernel comes from the LAPACK distribution and is used to apply a block of Householder reflectors. DTSQRT performs a block QR factorization of a matrix composed of two tiles, a triangular tile on top of a dense square tile. DSSRFB updates the matrix formed by coupling two square blocks and applying the resulting DTSQRT transformations. [7] gives a detailed description of the different kernels. However, minor modifications are needed for the DLARFB and DSSRFB kernels in order to apply the updates on the right side. Moreover, since the right orthogonal transformations do not destroy the zeroed structure and do not bring any fill-in elements, the computed reflectors can be stored in the lower annihilated part of the original matrix for later use. Although the algorithms work for rectangular matrices, for simplicity purposes, let us only focus on square matrices. Let NBT be the number of tiles in each direction. The tiled BHR algorithm with Householder reflectors then appears as follows:
The characters "L" and "R" stand for Left and Right updates. In each kernel call, the triplets (i, ii, iii) specify the tile location in the original matrix, as in figure 1: (i) corresponds to the reduction step in the general algorithm, (ii) gives the row index and (iii) represents the indice of the column. For example, in figure 1(a), the blue tiles represent the final data tiles, the white tiles are the zeroed tiles, the gray tiles are those which need to be processed and finally, the black tile corresponds to DTSQRT(1,4,1). In figure 1(b), the top black tile is DLARFB("R",3,1,4) while the bottom one is DLARFB("L",3,4,5).
These kernels are very rich in matrix-matrix operations. By working on small tiles with BDL, the elements are stored contiguous in memory and thus the access pattern to memory is more regular, which makes these kernels high performant.

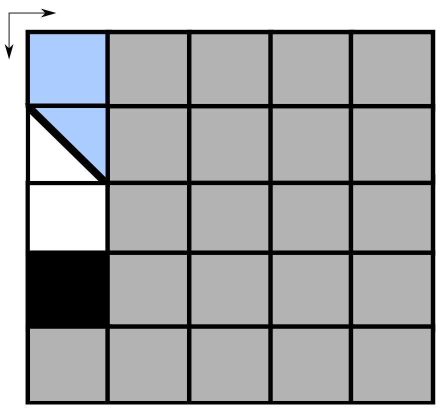
Algorithm 3 Tiled BHR Algorithm with Householder reflectors.

```
1: for $i = 1, 2$ to NBT$-1$ do
2:     DGEQRT($i,\ i+1,\ i$)
3:     for $j = i+1$ to NBT do
4:         DLARFB(”$L$”, $i,\ i+1\ ,j$)
5:     end for
6:     for $j = 1$ to NBT do
7:         DLARFB(”$R$”, $i,\ j,\ i+1$)
8:     end for
9:     for $k = i+2$ to NBT do
10:        DTSQRT($i,\ k,\ i$)
11:        for $j = i+1$ to NBT do
12:            DSSRFB(”$L$”, $i,\ k,\ j$)
13:        end for
14:        for $j = 1$ to NBT do
15:            DSSRFB(”$R$”, $i,\ j,\ k$)
16:        end for
17:    end for
18: end for
```

In the following subsection, 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. The first factorization kernel named DGEQRG is implemented to produce a block QR factorization applied on a square tile of size b with Givens rotations. The goal is obviously to find a way to overcome the overhead of the accumulation of the *local* Givens matrices g(i,j) in order to generate matrix-matrix operations for the left and the right updates. By using a recursive formula, as explained by Gill *et. al* in [11], the accumulation of the Givens matrices does not present a limitation anymore and its cost becomes almost negligible. While the first column is being annihilated, the *local* Givens matrices are accumulated and then they are applied at once to the rest of the tile, as is done for DGEQRT with Householder reflectors. As the elimination procedure takes place, a larger Givens matrix is built that corresponds to the product of all *local* Givens matrices g(i,j), again by using the technique described in [11]. Finally, DGEQRG gives as output the upper triangular tile as well as a dense Givens rotation matrix G_b, used later for the updates. The components c and s of the Givens matrix can also be saved in a single element (see Stewart in [21]) and stored in the lower zeroes part of the tile. The DGEMM kernels in algorithm 4, lines (4, 7) perform the product of G_b by the tile specified in the triplets, from left and right respectively. The


(a) BHR: Reduction Step 1.

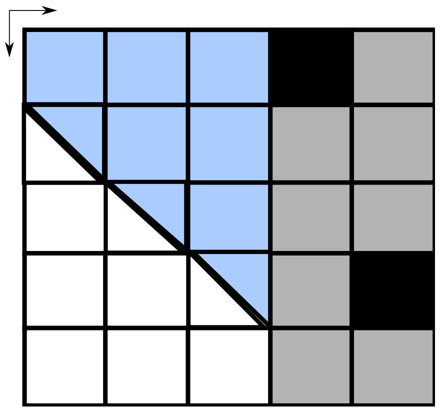

(b) BHR: Reduction Step 3.

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

second factorization called DTSQRG applies a blocked QR factorization on a matrix of size $2b \times b$ composed of a triangular tile on top of a square tile. This kernel inherits the properties of the annihilation procedure described above for the DGEQRG kernel, by using the top upper triangular tile as a reference. As a matter of fact, DTSQRG yields an upper triangular matrix of size $2b \times b$ and a Givens rotation square matrix $G_{2\times b}$, which has the shape depicted in figure 2. The updates, either left or right lines (12, 15), are straightforward; a call to DTRMM for the lower triangular structure followed by a call to DGEMM for the upper square structure achieves the desired transformations.

Figure 3 and 4 illustrate the step-by-step execution of algorithms 3 and 4, with Householder reflectors and Givens rotations, respectively, in order to eliminate the first tile column. It appears necessary then to efficiently schedule the kernels to get high performance in parallel.

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

3.2 Dynamic Data Driven Execution

A dynamic scheduling scheme similar to [7] has been extended for the two-sided orthogonal transformations. A Directed Acyclic Graph (DAG) is used to represent the data flow between the nodes/kernels. While the DAG is quite easy to draw for 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

Algorithm 4 Tiled BHR Algorithm with Givens rotations.

```
for $i = 1, 2$ to NBT$-1$ do
    DGEQRG($i,\ i+1,\ i$)
    for $j = i+1$ to NBT do
        DGEMM($i,\ i+1\ ,j$)
    end for
    for $j = 1$ to NBT do
        DGEMM($i,\ j,\ i+1$)
    end for
    for $k = i+2$ to NBT do
        DTSQRG($i,\ k,\ i$)
        for $j = i+1$ to NBT do
            DTRMM_DGEMM($i,\ k,\ j$)
        end for
        for $j = 1$ to NBT do
            DTRMM_DGEMM($i,\ j,\ k$)
        end for
    end for
end for
```

edges. The implementation of the scheduler has to be lightweight to minimize the overhead on the overall application performance. Also, the dynamic scheduler allows an out-of-order execution, and the idle time is very limited since there are no global synchronization points between the threads. Figure 5 shows the tracing of the dynamic data driven scheduler using eight cores, performing in that particular experiment the BHR with Householder reflectors. The six different kernels are clearly identified with their colors.

In the next section, we present the experimental results comparing our two BHR implementations with the state of the art library, i.e. LAPACK [4], ScaLAPACK [8] and MKL $V10$ [2].

4 Experimental Results

The experiments have been applied on two different shared memory Platforms: (P1) a dual-socket quad-core Intel Itanium 2 1.6 GHz (eight total cores) with 16GB of memory, and (P2) a quad-socket quad-core Intel Tigerton 2.4 GHz (16 total cores) with 32GB of memory. Hand tuning based on empirical data has been performed to determine the optimal block/tile size b for both implementations: 200 and 140 for the BHR algorithms with Householder reflectors and Givens rotations were chosen, respectively.

Figure 6 shows the elapsed time in seconds for small and large matrix sizes on (P1) with eight cores. The BHR algorithm based on Householder reflectors by far outperforms the others: for a 12000×12000 problem size, it runs approximately

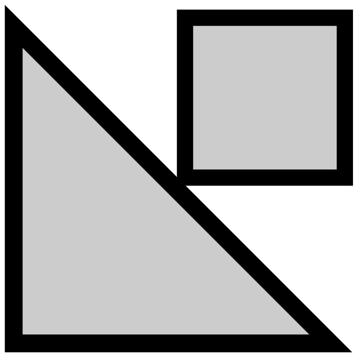


Fig. 2. Givens rotation matrix produced by DTSQRG used during the update procedures.

10 x faster than the full HR of ScaLAPACK, 8 x faster than the full HR of MKL and LAPACK and 3 x faster than the BHR with Givens rotations. Figure 7(a) presents the parallel performance in Gflops of the BHR algorithms on (P1). The BHR implementation with Householder reflectors runs at 82% of the machine theoretical peak of the system and at 92% of the DGEMM peak. Figure 7(b) zooms in the four other implementations and the parallel performance of the BHR with Givens rotations is significantly higher than the full HR of LAPACK, ScaLAPACK and MKL.

The same experiments have been conducted on (P2) with 16 cores. Figure 8 shows the execution time in seconds for small and large matrix sizes. For a 12000×12000 problem size, the BHR algorithm with Householder reflectors roughly runs 30 x faster than MKL and LAPACK, 15 x faster than ScaLAPACK and finally, 6 x faster than the BHR implementation with Givens rotation. Figure 9(a) presents the parallel performance in Gflops of the BHR algorithms. The BHR algorithm with Householder reflectors scales quite well while the matrix size increases, reaching 95 Gflops. It runs at 61% of the system theoretical peak and 72% of the DGEMM peak. The zoom-in seen in figure 9(b) highlights the weaknesses of the BHR algorithm with Givens rotations, i.e. the algorithm itself (50% more flops compared to Householder reflectors) and the non-optimized hand-coded factorization kernels. But still, the BHR algorithm with Givens rotations outperforms the full HR of MKL, LAPACK and ScaLAPACK. Note: the full HR of ScaLAPACK is twice as fast as than the full HR of MKL and LAPACK thanks to the Two-dimensional Block Cyclic Distribution.

The following section briefly comments on the previous work done in HR using two-sided orthogonal transformations.

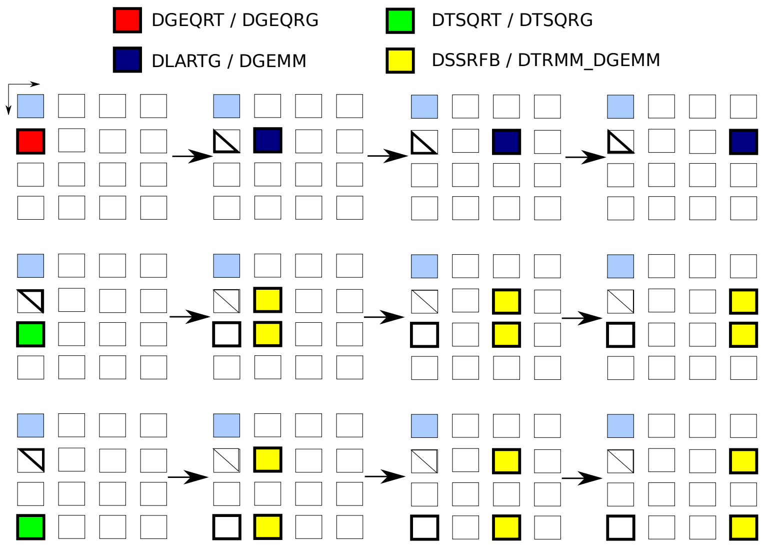


Fig. 3. Scheduling of the Left Orthogonal Transformation.

5 Related Work

In [18], the authors improved the performance of the HR algorithm with Householder reflectors included in LAPACK by shifting matrix-vector operations to more efficient matrix-matrix operations using a new blocked algorithm. By efficiently accumulating the reflectors and updating only the necessary blocks while the reduction takes place, their algorithm achieves more computation in BLAS 3 than the LAPACK HR implementation and improves data locality while reducing cache traffic. However, the parallelism is only expressed in the multithreaded BLAS using the fork-join approach, which definitely presents overhead due to thread synchronizations.

The authors in [15] describe a sequential HR implementation in the context of the QZ algorithm, based only on Givens rotations, which outperforms LAPACK. By working on *diamond* shaped tiles and regrouping disjoint Givens rotation sequences, their algorithm performs most of the computation in matrix-matrix multiplications with increased data locality.

The authors in [6] describe a parallel BHR on distributed memory systems using a two-dimensional mesh of processors. The factorization is done using a binary tree approach, where multiple column blocks can be reduced at the same time, which can ameliorate the overall parallel performance. Their implementation is hybrid, i.e., it combines Householder reflectors with Givens rotations. While the reduction of the column blocks to triangular form is done using Householder reflectors, those triangular column blocks are then annihilated using Givens rotations for sparsity purposes. Knowing the overhead of Givens ro-

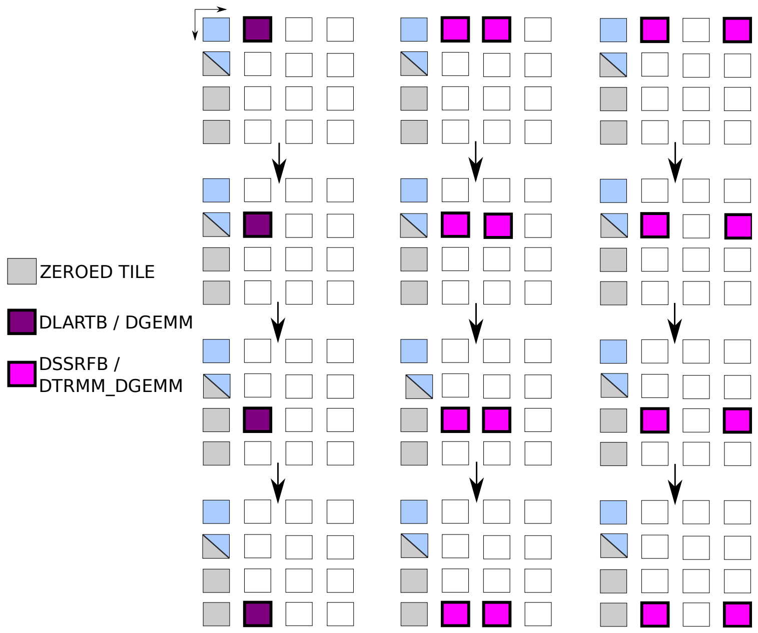


Fig. 4. Scheduling of the Right Orthogonal Transformation.

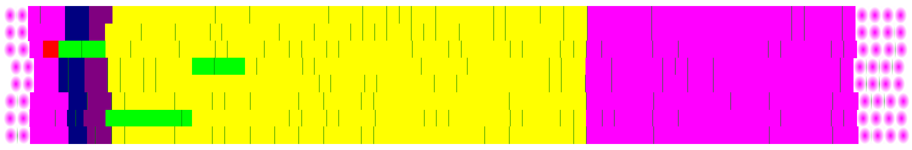
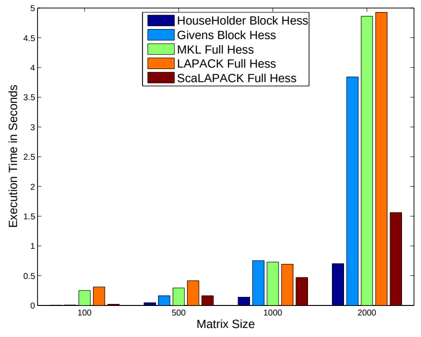


Fig. 5. Tracing of Dynamic Data Driven Execution with 8 cores.

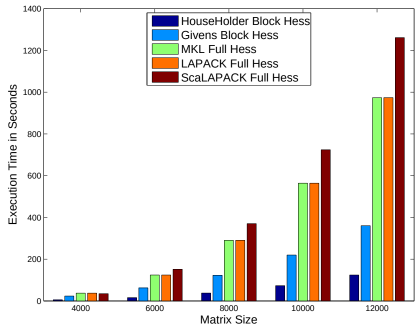
tations over the Householder reflectors, reducing all blocks only by Householder reflectors is probably more suitable.

6 Summary and Future Work

By exploiting the concepts of *algorithms-by-tiles* 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 achieves 72% (95 Gflops) 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 and the full HR with MKL, LAPACK and ScaLAPACK. This work can be extended to the rest of the family of the two-sided orthogonal transformations, i.e., block tri-

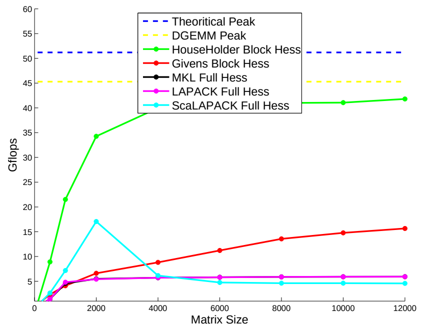


(a) Small Data Size.

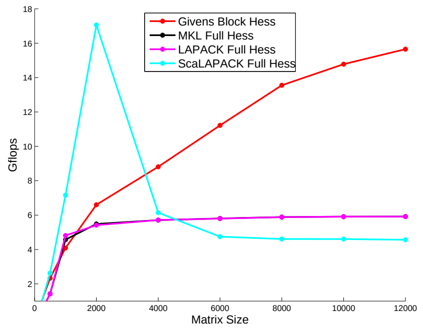


(b) Large Data Size.

Fig. 6. Elapsed time in seconds for the Block Hessenberg Reduction on a dual-socket quad-core Intel Itanium2 1.6 GHz with MKL BLAS V10.0.1.



(a) Performance comparisons.



(b) Zoom-in.

Fig. 7. Parallel Performance of the Block Hessenberg Reduction on a dual-socket quad-core Intel Itanium2 1.6 GHz processors with MKL BLAS V10.0.1.

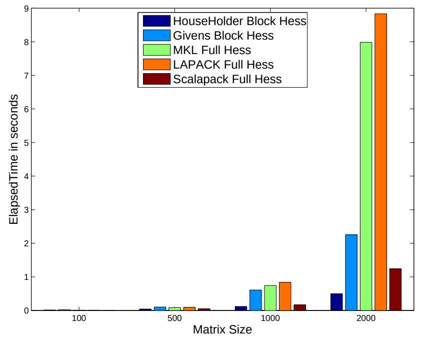
diagonalization and block bi-diagonalization reductions. These techniques can also be applied to the regular generalized EVP.

7 Acknowledgment

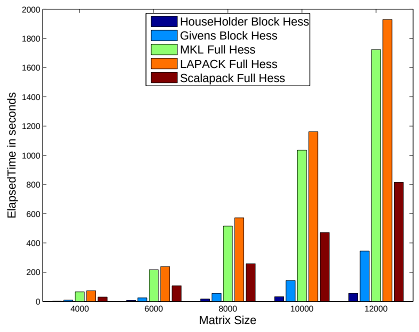
The authors thank Julien Langou and Alfredo Buttari for their insightful comments, which greatly helped to improve the quality of this article.

References

1. http://www.top500.org.
2. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.
3. A. A. Anda and H. Park. Fast plane rotations with dynamic scaling. *SIAM J. Matrix Anal. Appl.*, 15:162–174, 1994.

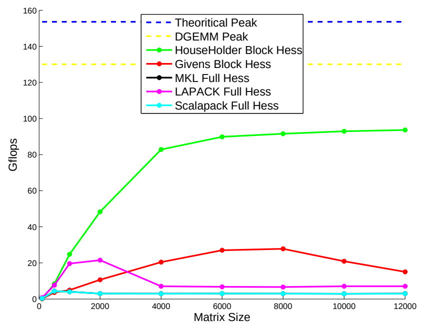


(a) Small Data Size.

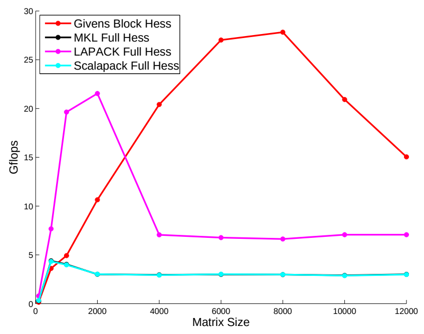


(b) Large Data Size.

Fig. 8. Elapsed time in seconds for the Block Hessenberg Reduction on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS V10.0.1.



(a) Performance comparisons.



(b) Zoom-in.

Fig. 9. Parallel Performance of the Block Hessenberg Reduction on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS V10.0.1.

4. 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.
5. M. Baboulin, L. Giraud, S. Gratton, and J. Langou. Parallel tools for solving incremental dense least squares problems. application to space geodesy. *To appear in Journal of Algorithms and Computational Technology*, 2(3), 2008.
6. M. W. Berry, J. J. Dongarra, and Y. Kim. A highly parallel algorithm for the reduction of a nonsymmetric matrix to block upper-Hessenberg form. LAPACK Working Note 68, Department of Computer Science, University of Tennessee, Knoxville, inst-UT-CS:adr, feb 1994. UT-CS-94-221, February 1994.
7. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel Tiled QR Factorization for Multicore Architectures. LAPACK Working Note 191, July 2007.
8. 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.

9. 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).
10. W. M. Gentleman. Least squares computations by Givens transformations without square roots. *J. Inst. Math. Appl.*, 12:329–336, 1973.
11. P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders. Methods for modifying matrix factorizations. *Math. Comp.*, 28:505–535, 1974.
12. 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.
13. S. Hammarling. A note on modifications to the Givens plane rotations. *J. Inst. Maths Applics*, 13:215–218, 1974.
14. T. H. Haveliwala and A. D. Kamvar. The second eigenvalue of the google matrix. Technical report, Stanford University, Apr. 18 2003.
15. B. Kagstrom, D. Kressner, E. Quintana-Orti, and G. Quintana-Orti. Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisted. LAPACK Working Note 198, February 2008.
16. J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equations on the CELL processor using Cholesky factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1–11, Sept. 2008.
17. J. Kurzak and J. Dongarra. QR Factorization for the CELL Processor. LAPACK Working Note 201, May 2008.
18. G. Quintana-Ortí and R. A. van de Geijn. Improving the performance of reduction to Hessenberg form. *ACM Trans. Math. Softw*, 32(2):180–194, 2006.
19. W. Rath. Fast Givens rotations for orthogonal similarity transformations. *Numer. Math.*, 40:47–56, 1982.
20. 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.
21. G. W. Stewart. The economical storage of plane rotations. *Numerische Mathematik*, 25:137–138, 1976.
22. L. N. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, Philadelphia, PA, 1997.