Scheduling Two-sided Transformations using Algorithms-by-Tiles on Multicore Architectures *LAPACK Working Note #214*

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

Abstract. The objective of this paper is to describe, in the context of multicore architectures, 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 coarse-grain dataflow model gains popularity as a paradigm for programming multicore architectures. By using the concepts of *algorithms-by-tiles* [Buttari *et al.*, 2007] along with efficient mechanisms for data-driven execution, these two-sided reductions achieve high performance computing. The main drawback of the *algorithms-by-tiles* 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.

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 [4] and ScaLAPACK [12], which deliver a small fraction of peak performance on current multicore processors and multi-socket systems of multicore processors, mostly following *Symmetric Multi-Processor* (SMP) architecture.

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

This paper presents different scheduling schemes 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]:

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

  The need to solve EVPs emerges from various computational science disciplines e.g., structural engineering, electronic structure calculations, computational fluid dynamics, and also, in information technology e.g., search engines rank websites [23]. 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, 37]:

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

  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 [13]. 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 originaly discussed in [9] for one-sided transformations, the *algorithms-by-tiles* approach is a combination of several parameters 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 *algorithms-by-tiles* 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 document is organized as follows: Section 2 recalls the standard HRD and BRD algorithms. Section 3 describes the parallel tiled HRD and BRD 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 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. Four 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).

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 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

Algorithm 2 Bidiagonal Reduction with Householder reflectors

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

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 annihilate one element of the matrix, which makes the total number of operations for such algorithm $8/3\ n^3$ (the lower order terms are neglected).

2.3 Limitations of the Standard Reductions

It is obvious that algorithms 1 and 2 are not efficient, especially because it is based on matrix-vector Level-2 BLAS operations. Also, a single 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 tiles instead in order to improve data locality and cache reuse. Also, the Householder reflectors are accumulated within the tiles and then applied at once, which potentially make those algorithms rich in matrix-matrix operations. The next section presents the parellel tiled versions of these two-sided reductions.

3 The Parallel Band Reductions

In this section, we present the parallel implementation of the band HRD and BRD algorithms.

3.1 Fast Kernel Descriptions

– The tiled band HRD kernels are identical to the ones used by Buttari *et. al* in [9] for the QR factorization. Basically, DGEQRT is used to do a QR blocked factorization using the WY technique for efficiently accumulating the Householder reflectors [35]. The DLARFB kernel comes from the LAPACK distribution and is used to apply a block of Householder reflectors. DTSQRT

Algorithm 3 Tiled HRD 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
```

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. [9] 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, the computed reflectors can be stored in the lower annihilated part of the original matrix for later use. Let NBT be the number of tiles in each direction. The tiled band HRD algorithm with Householder reflectors then appears as in algorithm 3.

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).

– There are eight overall kernels for the tiled band BRD implemented for the two phases, four for each phase. For phase 1 (left reduction), the first four kernels are exactly the ones used by Buttari *et al.* [9] for the QR factorization, in which the reflectors are stored in column major form. For phase 2 (right reduction), the reflectors are now stored in rows. DGELQT is used to do a LQ blocked factorization using the WY technique as well. DTSLQT

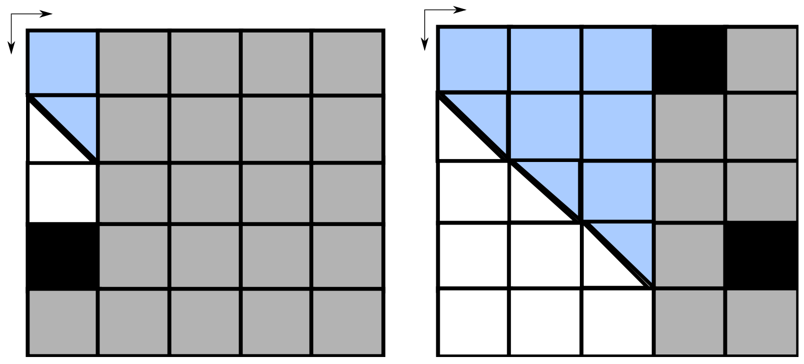

(a) HRD: Reduction Step 1. (b) HRD: Reduction Step 3.

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

performs a block LQ factorization of a matrix composed of two tiles, a triangular tile beside a dense square tile. Again, 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.

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 2: (i) corresponds to the reduction step in the general algorithm, (ii) gives the row index and (iii) represents the column index. For example, in figure 2(a), the black tile is the input dependency at the current step, the white tiles are the zeroed tiles, the bright gray tiles are those which need to be processed and finally, the dark gray tile corresponds to DTSQRT(1,4,1). In figure 2(b), the blue tiles represent the final data tiles and the dark gray tile is DLARFB(”R”,1,1,4). In figure 2(c), the reduction is at step 3 where the dark gray tiles represent DSSRFB(”L”,3,4,4). In figure 2(d), the dark gray tiles represent DSSRFB(”R”,3,4,5).

All the kernels presented in this section are very rich in matrix-matrix operations. By working on small tiles with block data layout, the elements are stored 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 Tiled Band BRD Algorithm with Householder reflectors.

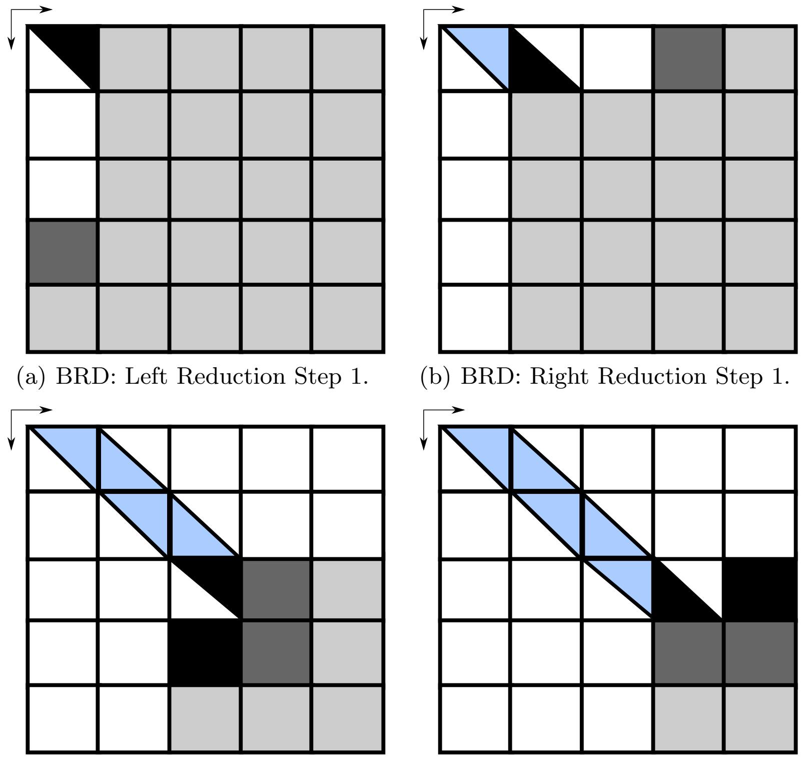
```
1: for $i = 1, 2$ to NBT do
2:     // QR Factorization
3:     DGEQRT($i, i, i$)
4:     for $j = i + 1$ to NBT do
5:         DLARFB("$L$", $i, i, j$)
6:     end for
7:     for $k = i + 1$ to NBT do
8:         DTSQRT($i, k, i$)
9:         for $j = i + 1$ to NBT do
10:            DSSRFB("$L$", $i, k, j$)
11:        end for
12:    end for
13:    if $i <$ NBT then
14:        // LQ Factorization
15:        DGELQT($i, i, i + 1$)
16:        for $j = i + 1$ to NBT do
17:            DLARFB("$R$", $i, j, i + 1$)
18:        end for
19:        for $k = i + 2$ to NBT do
20:            DTSLQT($i, i, k$)
21:            for $j = i + 1$ to NBT do
22:                DSSRFB("$R$", $i, j, k$)
23:            end for
24:        end for
25:    end if
26: end for
```

3.2 Parallel Kernel Executions

Figure 3(a) and 3(b) illustrate 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) is the only part of the algorithm which has to be done sequentially. The updates kernels can be run concurrently as long as the order in which the panel factorization has been executed is preserved during the update procedures, for numerical correctness.

Figure 4(a) and 4(b) illustrate 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, DTSQRT, DGELQT and DTSLQT kernels) is also the only part of the algorithm which has to be done sequentially. The updates kernels can then be run in parallel as long as the order in which the panel factorizations have been executed is preserved during the update procedures.

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).


(a) BRD: Left Reduction Step 1. (b) BRD: Right Reduction Step 1.

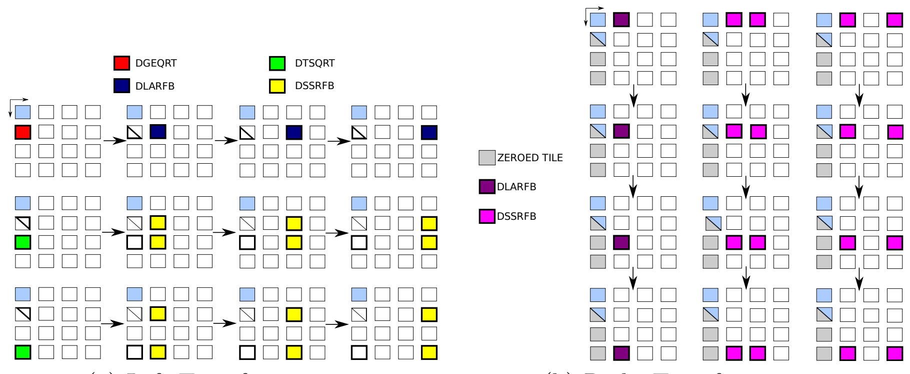
(c) BRD: Left Reduction Step 3. (d) BRD: Right Reduction Step 3.

Fig. 2. BRD algorithm applied on a tiled Matrix with NBT= 5.

The next section describes the number of operations needed to perform those reductions.

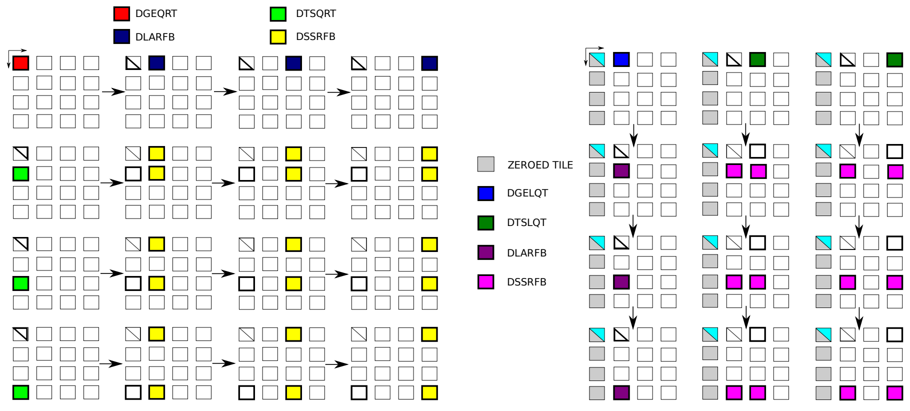
3.3 Arithmetic Complexity

By using updating factorization techniques as suggested in [19, 36], the kernels for both reducing algorithms can be applied to tiles of the original matrix. Using updating techniques to tile the algorithms have first been proposed by Yip [38] for LU to improve the efficiency of out-of-core solvers, and were recently reintroduced in [20, 33] for LU and QR, once more in the out-of-core context. The cost of these updating techniques is an increase in the operation count for the whole HRD and BRD reductions. However, as suggested in [15–17], by setting up inner-blocking within the tiles during the panel factorizations and the trailing submatrix update, DGEQRT-DGELQT-DTSQRT-DTSLQT kernels and DLARFB-DSSRFB kernels respectively, those extra flops become negligible provided $s << b$, with b being the tile size (equivalent to the bandwidth of



(a) Left Transformation. (b) Right Transformation.

Fig. 3. Parallel tiled band HRD scheduling.



(a) Left Transformation. (b) Right Transformation.

Fig. 4. Parallel tiled band BRD scheduling.

the matrix) and s being the inner-blocking size (see Buttari *et al.* [9] for further information). This blocking approach has been also described in [20, 34].

The algorithmic complexity for the band HRD is divided into two steps. The left transformation is basically a QR factorization which counts for $4/3\ n\ (n - b)\ (n - b)$. The cost of the right transformation is $2\ n^3$. The total number of flops is then $2\ (n^3 + \frac{2}{3}\ n\ (n - b)\ (n - b))$. Compared to the full HRD reduction complexity, i.e., $10/3\ n^3$, the band HRD algorithm is doing $O(n^2\ b)$ less flops, which is a negligible expense of the overall HRD algorithm cost provided $n >> b$.

The algorithmic complexity for the band BRD is also split into two phases: QR factorization with $4/3\ n^3$ and a band LQ factorization with $4/3\ n\ (n - b)\ (n - b)$. The total number of flops is then $4/3\ (n^3 + n\ (n - b)\ (n - b))$. Compared to the full BRD reduction complexity, i.e., $8/3\ n^3$, the band BRD algorithm is doing $O(n^2\ b)$ less flops, which is a negligible expense of the overall BRD algorithm cost provided $n\ >>\ b$.

However, it is noteworthy to mention the high cost of reducing the band hessenberg / bidiagonal matrix to the full reduced matrix. Indeed, using technics such as bulge chasing to reduce the band matrix is very exepensive 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 sill under investigations.

The next section explains in details the limitations of the concept of *algorithms-by-tiles* for two-sided transformations, i.e. the band reduction.

3.4 Limitations of *Algorithms-by-Tiles* Approach for Two-Sided Transformations

The concept of *algorithms-by-tiles* 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.

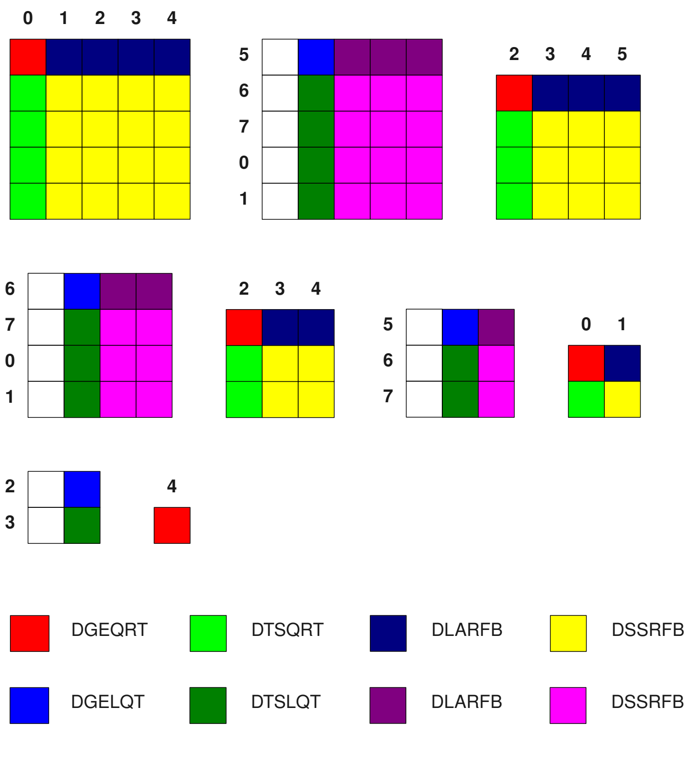


Fig. 5. BRD Task Partitioning with eight cores on a 5 × 5 tile matrix.

4.2 Hand-coded Dynamic Scheduling

The dynamic scheduling scheme similar to [9] has been extended for the two-sided orthogonal transformations. A Directed Acyclic Graph (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 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) [1] 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 Supercalar and Cell Superscalar [6, 32]. While Grid Superscalar and Cell Superscalar address parallel software development for Grid enviroments 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.

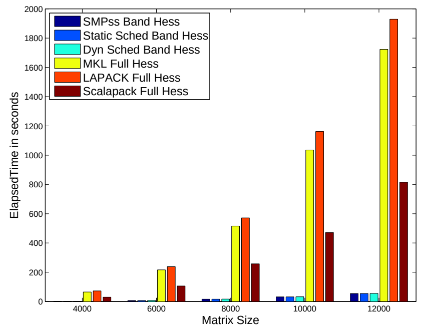
5 Experimental Results

The experiments have been achieved on a quad-socket quad-core Intel Tigerton 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 tiled 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.

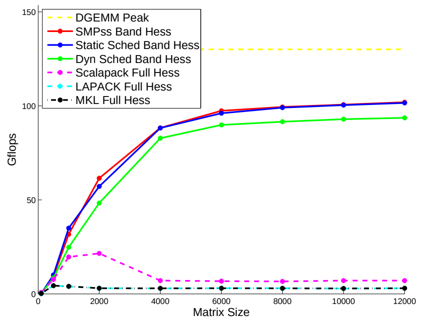
Figures 6(a) and 7(a) show the band HRD and BRD execution time in seconds for different matrix sizes. They outperform by far the MKL, LAPACK and ScaLAPACK implementations. Figures 6(b) and 7(b) present the parallel performance in Gflop/s of the band HRD and BRD algorithms. The different scheduler implementations 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, SMPSs 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.

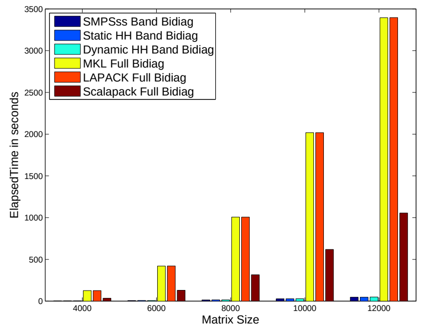


(a) Elapsed time in seconds.

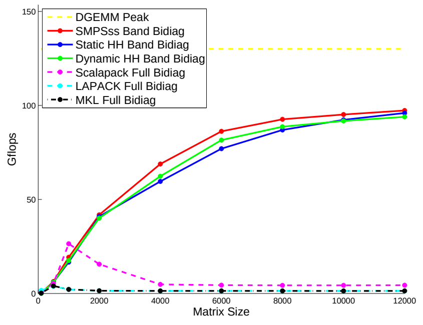


(b) Performance in Gflop/s.

Fig. 6. Experimenting band HRD on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS V10.0.1.



(a) Elapsed time in seconds.



(b) Performance in Gflop/s.

Fig. 7. Experimenting band BRD 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 [2, 3] 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 of this work have been investigating these ideas within the framework *Parallel Linear Algebra for Multicore Architectures* (PLASMA) at the University of Tennessee [8, 10, 11, 28].

Seminal work in the context of the *tile QR* factorization was done by Elmroth et al. [15–17]. 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 [10, 11], and Kurzak et al. an implementation on the CELL processor [27].

Seminal work on performance-oriented data layouts for dense linear algebra was done by Gustavson et al. [18, 22] and Elmroth et al. [14] and was also investigated by Park et al. [30, 31].

7 Conclusion 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 (i.e., SMPSs), the HRD and BRD algorithms with Householder reflectors achieve 102 Gflop/s and 97 Gflop/s respectively , on a 12000×12000 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 *algorithms-by-tiles* 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. The authors are looking, for example, at one-sided HRD implementations done by Hegland *et al.* [24] and one-sided BRD implementations done by Barlow *et al.* [5] and later, Bosner *et al.* [7] to try to overcome those limitations.

References

1. SMP Superscalar (SMPSs) User's Manual, Version 2.0. Barcelona Supercomputing Center, 2008.
2. 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.
3. 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.
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. 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.
6. 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.
7. N. Bosner and J. L. Barlow. Block and parallel versions of one-sided bidiagonalization. *SIAM J. Matrix Anal. Appl.*, 29(3):927–953, 2007.

8. 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.
9. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel Tiled QR Factorization for Multicore Architectures. LAPACK Working Note 191, July 2007.
10. 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.
11. A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parellel Comput. Syst. Appl.*, 35:38–53, 2009.
12. 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.
13. K. E. Danforth Christopher M. and M. Takemasa. Estimating and correcting global weather model error. *Monthly weather review*, 135(2):281–299, 2007.
14. I. J. E. Elmroth, F. G. Gustavson and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review, 46(1):3–45, 2004.
15. 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. http://dx.doi.org/10.1007/BFb0095328.
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. 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.
18. J. A. G. F. G. Gustavson 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.
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.
23. T. H. Haveliwala and A. D. Kamvar. The second eigenvalue of the google matrix. Technical report, Stanford University, Apr. 18 2003.

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. http://dx.doi.org/10.1109/TPDS.2007.70813.
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. B. H. N. Park 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. B. H. N. Park 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. 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.
33. 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.
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. G. W. Stewart. *Matrix Algorithms Volume I: Matrix Decompositions.* SIAM, Philadelphia, 1998.
37. L. N. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, Philadelphia, PA, 1997.
38. E. L. Yip. Fortran subroutines for out-of-core solutions of large complex linear systems. *Technical Report CR-159142, NASA*, November 1979.