

# Scheduling Two-sided Transformations using Tile Algorithms on Multicore Architectures

**Hatem Ltaief**

Department of Electrical Engineering and Computer Science,  
University of Tennessee, USA

**Jakub Kurzak**

Department of Electrical Engineering and Computer Science,  
University of Tennessee, USA

**Jack Dongarra<sup>1</sup>**

Department of Electrical Engineering and Computer Science,  
University of Tennessee, USA

Computer Science and Mathematics Division,

Oak Ridge National Laboratory, USA

School of Mathematics & School of Computer Science,

University of Manchester, United Kingdom

**Rosa M. Badia**

Barcelona Supercomputing Center - Centro Nacional de Supercomputación

Consejo Nacional de Investigaciones Cientificas, Spain

---

<sup>1</sup>Corresponding author.

## ABSTRACT

The objective of this paper is to describe, in the context of multicore architectures, three different scheduler implementations for the two-sided linear algebra transformations, in particular the Hessenberg and Bidiagonal reductions which are the first steps for the standard eigenvalue problems and the singular value decompositions respectively. State-of-the-art dense linear algebra softwares, such as the LAPACK and ScaLAPACK libraries, suffer performance losses on multicore processors due to their inability to fully exploit thread-level parallelism. At the same time the fine-grain dataflow model gains popularity as a paradigm for programming multicore architectures. Buttari *et al.* [2007] introduced the concept of *tile algorithms* in which parallelism is no longer hidden inside Basic Linear Algebra Subprograms but is brought to the fore to yield much better performance. Along with efficient scheduling mechanisms for data-driven execution, these *tile* two-sided reductions achieve high performance computing by reaching up to 75% of the DGEMM peak on a  $12000 \times 12000$  matrix with 16 Intel Tigerton 2.4 GHz processors. The main drawback of the *tile algorithms* approach for two-sided transformations is that the full reduction can not be obtained in one stage. Other methods have to be considered to further reduce the band matrices to the required forms.

**KEYWORDS:** scheduling, multicore, linear algebra, matrix factorization, two-sided transformations

## 1 Introduction

The current trend in the semiconductor industry to double the number of execution units on a single die is commonly referred to as *the multicore discontinuity*. This term reflects the fact that existing software model is inadequate for the new architectures and existing code base will be incapable of delivering increased performance, possibly not even capable of sustaining current performance.

This problem has already been observed with state-of-the-art dense linear algebra libraries, LAPACK [5] and ScaLAPACK [13], which deliver a small fraction of peak performance on current multicore processors and multi-socket systems of multicore processors. Most of the algorithms implemented within both softwares can be described as the repetition of two fundamental steps:

- panel factorization : depending of the Linear Algebra operation that has to be performed, a number of transformations are computed for a small portion of the matrix (the so called panel). These transformations, computed by means of Level-2 BLAS operations, can be accumulated for efficient later reuse.
- trailing submatrix update : in this step, all the transformations that

have been accumulated during the panel factorization, can be applied at once to the rest of the matrix (i.e. the trailing submatrix) by means of Level-3 BLAS operations.

In fact, the parallelism in those frameworks is only expressed at the level of BLAS which follows the principles of the expensive *fork-join* approach. Substantially, both LAPACK and ScaLAPACK implement sequential algorithms that rely on parallel building blocks (i.e., the BLAS operations). As multi-core systems require finer granularity and higher asynchronicity, considerable advantages may be obtained by reformulating old algorithms or developing new algorithms in a way that their implementation can be easily mapped on these new architectures.

Buttari *et al.* [2007] introduced the concept of *tile algorithms* in which parallelism is no longer hidden inside Basic Linear Algebra Subprograms but is brought to the fore to yield much better performance. Operations in the standard LAPACK algorithms for some common factorizations are then broken into sequences of smaller tasks in order to achieve finer granularity and higher flexibility in the scheduling of tasks to cores.

This paper presents different scheduling schemes using *tile algorithms* for the two-sided linear algebra transformations, in particular the Hessenberg and Bidiagonal reductions (HRD and BRD).

- The HRD is very often used as a pre-processing step in solving the

standard eigenvalue problems (EVP) [19]:

$$(A - \lambda I) x = 0,$$

$$\text{with } A \in \mathbb{R}^{n \times n}, x \in \mathbb{C}^n, \lambda \in \mathbb{C}.$$

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 orthogonal transformations from the left ( $Q$ ) as well as from the right ( $Q^T$ ) as follows:

$$H = Q \times A \times Q^T,$$

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

- The BRD of a general, dense matrix is very often used as a pre-processing step for calculating the singular value decompositions (SVD) [19, 36]:

$$A = X \Sigma Y^T,$$

$$\text{with } A \in \mathbb{R}^{m \times n}, X \in \mathbb{R}^{m \times m}, \Sigma \in \mathbb{R}^{m \times n}, Y \in \mathbb{R}^{n \times n}.$$

The necessity of calculating SVDs emerges from various computational

science disciplines, e.g., in statistics where it is related to principal component analysis, in signal processing and pattern recognition, and also in numerical weather prediction [14]. The basic idea is to transform the dense matrix  $A$  to an upper bidiagonal form  $B$  by applying successive distinct orthogonal transformations from the left ( $U$ ) as well as from the right ( $V$ ) as follows:

$$B = U^T \times A \times V,$$

$$A \in \mathbb{R}^{n \times n}, U \in \mathbb{R}^{n \times n}, V \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times n}.$$

As originally discussed in [10] for one-sided transformations, the *tile algorithms* approach is a combination of several concepts which are essential to match the architecture associated with the cores: (1) fine granularity to reach a high level of parallelism and to fit the cores' 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) data-driven scheduler to ensure any enqueued tasks can immediately be processed as soon as all their data dependencies are satisfied.

By using those concepts along with efficient scheduler implementations for data-driven execution, these two-sided reductions achieve high performance computing. However, the main drawback of the *tile algorithms* approach for two-sided transformations is that the full reduction can not be obtained in one stage. Other methods have to be considered to further reduce the band

matrices to the required forms. A section in this paper will address the origin of this issue.

The remainder of this paper is organized as follows: Section 2 recalls the standard HRD and BRD algorithms. Section 3 describes the parallel HRD and BRD tile algorithms. Section 4 outlines the different scheduling schemes. Section 5 presents performance results for each implementation. Section 6 gives a detailed overview of previous projects in this area. Finally, section 7 summarizes the results of this paper and presents the ongoing work.

## 2 Description of the two-sided transformations

In this section, we review the original HRD and BRD algorithms using orthogonal transformations based on Householder reflectors.

### 2.1 The Standard Hessenberg Reduction

The standard HRD 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  $v_j$  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. Four flops

---

**Algorithm 1** Hessenberg Reduction with Householder reflectors

---

```
1: for  $j = 1$  to  $n - 2$  do  
2:    $x = A_{j+1:n,j}$   
3:    $v_j = \text{sign}(x_1) \|x\|_2 e_1 + x$   
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
```

---

are needed to update one element of the matrix. The number of operations required by the left transformation (line 5) is then (the lower order terms are neglected):

$$\begin{aligned} \sum_{j=1}^n 4(n-j)^2 &= 4\left(\sum_{j=1}^n n^2 - 2n \sum_{j=1}^n j + \sum_{j=1}^n j^2\right) \\ &\simeq 4\left(n^3 - n^3 + \frac{2n^3}{6}\right) \\ &= \frac{4}{3}n^3. \end{aligned}$$

Similarly, the number of operations required by the right transformation (line 6) is then:

$$\begin{aligned} \sum_{j=1}^n 4n(n-j) &= 4n\left(\sum_{j=1}^n n - \sum_{j=1}^n j\right) \\ &\simeq 4n\left(n^2 - \frac{n^2}{2}\right) \\ &= 2n^3. \end{aligned}$$



The total number of operations for such algorithm is finally  $4/3n^3 + 2n^3 = 10/3n^3$ .

## 2.2 The Standard Bidiagonal Reduction

The standard BRD algorithm based on Householder reflectors interleaves two factorizations methods, i.e. QR (left reduction) and LQ (right reduction) decompositions. The two phases are written as follows:

---

**Algorithm 2** Bidiagonal Reduction with Householder reflectors

---

```

1: for  $j = 1$  to  $n$  do
2:    $x = A_{j:n,j}$ 
3:    $u_j = \text{sign}(x_1) \|x\|_2 e_1 + x$ 
4:    $u_j = u_j / \|u_j\|_2$ 
5:    $A_{j:n,j:n} = A_{j:n,j:n} - 2 u_j (u_j^* A_{j:n,j:n})$ 
6:   if  $j < n$  then
7:      $x = A_{j,j+1:n}$ 
8:      $v_j = \text{sign}(x_1) \|x\|_2 e_1 + x$ 
9:      $v_j = v_j / \|v_j\|_2$ 
10:     $A_{j:n,j+1:n} = A_{j:n,j+1:n} - 2 (A_{j:n,j+1:n} v_j) v_j^*$ 
11:   end if
12: end for

```

---

Algorithm 2 takes as input a dense matrix  $A$  and gives as output the upper bidiagonal decomposition. The reflectors  $u_j$  and  $v_j$  can be saved in the lower and upper parts of  $A$ , respectively, for storage purposes and used later if necessary. The bulk of the computation is located in line 5 and in line 10 in which the reflectors are applied to  $A$  from the left and then from the right, respectively. Four flops are needed to update one element of the matrix. The left transformations (line 5) is exactly the same than the HRD algorithm

and thus, the number of operations required, as explained in (1), is  $4/3 n^3$  (the lower order terms are neglected). The right transformation (line 10) is actually the transpose of the left transformation and requires the same amount of operations, i.e.,  $4/3 n^3$ . The overall number of operations for such algorithm is finally  $8/3 n^3$ .

### **2.3 The LAPACK Block Algorithms**

The algorithms implemented in LAPACK leverage the idea of blocking to limit the amount of bus traffic in favor of a high reuse of the data that is present in the higher level memories which are also the fastest ones. The idea of blocking revolves around an important property of Level-3 BLAS operations, the so called surface-to-volume property, that states that  $O(n^3)$  floating point operations are performed on  $O(n^2)$  data. Because of this property, Level-3 BLAS operations can be implemented in such a way that data movement is limited and reuse of data in the cache is maximized. Blocking algorithms consists in recasting Linear Algebra algorithms in a way that only a negligible part of computations is done in Level-2 BLAS operations (where no data reuse possible) while the most part is done in Level-3 BLAS.

### **2.4 Limitations of the Standard and Block Algorithms**

It is obvious that Algorithms 1 and 2 are not efficient, especially because it is based on vector-vector and matrix-vector operations, i.e. Level-1 and Level-2

BLAS. Those operations are memory-bound on modern processors, i.e. their rate of execution is entirely determined by the memory latency suffered in bringing the operands from main memory into the floating point register file.

The corresponding LAPACK block algorithms overcome some of those issues by accumulating the Householder reflectors within the panel and then, by applying at once to the rest of the matrix, i.e. the trailing submatrix, which potentially make those algorithms rich in matrix-matrix (Level-3 BLAS) operations. However, the scalability of block factorizations is limited on a multicore system when parallelism is only exploited at the level of the BLAS routines. This approach will be referred to as the fork-join approach since the execution flow of a block factorization would show a sequence of sequential operations (i.e. the panel factorizations) interleaved to parallel ones (i.e., the trailing submatrix updates). Also, an entire column/row is reduced at a time, which engenders a large stride access to memory.

The whole idea is to transform these algorithms to work on a matrix split into square tiles, with clean-up regions if necessary, in the case where the size of the matrix does not divide evenly. All the elements within the tiles are contiguous in memory following the efficient BDL storage format. and thus the access pattern to memory is more regular. At the same time, this fine granularity greatly improves data locality and cache reuse as well as the degree of parallelism. The Householder reflectors are now accumulated within the tiles during the panel factorization which decrease the length of the stride access to memory. This algorithmic strategy allows clever reuse

of operands already present in registers, and so can run at very high rates. Those operations are indeed compute-bound, i.e. their rate of execution principally depends on the CPU floating point operations per second.

The next section presents the parallel tile versions of these two-sided reductions.

### 3 The Parallel Band Reductions

In this section, we describe the parallel implementation of the HRD and BRD algorithms which reduce a general matrix to band form using tile algorithms.

#### 3.1 Fast Kernel Descriptions

- There are four kernels to perform the tile HRD 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  $j$ .
  - **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 stored in column major format and an upper triangular matrix  $T_{k,k-1}$  as defined by the WY technique [35] 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.

- **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 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 stored in column major format and the matrix  $T_{i,k-1}$  resulting from accumulating the reflectors  $V_{i,k-1}$ .
- **DLARFB**: this kernel is used to apply the transformations computed by 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).
- **DSSRFB**: this kernel applies the reflectors  $V_{i,k-1}$  and the matrix  $T_{i,k-1}$  computed by 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 [10], the right variants for DLARFB and DSSRFB have been developed. The other kernels are exactly the same as [10]. The tile HRD algorithm with Householder reflectors then appears as in Algorithm 3. Figure 1 shows the HRD 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

---

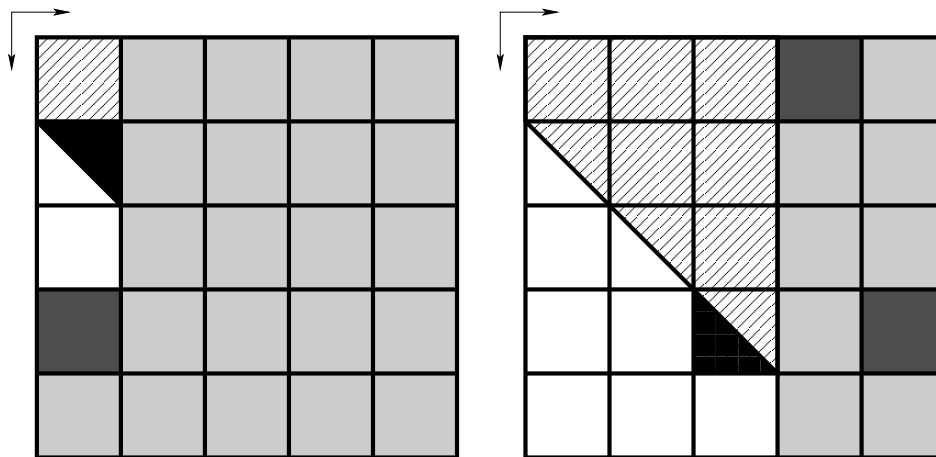
**Algorithm 3** Tile Band HRD Algorithm with Householder reflectors.

---

```
1: for  $k = 2$  to  $nt$  do
2:    $R_{k,k-1}, V_{k,k-1}, T_{k,k-1} \leftarrow \text{DGEQRT}(A_{k,k-1})$ 
3:   for  $j = k$  to  $nt$  do
4:      $A_{k,j} \leftarrow \text{DLARFB}(\textit{left}, V_{k,k-1}, T_{k,k-1}, A_{k,j})$ 
5:   end for
6:   for  $j = 1$  to  $nt$  do
7:      $A_{j,k} \leftarrow \text{DLARFB}(\textit{right}, V_{k,k-1}, T_{k,k-1}, A_{j,k})$ 
8:   end for
9:   for  $i = k + 1$  to  $nt$  do
10:     $R_{k,k-1}, V_{i,k-1}, T_{i,k-1} \leftarrow \text{DTSQRT}(R_{k,k-1}, A_{i,k-1})$ 
11:    for  $j = k$  to  $nt$  do
12:       $A_{k,j}, A_{i,j} \leftarrow \text{DSSRFB}(\textit{left}, V_{i,k-1}, T_{i,k-1}, A_{k,j}, A_{i,j})$ 
13:    end for
14:    for  $j = 1$  to  $nt$  do
15:       $A_{j,k}, A_{j,i} \leftarrow \text{DSSRFB}(\textit{right}, V_{i,k-1}, T_{i,k-1}, A_{j,k}, A_{j,i})$ 
16:    end for
17:  end for
18: end for
```

---

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 using the upper structure of the reference tile (in black). This operation is done by the kernel  $\text{DTSQRT}(R_{2,1}, A_{4,1}, T_{4,1})$ . In Figure 1(b), the reflectors located in the lower part of the reference tile (in black) of the third panel are accordingly applied to the trailing submatrix, e.g., the top dark gray tile is  $\text{DLARFB}(\textit{right}, V_{4,3}, T_{4,3}, A_{1,4})$  while the bottom one is  $\text{DLARFB}(\textit{left}, V_{4,3}, T_{4,3}, A_{4,5})$ .



(a) HRD reduction at step 1.

(b) HRD reduction at step 3.

Figure 1: HRD algorithm applied on a 5-by-5 tile matrix.

- There are eight overall kernels for the tile band BRD implemented for the two phases, four for each phase. For phase 1 (left reduction), the kernels are exactly the ones described above for the tile HRD algorithm,

in which the reflectors are stored in column major format. For phase 2 (right reduction), the reflectors are now stored in row major format.

- **DGELQT**: this kernel performs the  $LQ$  blocked factorization of an upper diagonal tile  $A_{k,k+1}$  of the input matrix. It produces a lower triangular matrix  $R_{k,k+1}$ , a unit upper triangular matrix  $V_{k,k+1}$  containing the Householder reflectors stored in row major format and an upper triangular matrix  $T_{k,k+1}$  as defined by the WY technique [35] 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 lower triangular matrix  $R_{k,k+1}$ , called *reference tile*, is eventually used to annihilate the subsequent tiles located on the right, on the same panel/row.
- **DTSLQT**: this kernel performs the  $LQ$  blocked factorization of a matrix built by coupling the reference tile  $R_{k,k+1}$  that is produced by DGELQT with a tile  $A_{k,j}$  located on the same row. It produces an updated  $R_{k,k+1}$  factor,  $V_{k,j}$  matrix containing the Householder reflectors stored in row major format and the matrix  $T_{k,j}$  resulting from accumulating the reflectors  $V_{k,j}$ .
- **DLARFB**: this kernel is used to apply the transformations computed by DGELQT ( $V_{k,k+1}, T_{k,k+1}$ ) to the tile column  $A_{k+1:nt,k+1}$  (right updates).
- **DSSRFB**: this kernel applies the reflectors  $V_{k,j}$  and the matrix



$T_{k,j}$  computed by DTSLQT to two tile columns  $A_{k+1:nt,k+1}$  and  $A_{k+1:nt,k+1}$  (right updates).

The tile BRD algorithm with Householder reflectors then appears as in Algorithm 4. Only minor modifications are needed for the DLARFB and DSSRFB kernels to take into account the row storage of the reflectors. Moreover, the computed left and right reflectors can be stored in the lower and upper annihilated parts of the original matrix, for later use. Although the algorithm works for rectangular matrices, for simplicity purposes, only square matrices are considered.

Figure 2 highlights the band BRD algorithm on a tile matrix with  $nt = 5$ . The notations and colors introduced previously are reused here. Figure 2(a) shows how the left reduction procedure works during the first step. The dark gray tile corresponds to  $DTSQRT(R_{1,1}, A_{4,1}, T_{4,1})$  which gets annihilated using the upper structure of the reference tile (black). In Figure 2(b), the right reduction procedure occurs in which the dark gray tile corresponding to  $DTSLQT(R_{1,2}, A_{1,4}, T_{1,4})$  gets annihilated using the lower structure of the reference tile black. In Figure 2(c), the reduction is at step 3 and one of the trailing submatrix update operations applied on the left is represented by the dark gray tiles  $DSSRFB(left, V_{4,3}, T_{4,3}, A_{3,4}, A_{4,4})$ . In Figure 2(d), one of the trailing submatrix update operations applied on the right is represented by the dark gray tiles in  $DSSRFB(right, V_{3,5}, T_{3,5}, A_{4,4}, A_{4,5})$ .

All the kernels presented in this section are very rich in matrix-matrix operations. By working on small tiles with BDL, the elements are contiguous in memory and thus the access pattern to memory is more regular, which makes these kernels high performing. It appears necessary then to efficiently schedule the kernels to get high performance in parallel.

---

**Algorithm 4** Tile Band BRD Algorithm with Householder reflectors.

---

```

1: for  $k = 1$  to  $nt$  do
2:   // QR Factorization
3:    $R_{k,k}, V_{k,k}, T_{k,k} \leftarrow \text{DGEQRT}(A_{k,k})$ 
4:   for  $j = k + 1$  to  $nt$  do
5:      $A_{k,j} \leftarrow \text{DLARFB}(\textit{left}, V_{k,k}, T_{k,k}, A_{k,j})$ 
6:   end for
7:   for  $i = k + 1$  to  $nt$  do
8:      $R_{k,k}, V_{i,k}, T_{i,k} \leftarrow \text{DTSQRT}(R_{k,k}, A_{i,k})$ 
9:     for  $j = k + 1$  to  $nt$  do
10:       $A_{k,j}, A_{i,j} \leftarrow \text{DSSRFB}(\textit{left}, V_{i,k}, T_{i,k}, A_{k,j}, A_{i,j})$ 
11:    end for
12:   end for
13:   if  $k < nt$  then
14:     // LQ Factorization
15:      $R_{k,k+1}, V_{k,k+1}, T_{k,k+1} \leftarrow \text{DGELQT}(A_{k,k+1})$ 
16:     for  $j = k + 1$  to  $nt$  do
17:        $A_{j,k+1} \leftarrow \text{DLARFB}(\textit{right}, V_{k,k+1}, T_{k,k+1}, A_{j,k+1})$ 
18:     end for
19:     for  $j = k + 2$  to  $nt$  do
20:        $R_{k,k+1}, V_{k,j}, T_{k,j} \leftarrow \text{DTSLQT}(R_{k,k+1}, A_{k,j})$ 
21:       for  $i = k + 1$  to  $nt$  do
22:          $A_{i,k+1}, A_{i,j} \leftarrow \text{DSSRFB}(\textit{right}, V_{k,i}, T_{k,i}, A_{i,k+1}, A_{i,j})$ 
23:       end for
24:     end for
25:   end if
26: end for

```

---

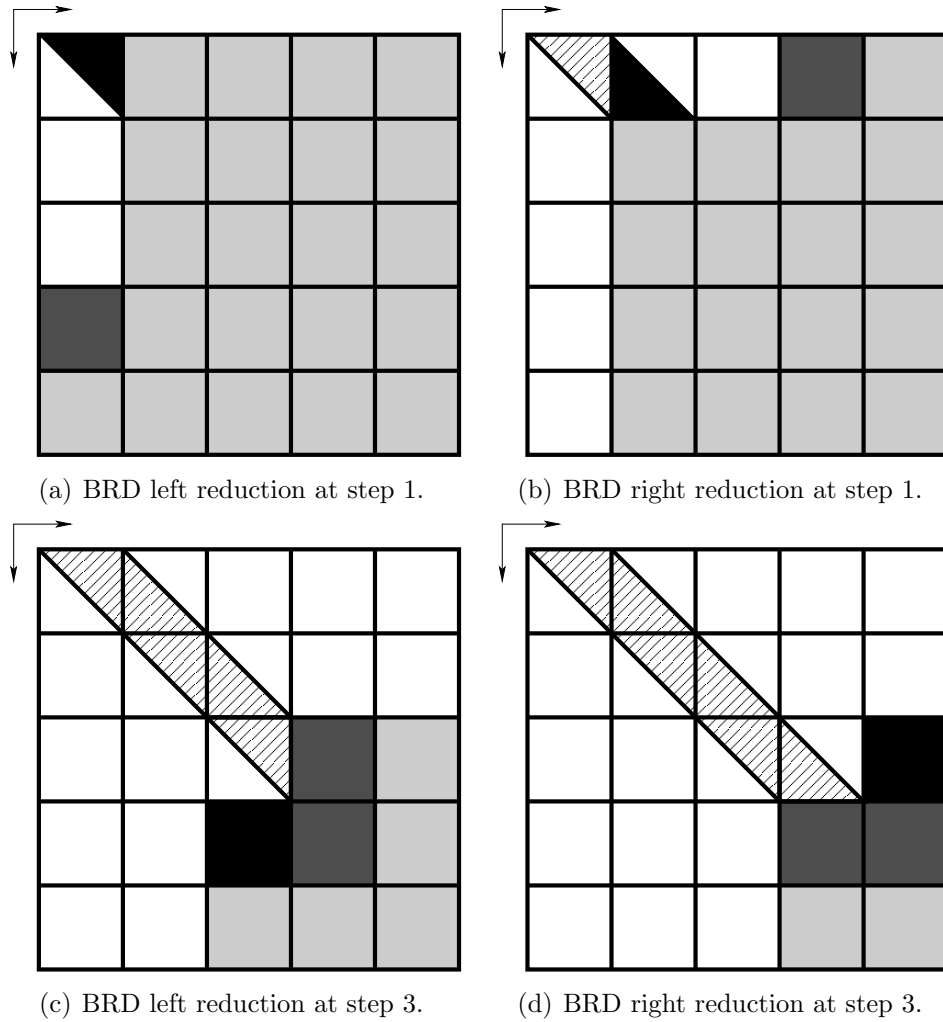


Figure 2: BRD algorithm applied on a 5-by-5 tile matrix.

### 3.2 Parallel Kernel Executions

In this section, we identify the dependencies between tasks by introducing a graphical representation of the parallel executions of Algorithms 3 and 4.

Figure 3 illustrates the step-by-step execution of Algorithm 3 in order

to eliminate the first tile column. The factorization of the panel (DGEQRT and DTSQRT kernels) during the left transformation (Figure 3(a)) is the only part of the algorithm which has to be done sequentially in a domino-like fashion. The updates kernels (DLARFB and DSSRFB) applied during the left as well as the right transformations (Figures 3(a) and 3(b)) can be scheduled concurrently as long as the order in which the panel factorization kernels have been executed is preserved during the corresponding update operations, for numerical correctness. The shape of the band Hessenberg matrix starts to appear as shown in the bottom right matrix in Figure 3(b).

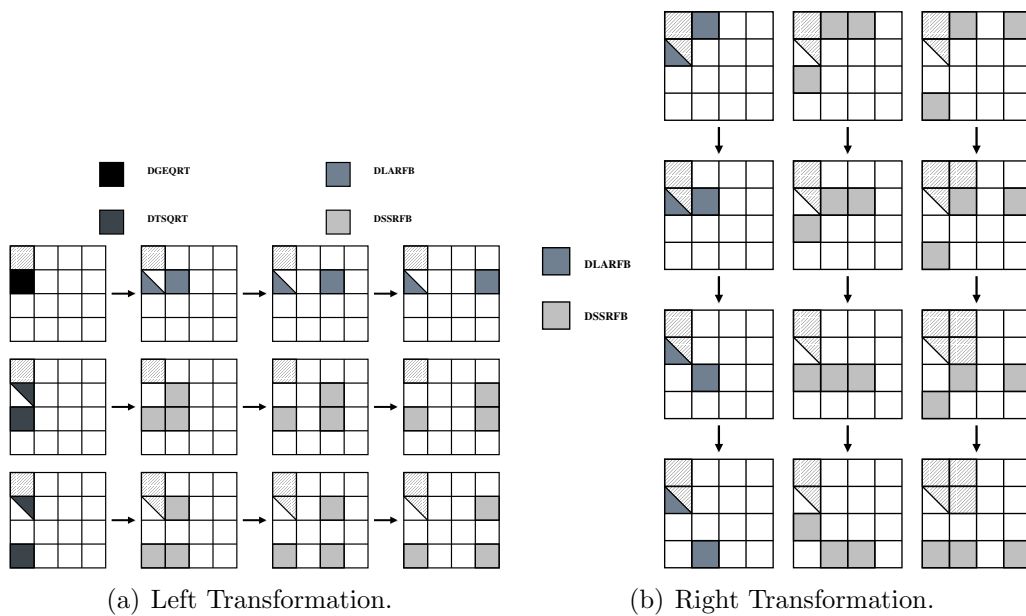


Figure 3: Parallel tile band HRD scheduling.

Figure 4 illustrates the step-by-step execution of Algorithm 4 to eliminate

the first tile column and tile row. The factorization of the row/column panels, DGEQRT and DTSQRT kernels for the left transformation (Figure 4(a)), DGELQT and DTSLQT kernels for the right transformation (Figure 4(b)), is also the only part of the algorithm which has to be done sequentially. Again, the updates kernels can then be run in parallel as long as the order in which the panel factorization kernels have been executed is preserved during the update operations. The shape of the band bidiagonal matrix starts to appear as shown in the bottom right matrix in Figure 4(b).

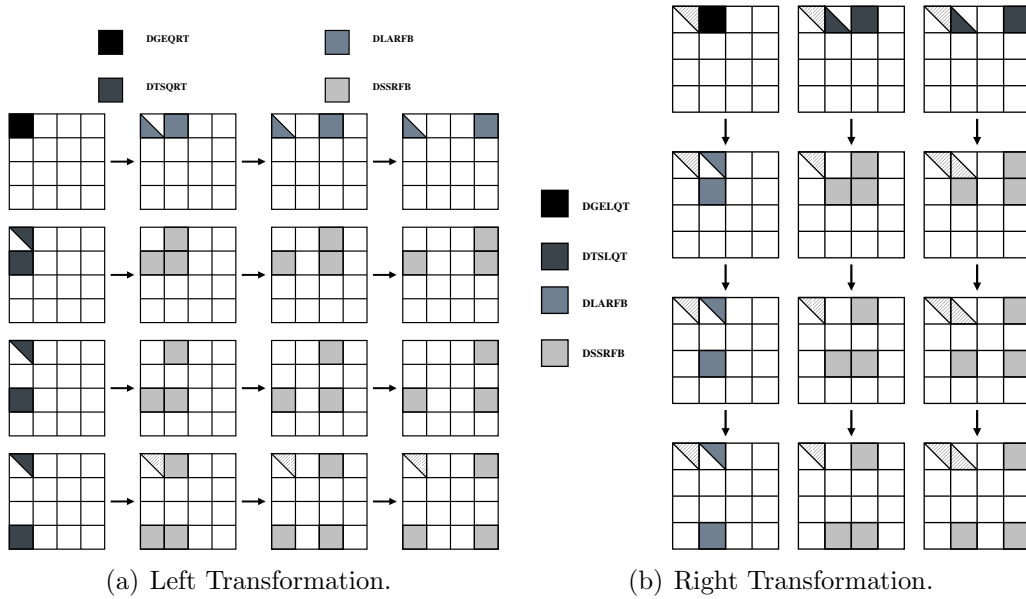


Figure 4: Parallel tile band BRD scheduling.

From Figures 3 and 4 illustrating Algorithms 3 and 4 respectively, we can actually represent a Directed Acyclic Graph (DAG) where nodes are elementary tasks that operate on tiles and where edges represent the dependencies among them. Finally, the data driven execution scheduler has to ensure the

pool of tasks generated by Algorithms 3 and 4 are processed as soon as their respective dependencies are satisfied (more details in section 4).

The next section describes the number of operations needed to perform those reductions using *tile algorithms*.

### 3.3 Arithmetic Complexity

This section presents the complexity of the two band reductions (HRD and BRD).

#### 3.3.1 Band HRD Complexity

If an unblocked algorithm is used with Householder reflectors (see Algorithm 1), the algorithmic complexity for the band HRD algorithm is  $10/3 n (n - b) (n - b)$ , with  $b$  being the tile size. So, compared to the full HRD complexity, i.e.,  $10/3 n^3$ , the band HRD algorithm is performing  $O(n^2 b)$  fewer flops.

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

- **DGEQRT**:  $4/3b^3$  to perform the factorization of the reference tile  $A_{k,k-1}$  and  $2/3b^3$  for computing  $T_{k,k-1}$ .
- **DLARFB**: since  $V_{k,k-1}$  and  $T_{k,k-1}$  are triangular,  $3b^3$  floating-point operations are performed in this kernel.

- **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.
- **DSSRFB**: 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 [10]. The total number of floating-point operations for the band HRD algorithm is then:

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

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 band HRD algorithm. However, as suggested in [15, 16, 17], by setting up inner-blocking within the tiles during the panel factorizations as well as the trailing submatrix updates (i.e., left and right), DGEQRT-DTSQRT kernels and DLARFB-DSSRFB 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 [20, 34]. To understand how this cuts the operation count of the band HRD algorithm, it is important to note that the DGEQRT, DLARFB and DTSQRT kernels only account for lower order terms in the total operation count for the band HRD algorithm. It is, thus, possible to ignore these terms and derive the operation count for the band HRD algorithm as the sum of the cost of all the DSSRFB kernels. The  $T_{i,k-1}$  generated by DTSQRT and used by DSSRFB are not upper triangular anymore but becomes upper-triangular by block thanks to inner-blocking. The cost of a single DSSRFB call drops down, and by ignoring the lower order terms, it is now  $4b^3 + sb^2$ . The total cost of the band HRD algorithm with internal blocking is then:

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

The operation count for the band HRD algorithm with internal blocking is larger 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.



### 3.3.2 Band BRD Complexity

Similarly, the same methodology is applied to compute the complexity of the band BRD algorithm. If an unblocked algorithm is used with Householder reflectors (see Algorithm 2), the algorithmic complexity for the band BRD algorithm is  $4/3 n^3$  (left updates)  $+4/3 n (n - b) (n - b)$  (right updates)  $= 4/3 (n^3 + n (n - b) (n - b))$ , with  $b$  being the tile size. So, compared to the full BRD complexity, i.e.,  $8/3 n^3$ , the band BRD algorithm is performing  $O(n^2 b)$  fewer flops.

The kernels involved in Algorithm 4 in the context of *tile algorithms* during the left transformations are the same than Algorithm 3. The right transformations actually correspond to the transpose of the left transformations and thus, they have the same number of operations. The total number of floating-point operations for the band BRD algorithm is then:

$$\begin{aligned} \sum_{k=2}^{nt} 2 \times (2b^3 + 3(nt - k)b^3 + \frac{10}{3}(nt - k)b^3 + 5(nt - k)^2b^3) & \quad (3) \\ & \simeq 2 \times \frac{5}{3}nt^3b^3 \\ & = \frac{10}{3}n^3 \end{aligned}$$

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 band BRD algorithm. Again, by setting up inner-blocking

within the tiles during the panel factorizations as well as the trailing submatrix updates (i.e., left and right), DGEQRT-DTSQRT-DLARFB-DSSRFB kernels and DGELQT-DTSLQT-DLARFB-DSSRFB respectively, those extra flops become negligible provided  $s \ll b$ , with  $s$  being the inner-blocking size. To understand how this cuts the operation count of the band BRD algorithm, it is important to note that the DGEQRT, DGELQT, DTSQRT, DTSLQT and DLARFB kernels only account for lower order terms in the total operation count for the band BRD algorithm. It is, thus, possible to ignore these terms and derive the operation count for the band BRD algorithm as the sum of the cost of all the DSSRFB kernels. The  $T_{i,k-1}$  and  $T_{k,j}$  generated by DTSQRT/DTSLQT respectively and used by DSSRFB are not upper triangular anymore but becomes upper-triangular by block thanks to inner-blocking. The total cost of the band BRD algorithm with internal blocking is then:

$$\begin{aligned} \sum_{k=2}^{nt} 2 \times (4b^3 + sb^2)(nt - k)^2 &\simeq 2 \times (4b^3 + sb^2) \left(\frac{1}{3}nt^3\right) & (4) \\ &= \left(1 + \frac{s}{4b}\right) \left(\frac{8}{3}n^3\right). \end{aligned}$$

The operation count for the band BRD algorithm with internal blocking is larger 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 band BRD algorithm performs 25% more floating-point operations than the

unblocked algorithm, as stated above.

However, it is noteworthy to mention the high cost of reducing the band Hessenberg / bidiagonal matrix to the full reduced matrix. Indeed, using techniques such as bulge chasing to reduce the band matrix, especially for the band Hessenberg, is very expensive and may dramatically slow down the overall algorithms. Another approach would be to apply the QR algorithm (non symmetric EVP) or the Divide-and-Conquer (SVD) on the band matrix but those strategies are still under investigations.

The next section explains the limitation origins of the *tile algorithms* concept for two-sided transformations, i.e. the reduction up to band form.

### **3.4 Limitations of *Tile Algorithms* Approach for Two-Sided Transformations**

The concept of *tile algorithms* is very suitable for one-sided methods (i.e. Cholesky, LU, 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. For the HRD, we shifted one tile bottom from the top-left corner of the matrix. For the BRD, we decided to shift one tile right from the top-left corner of the

matrix. For the latter algorithm, we could have also performed the shift one tile bottom from the top-left corner of the matrix.

In the following part, we present a comparison of three approaches for tile scheduling, i.e., a static data driven execution scheduler, a hand-coded dynamic data driven execution scheduler and finally, a dynamic scheduler using SMP Superscalar framework.

## 4 Description of the Scheduling Implementations

This section describes three scheduler implementations: a static scheduler where the scheduling is predetermined ahead and two dynamic schedulers where decisions are made at runtime.

### 4.1 Static Scheduling

The static scheduler used here is a derivative of the scheduler used successfully in the past to schedule Cholesky and QR factorizations on the Cell processor [25, 26]. The static scheduler imposes a linear order on all the tasks in the factorization. Each thread traverses the tasks space in this order picking a predetermined subset of tasks for execution. In the phase of applying transformations from the left each thread processes one block-column of the matrix; In the phase of applying transformations from the right each

thread processes one block-row of the matrix (Figure 5). A dependency check is performed before executing each task. If dependencies are not satisfied the thread stalls until they are (implemented by busy waiting). Dependencies are tracked by a progress table, which contains global progress information and is replicated on all threads. Each thread calculates the task traversal locally and checks dependencies by polling the local copy of the progress table. Due to its decentralized nature, the mechanism is much more scalable and of virtually no overhead. This technique allows for pipelined execution of factorizations steps, which provides similar benefits to dynamic scheduling, namely, execution of the inefficient Level 2 BLAS operations in parallel with the efficient Level 3 BLAS operations. Also, processing of tiles along columns and rows provides for greater data reuse between tasks, to which the authors attribute the main performance advantage of the static scheduler. The main disadvantage of the technique is potentially suboptimal scheduling, i.e., stalling in situations where work is available. Another obvious weakness of the static schedule is that it cannot accommodate dynamic operations, e.g., *divide-and-conquer* algorithms.

## 4.2 Hand-coded Dynamic Scheduling

The dynamic scheduling scheme similar to [10] has been extended for the two-sided orthogonal transformations. A DAG is used to represent the data flow between the tasks/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

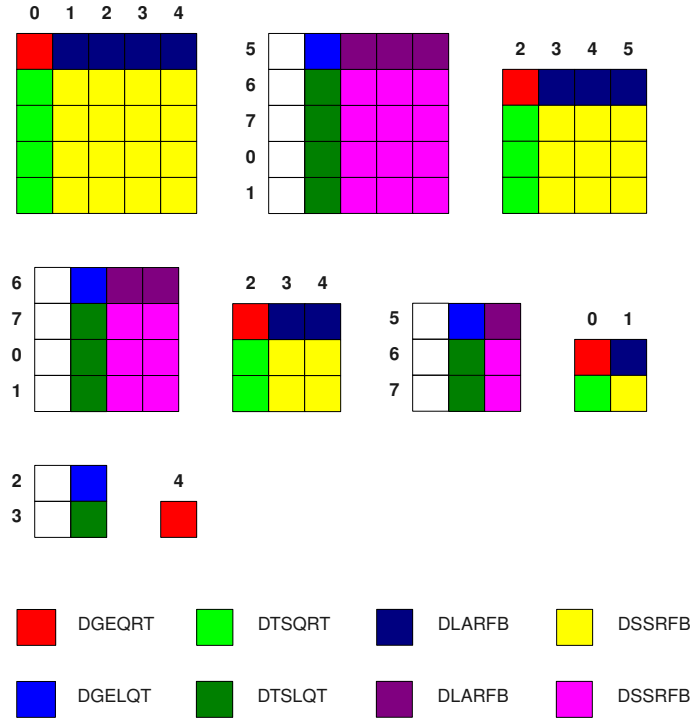


Figure 5: BRD Task Partitioning with eight cores on a  $5 \times 5$  tile matrix.

and it is even more difficult to process than the one created by the one-sided orthogonal transformations. Indeed, the right updates impose severe constraints on the scheduler by filling up the DAG with multiple additional edges. 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 though. The centralized nature of the scheduler may inherently be non-scalable with the number of threads. Also, the need for scanning potentially large table window, in order to find work, may inherently be non-scalable with the problem size. However, this organization does not cause too much performance problems for the numbers of threads, problem sizes and task granularities investigated in this paper.

### 4.3 SMPSs

SMP Superscalar (SMPSs) [2, 32] is a parallel programming framework developed at the Barcelona Supercomputer Center (Centro Nacional de Supercomputación), part of the STAR Superscalar family, which also includes Grid Superscalar and Cell Superscalar [7, 33]. While Grid Superscalar and Cell Superscalar address parallel software development for Grid environments and the Cell processor respectively, SMP Superscalar is aimed at "standard" (x86 and like) multicore processors and symmetric multiprocessor systems. The programmer is responsible for identifying parallel tasks, which have to be side-effect-free (atomic) functions. Additionally, the programmer needs to specify the directionality of each parameter (input, output, inout). If the size of a parameter is missing in the C declaration (e.g., the parameter is passed by pointer), the programmer also needs to specify the size of the memory region affected by the function. However, the programmer 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.

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 (Fortran code are also supported). 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. 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.

By looking at the characteristics of the three schedulers, we can draw some basic conclusions. The static and the hand-coded dynamic schedulers are using orthogonal approaches: the former emphasizes on data reuse between tasks while the latter does not stall if work is available. The philosophy behind the dynamic scheduler framework from SMPSS falls in the middle of the two previous schedulers because not only it proceeds as soon as work is available, but also it tries to reuse data as much as possible. Another aspect which has to be taken into account is the coding effort. Indeed, the easy of



use of SMPSs makes it very attractive for end-users and puts it on top of the other schedulers discussed in this paper.

## 5 Experimental Results

The experiments have been achieved on a quad-socket quad-core Intel Tiger-ton 2.4 GHz (16 total cores) with 32GB of memory. Hand tuning based on empirical data has been performed for large problems to determine the optimal tile size  $b = 200$  and inner-blocking size  $s = 40$  for the tile band HRD and BRD algorithms. The block sizes for LAPACK and ScaLAPACK (configured for shared-memory) have also been hand tuned to get a fair comparison,  $b = 32$  and  $b = 64$  respectively.

Figures 6 and 8 show the band HRD and BRD execution time in seconds for different matrix sizes. They outperform by far the MKL, LAPACK and ScaLAPACK implementations. The authors understand that it may not be a fair comparison to do against those latter libraries, since the reduction is completely achieved in that case. The purpose of showing such performance curves is only to give a rough idea in term of elapsed time and performance, of the whole reduction process.

On the other hand, Figures 7 and 9 present the parallel performance in Gflop/s of the band HRD and BRD algorithms. The different scheduler implementations scale quite well while the matrix size increases.

For the band HRD, the static scheduling and SMPSs are having very

similar performance reaching 102 Gflop/s, i.e. 67% of the system theoretical peak and 78% of DGEMM peak for large matrix size. The dynamic scheduling asymptotically reaches 94 Gflop/s, runs at 61% of the system theoretical peak and 72% of the DGEMM peak.

For the band BRD, SMPs is running slightly better than the two other schedulers reaching 97 Gflop/s, i.e. 63% of the system theoretical peak and 75% of DGEMM peak. The static and dynamic scheduling reach 94 Gflop/s, runs at 61% of the system theoretical peak and 72% of the DGEMM peak.

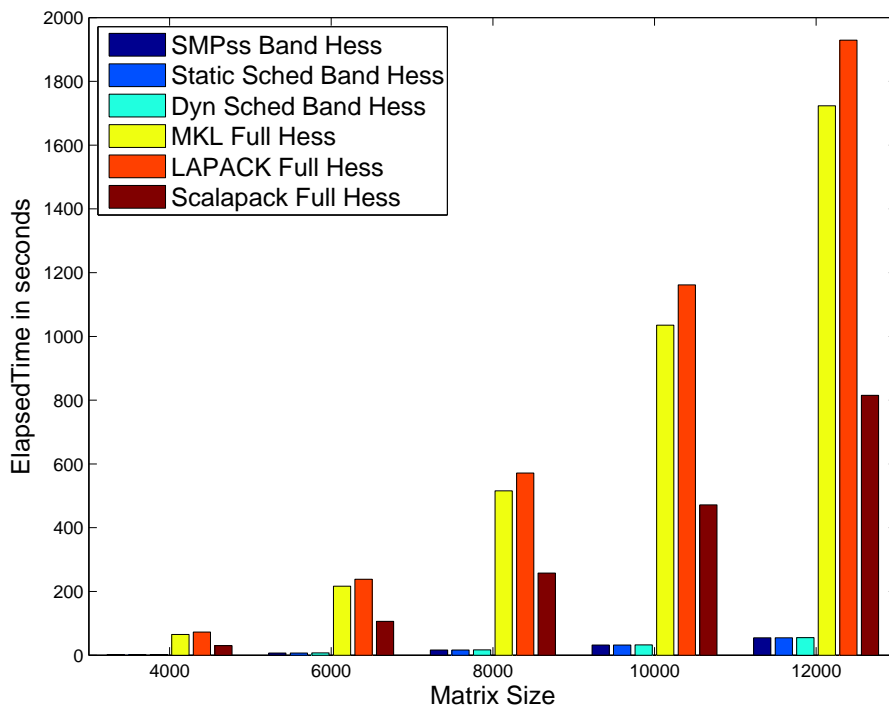


Figure 6: Elapsed time in seconds of the band HRD on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS V10.0.1.

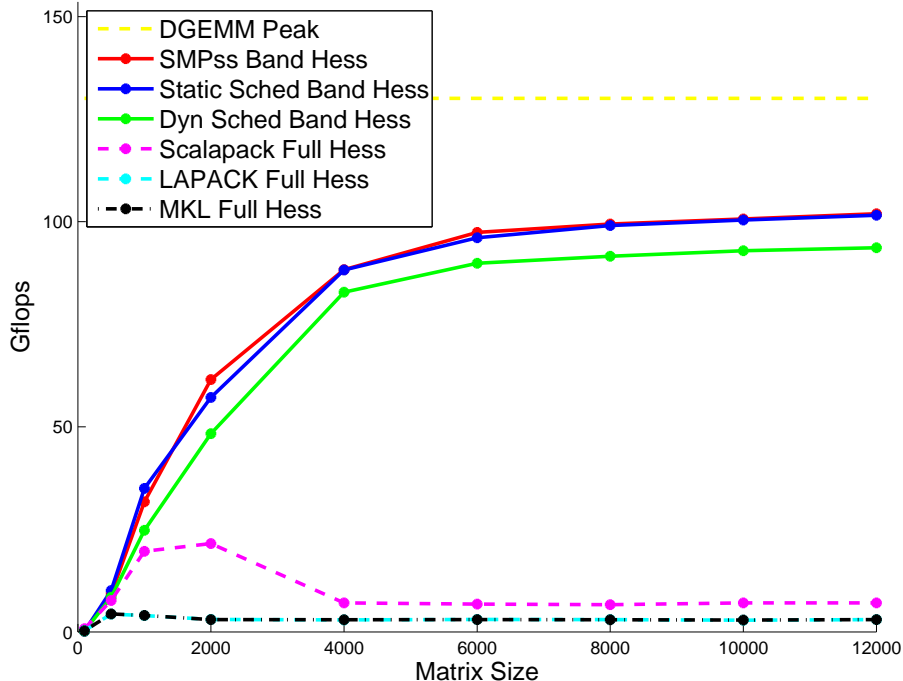


Figure 7: Performance in Gflop/s of the band HRD on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS V10.0.1.

## 6 Related Work

Dynamic data-driven scheduling is an old concept and has been applied to dense linear operations for decades on various hardware systems. The earliest reference, that the authors are aware of, is the paper by Lord, Kowalik and Kumar [29]. A little later dynamic scheduling of LU and Cholesky factorizations were reported by Agarwal and Gustavson [3, 4]. Throughout the years dynamic scheduling of dense linear algebra operations has been used in numerous vendor library implementations such as ESSL, MKL and ACML (numerous references are available on the Web). In recent years the authors

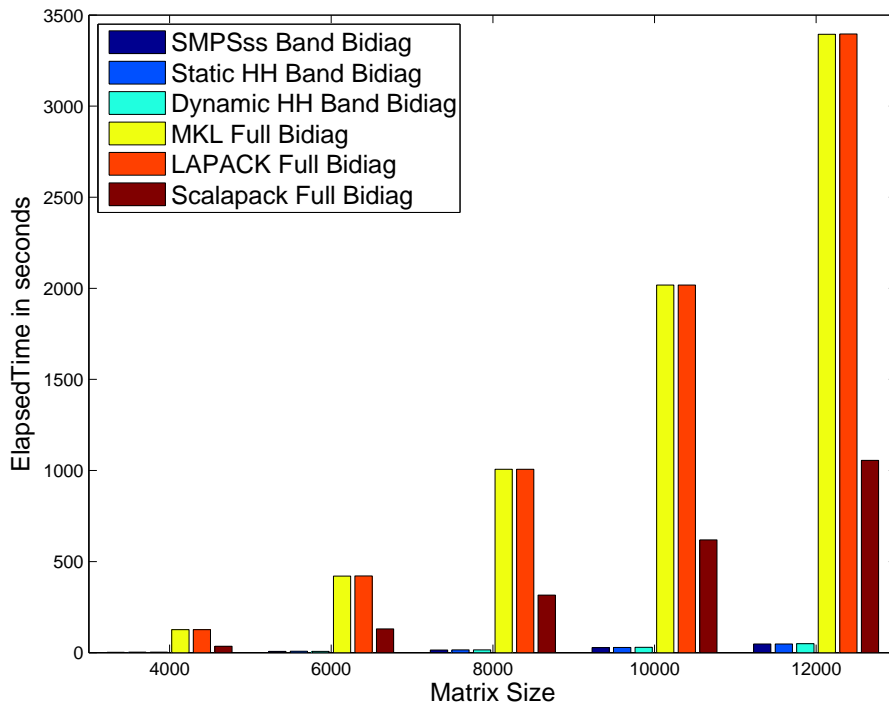


Figure 8: Elapsed time in seconds of the band BRD on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS V10.0.1.

of this work have been investigating these ideas within the framework *Parallel Linear Algebra for Multicore Architectures* (PLASMA) at the University of Tennessee [9, 11, 12, 28].

Seminal work in the context of the *tile QR* factorization was done by Elmroth et al. [15, 16, 17]. Gunter et al. presented an "out-of-core" (out-of-memory) implementation [21], Buttari et al. an implementation for "standard" (x86 and alike) multicore processors [11, 12], and Kurzak et al. an implementation on the CELL processor [27].

Seminal work on performance-oriented data layouts for dense linear alge-

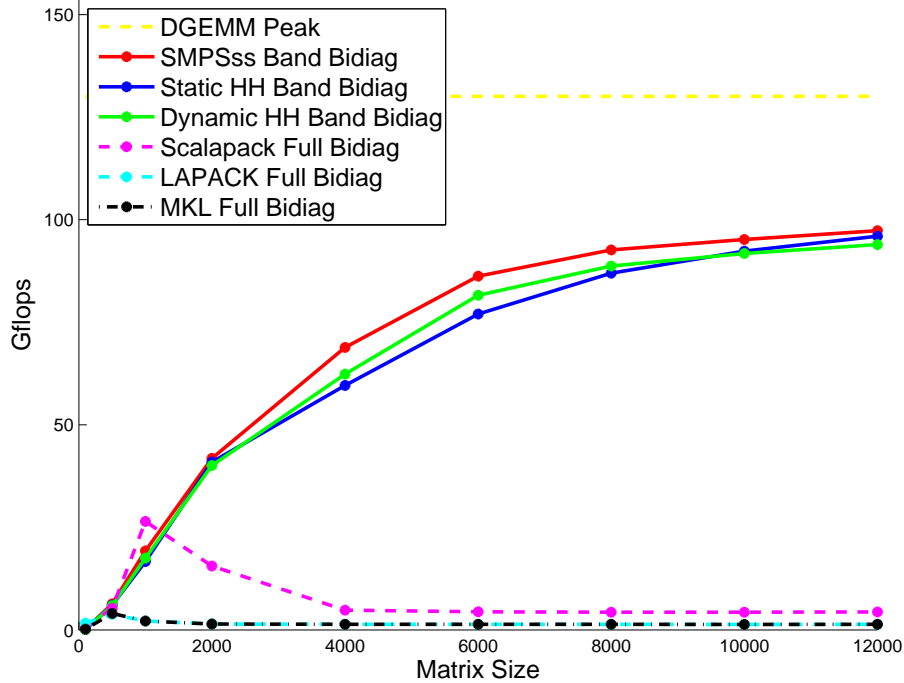


Figure 9: Performance in Gflop/s of the band BRD on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS V10.0.1.

bra was done by Gustavson et al. [23, 22] and Elmroth et al. [18] and was also investigated by Park et al. [30, 31].

## 7 Conclusion and Future Work

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 (i.e., SMPSSs), the HRD and BRD algorithms with Householder reflectors achieve

102 Gflop/s and 97 Gflop/s respectively , on a  $12000 \times 12000$  matrix size with 16 Intel Tigerton 2.4 GHz processors. These algorithms perform most of the operations in Level-3 BLAS.

The main drawback of the *tile algorithms* approach for two-sided transformations is that the full reduction can not be obtained in one stage. Other methods have to be considered to further reduce the band matrices to the required forms. For example, the sequential framework PIRO\_BAND [1] efficiently performs the reduction of band matrices to bidiagonal form using the bulge chasing method. The authors are also looking at one-sided HRD implementations done by Hegland *et al.* [24] and one-sided BRD implementations done by Barlow *et al.* [6] and later, Bosner *et al.* [8] to reduce the original matrix through a one-stage only procedure.

## References

- [1] PIRO\_BAND: PIpelined ROtations for BAnd Reduction. <http://www.cise.ufl.edu/~srajaman/>.
- [2] SMP Superscalar (SMPSs) User's Manual, Version 2.0. Barcelona Supercomputing Center, 2008.
- [3] R. C. Agarwal and F. G. Gustavson. A parallel implementation of matrix multiplication and LU factorization on the IBM 3090. Proceedings of

- the IFIP WG 2.5 Working Conference on Aspects of Computation on Asynchronous Parallel Processors, pages 217–221, Aug 1988.
- [4] R. C. Agarwal and F. G. Gustavson. Vector and parallel algorithms for Cholesky factorization on IBM 3090. Proceedings of the 1989 ACM/IEEE conference on Supercomputing, pages 225–233, Nov 1989.
- [5] 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.
- [6] J. L. Barlow, N. Bosner, and Z. Drmač. A new stable bidiagonal reduction algorithm. *Linear Algebra and its Applications*, 397(1):35–84, Mar. 2005.
- [7] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A programming model for the Cell BE architecture. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Nov. 11-17 2006.
- [8] N. Bosner and J. L. Barlow. Block and parallel versions of one-sided bidiagonalization. *SIAM J. Matrix Anal. Appl.*, 29(3):927–953, 2007.
- [9] A. Buttari, J. J. Dongarra, P. Husbands, J. Kurzak, and K. Yelick. Multithreading for synchronization tolerance in matrix factorization. In *Scientific Discovery through Advanced Computing, SciDAC 2007*, Boston,

- MA, June 24-28 2007. Journal of Physics: Conference Series 78:012028, IOP Publishing.
- [10] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel Tiled QR Factorization for Multicore Architectures. LAPACK Working Note 191, July 2007.
- [11] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008.
- [12] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput. Syst. Appl.*, 35:38–53, 2009.
- [13] 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.
- [14] K. E. Danforth Christopher M. and M. Takemasa. Estimating and correcting global weather model error. *Monthly weather review*, 135(2):281–299, 2007.
- [15] E. Elmroth and F. G. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. *Applied Parallel Computing*,



- Large Scale Scientific and Industrial Problems, 4th International Workshop, PARA, Lecture Notes in Computer Science, 31(1):1541:120–128, Jun 1998.
- [16] 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.
- [17] E. Elmroth and F. G. Gustavson. High-performance library software for QR factorization. Applied Parallel Computing, New Paradigms for HPC in Industry and Academia, 5th International Workshop, PARA, Lecture Notes in Computer Science, pages 1947:53–63, Jun 2000.
- [18] E. Elmroth, F. G. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.
- [19] 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.
- [20] 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.

- [21] B. C. Gunter and R. A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, 2005.
- [22] F. G. Gustavson. New generalized matrix data structures lead to a variety of high-performance algorithms. *Proceedings of the IFIP WG 2.5 Working Conference on Software Architectures for Scientific Computing Applications*, Kluwer Academic Publishers, pages 211–234, Oct 2000.
- [23] F. G. Gustavson, J. A. Gunnels, and J. C. Sexton. Minimal data copy for dense linear algebra factorization. *Applied Parallel Computing, State of the Art in Scientific Computing, 8th International Workshop, PARA, Lecture Notes in Computer Science*, pages 4699:540–549, Jun 2006.
- [24] M. Hegland, M. Kahn, and M. Osborne. A parallel algorithm for the reduction to tridiagonal form for eigendecomposition. *SIAM J. Sci. Comput.*, 21(3):987–1005, 1999.
- [25] J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equation on the CELL processor using Cholesky factorization. *Trans. Parallel Distrib. Syst.*, 19(9):1175–1186, 2008.
- [26] J. Kurzak and J. Dongarra. QR Factorization for the CELL Processor. LAPACK Working Note 201, May 2008.
- [27] J. Kurzak and J. J. Dongarra. QR factorization for the CELL processor. *Scientific Programming*, accepted.

- [28] J. Kurzak and J. J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. *Applied Parallel Computing, State of the Art in Scientific Computing, 8th International Workshop, PARA, Lecture Notes in Computer Science*, pages 4699:147–156, Jun 2006.
- [29] R. E. Lord, J. S. Kowalik, , and S. P. Kumar. Solving linear algebraic equations on an MIMD computer. *J. ACM*, 30(1):103–117, 1983.
- [30] N. Park, B. Hong, and V. K. Prasanna. Analysis of memory hierarchy performance of block data layout. Proceedings of the 2002 International Conference on Parallel Processing, ICPP’02, IEEE Computer Society, pages 35–44, 2002.
- [31] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):640–654, 2003.
- [32] J. M. Pérez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *CLUSTER*, pages 142–151. IEEE, 2008.
- [33] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM J. Res. & Dev.*, 51(5):593–604, 2007.

- [34] 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.
- [35] 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.
- [36] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.