Two-Stage Tridiagonal Reduction for Dense Symmetric Matrices using Tile Algorithms on Multicore Architectures

Piotr Luszczek*, Hatem Ltaief* and Jack Dongarra*
*Innovative Computing Laboratory, University of Tennessee, Knoxville, TN 37996, USA
Email: luszczek,ltaief,dongarra@eecs.utk.edu

***Abstract*—While successful implementations have already been written for one-sided transformations (e.g., QR, LU and Cholesky factorizations) on multicore architecture, getting high performance for two-sided reductions (e.g., Hessenberg, tridiagonal and bidiagonal reductions) is still an open and difficult research problem due to expensive memory-bound operations occurring during the panel factorization. The processor-memory speed gap continues to widen, which has even further exacerbated the problem. This paper focuses on an efficient implementation of the tridiagonal reduction, which is the first algorithmic step toward computing the spectral decomposition of a dense symmetric matrix. The original matrix is translated into a *tile* layout i.e., a high performance data representation, which substantially enhances data locality. Following a two-stage approach, the tile matrix is then transformed into band tridiagonal form using compute intensive kernels. The band form is further reduced to the required tridiagonal form using a *left-looking* bulge chasing technique to reduce memory traffic and memory contention. A dependence translation layer associated with a dynamic runtime system allows for scheduling and overlapping tasks generated from both stages. The obtained tile tridiagonal reduction significantly outperforms the state-of-the-art numerical libraries (10X against multithreaded LAPACK with optimized MKL BLAS and 2.5X against the commercial numerical software Intel MKL) from medium to large matrix sizes.**

***Keywords*-Tridiagonal Reduction; Tile Algorithms; Bulge Chasing; Translation Layer; Scheduling;**

I. Introduction

According to the last Top500 list from June 2010 [1], 99% of the fastest parallel systems in the world were based on multicores. The first ranked system, *Jaguar* from Oak Ridge National Laboratory, achieved an astonishing peak performance of 1.2 Pflop/s. This confronts the scientific software community with both a daunting challenge and a unique opportunity. The challenge arises from the disturbing mismatch between the design of systems based on this chip architecture – hundreds of thousands of nodes, a million or more powerful cores, reduced bandwidth and memory available to cores – and the components of the traditional software stack, such as numerical libraries, on which scientific applications have relied for their accuracy and performance. For example, the processor-memory speed gap will continue to widen. The memory speed is clearly expected to grow by an order of magnitude *less* compared to the processor speed in the next few years [2].

This trend has already started to negatively impact the performance of important and widely-used numerical applications. Indeed, while successful implementations have already been written for compute intensive one-sided transformations (e.g., QR, LU and Cholesky factorizations) on multicore architectures [3], [4], getting high performance for two-sided reductions (e.g., Hessenberg, tridiagonal and bidiagonal reductions) is still an open and difficult research problem due to expensive memory-bound operations occurring during the panel factorization. Generally speaking, there are basically two ways to alleviate bottlenecks related to memory bandwidth: (1) a software solution, which involves redesigning the algorithm to cast most operations in Level 3 BLAS and hide the slow memory accesses or (2) a hardware solution, by relying on high bandwidth devices such as GPU accelerators.

This paper proposes a software solution to overcome the memory bottleneck of the tridiagonal reduction (TRD), which is the pre-processing algorithmic step toward computing the spectral decomposition of a dense symmetric matrix [5], [6]. The common way of stating the problem is:

$$Ax = \lambda x, \qquad A \in \mathbb{R}^{n \times n},\ x \in \mathbb{R}^n,\ \lambda \in \mathbb{R}.$$

with A being a symmetric or Hermitian matrix ($A = A^T$ or $A = A^H$), λ – an eigenvalue, and x the corresponding eigenvector. The goal is to transform the matrix A into a symmetric (or hermitian) tridiago-

nal matrix S:

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

The necessity of calculating eigenvalues/eigenvectors emerges from various computational science disciplines e.g., in quantum chemistry [7], quantum mechanics [8], quantum physics [9] and statistics when computing the principal component analysis of the symmetric covariance matrix. Moreover, the tridiagonalisation step is the most time consuming phase. It can reach more than 90% of the elapsed time when calculating the eigenvalues and roughly 50% when both eigenvalues and eigenvectors are to be computed. In our tridiagonalisation implementation, the original matrix is translated into a *tile* layout i.e., a high performance data representation, which enhances memory accesses and data locality by appropriately fitting the core small caches. Following a two-stage approach, the tile matrix is then transformed into band tridiagonal form using compute intensive kernels. The band form is further reduced to the required tridiagonal form using a new bulge chasing technique with a *left-looking* variant to reduce memory traffic and memory contention. This second stage uses non-tile algorithm on top of tile layout storage. An original dependence translation layer (DTL) allows to map the access pattern of the second stage onto tile layout. The dynamic runtime system SMPSs [10] enables to schedule and overlap tasks generated from both stages. The obtained tile tridiagonal reduction significantly outperforms the state-of-the-art numerical libraries from medium to large matrix sizes.

The remainder of this paper is organized as follows: Section II gives a detailed overview of previous projects in this area. Section III lays out the new contributions of this paper. Section IV recalls the tridiagonal reduction approach used in the state-of-the-art numerical libraries. Section V describes the two-stage approach and Section VI presents some implementation details. Section VII presents performance results of the overall algorithm. Also, comparison tests are run on shared-memory architectures against the corresponding routines from LAPACK [11], ScaLAPACK [12], [13], SBR Toolbox [14] and the vendor library MKL [15]. Finally, Section VIII summarizes the results of this paper and describes the ongoing work.

II. Related Work

The two-stage approach for performing two-sided reductions is the de facto methodology in order to cast slow memory operations into fast compute intensive ones. Bischof et al. [14] developed a toolbox called Successive Band Reductions (SBR) to reduce a symmetric dense matrix to tridiagonal form, required to solve the symmetric eigenvalue problem (SEVP). This toolbox applies two-sided Householder transformations to the matrix and successively reduces the bandwidth until the tridiagonal form is reached. SBR relies heavily on multithreaded optimized BLAS to achieve parallel performance, which follows the expensive fork-join paradigm. More recently, Davis and Rajamanickam [16] implemented a similar toolbox called PIRO_BAND, which reduces symmetric and non-symmetric band matrices to tridiagonal and bidiagonal forms needed for the SEVP and the singular value decomposition, respectively. The orthogonal transformations are based on pipelined plane rotations, which improves the overall time to solution compared to SBR. Finally, Kågström et al. [17] described a two-stage approach in the context of Hessenberg-Triangular reduction for the generalized eigenvalue problem for dense matrix. The matrix is first reduced to band Hessenberg by applying accumulated Givens rotations within high performant kernels. The extra off-diagonal elements are then annihilated and chased down using the standard bulge chasing technique.

With the emergence of high bandwidth and high efficient devices such as GPUs, accelerating by orders of magnitude memory-bound and compute-bound operations becomes accessible. Tomov and Dongarra [18] presented a novel Hessenberg reduction algorithm, which takes advantage of the high bandwidth of the GPU by off-loading the expensive level 2 BLAS operations of the panel factorization to the device. The same approach is also applicable to the other two-sided transformations, i.e., the tridiagonal and bidiagonal reductions. The opposite was done by Bientinesi et al. [19] who accelerated the first stage (the reduction to *band* tridiagonal) of the SBR toolbox, which is the most compute intensive, by off-loading the compute-bound kernels to the GPU. The computation of the second stage (reduction to tridiagonal form) still remains on the host though.

III. Contributions

This section highlights the main contributions of our work.

- We have created a set of new Level 3 BLAS kernels that achieve high performance (see Section V-A). These efficient kernels are used during the first stage while reducing the tile matrix to band tridiagonal form. They are also reused to some extent during the second stage by proper recasting in terms of Level 2 BLAS kernels (see Section V-B).
- We have also devised a novel "bulge chasing" implementation for the second stage of the tridiagonal reduction. It works on top of either standard column-major layout or the standard tile layout when used in combination with the Dependence Translation Layer (DTL) (see Section VI-A).
- The memory bottleneck seen in the second stage, and expressed by previous approaches in Section II, is alleviated by implementing a novel *left-looking* bulge chasing procedure (on top of tile layout) rather than using a hardware solution (see Section VI-D). This allows us to drastically improve the data locality and to decrease the memory traffic within the multicore system.
- Representing algorithms using fine granularity tiles creates a tremendous number of tasks that can be dynamically executed in parallel in an out-of-order fashion. The common, strong synchronization point between both stages is then dramatically weakened. The fine granularity tasks are simply scheduled as soon as their data dependencies are satisfied and this is ensured through the DTL. As a result, the tasks from both stages can potentially overlap and our experiments show that they in fact do so. As a consequence, the upper left corner of the matrix attains its final tridiagonal form, while the lower right corner of the matrix is still being worked on by the first stage of the reduction.

IV. Background

This section describes the evolution of the algorithmic approaches in the dense linear algebra area. From block to tile algorithms, the state-of-the-art libraries had to undergo, at each time, a major redesign to better suit the new emerging architectures.

A. Block Algorithms

The LAPACK library provides a broad set of linear algebra operations aimed at achieving high performance on systems equipped with memory hierarchies. The algorithms implemented in LAPACK leverage the idea of blocking to limit the amount of bus traffic in favor of a high data reuse that is present in the higher level caches, which are also the fastest ones. The idea of blocking revolves around an important property of Level-3 BLAS operations (Matrix-Matrix multiplication), the so called surface-to-volume property, which states that ($\theta(n^3)$) floating point operations are performed on ($\theta(n^2)$) 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. Block algorithms consist of recasting linear algebra algorithms in a way that only a negligible part of computations is done in Level-2 BLAS operations (Matrix-Vector multiplication, where no data reuse possible) while most is done in Level-3 BLAS. Most of these algorithms 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.
- Trailing submatrix update : in this step, all the transformations that have been accumulated during the panel factorization step can be applied at once to the rest of the matrix (i.e., the trailing submatrix) by means of Level-3 BLAS operations.

Although the panel factorization can be identified as a sequential execution that represents a small fraction of the total number of performed FLOPS ($\theta(n^2)$) when compared with the total performed FLOPS ($\theta(n^3)$), the scalability of block factorizations is limited on a multicore system. Indeed, the panel factorization is rich in Level-2 BLAS operations that cannot be efficiently parallelized on currently available shared memory machines. Moreover, the parallelism is only exploited at the level of the BLAS routines.

This is even more critical for algorithms like two-sided transformations (namely Hessenberg, tridiagonal and bidiagonal reductions) where the panel computation is probably the most expensive one due to Level-2 BLAS operations accessing the entire matrix. As presented in LAPACK, those types of reductions follow a one-stage approach where the reduced form is obtained without intermediate steps. Later, the SBR toolkit introduced a

two-stage approach where the matrix is first reduced to a band form during a compute intensive phase and eventually to the final required form through a memory-bound phase. Although the performance numbers show some improvement (see Section VII), they are still far from the peak of the system.

All in all, block algorithms comply an inefficient fork-join model, since the execution flow of a block factorization represents a sequence of sequential operations (panel factorizations) interleaved with parallel ones (updates of the trailing submatrices).

B. Tile Algorithms

A solution to this fork-join bottleneck in block algorithms has been presented in [20]–[24]. The approach consists of breaking the panel factorization and trailing submatrix update steps into smaller tasks that operate on tiles i.e., a set of b contiguous columns where b is the block size (see Figure 1), in order to fit the small core caches. The algorithm can then be represented as a Directed Acyclic Graph (DAG) where nodes represent tasks, either panel factorization or update of a block-column, and edges represent dependencies among them.

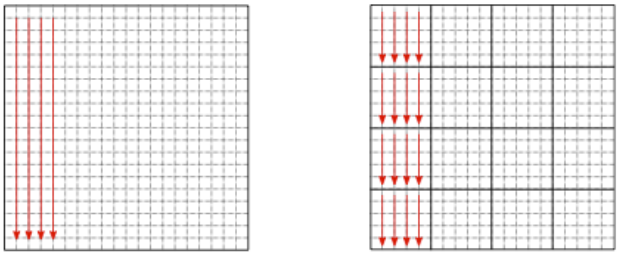


Figure 1. Translation from LAPACK Layout to Tile Data Layout

The execution of the algorithm is performed by asynchronously scheduling the tasks in a way such that dependencies are not violated. This asynchronous scheduling results in an out-of-order execution where slow, sequential tasks are hidden behind parallel ones. It is possible to achieve close to system peak performance for one-sided factorizations, i.e., Cholesky, QR and LU [25].

However, the integration of the new tile layout makes the redesign of the two-sided reductions from LAPACK very challenging [26]. As explained in the next section, an ambitious way to overcome this difficulty is to implement the two-stage approach for the tridiagonal reduction using tile algorithms.

V. A Two-Stage Tridiagonal Reduction using Tile Algorithms

A two-stage approach has demonstrated some potential in computing two-sided transformations in the context of block algorithms, but not enough to adequately address multicore architecture by itself. This section explains the challenges of executing the two-stage tridiagonal reduction on top of tile layout instead, where fine granularity and high degree parallelism can bring to the fore new opportunities to achieve high performance.

A. The First Stage: Reduction to Band Tridiagonal

The idea of this first stage is straightforward. The original symmetric matrix is reduced to a temporary band tridiagonal form consisting of b off-diagonals, with b the tile size.

This stage is highly compute-intensive, relying on new optimized kernels as well as kernels coming from the tile QR reduction [21].

The descriptions of the tile QR kernels are as follow:

- DGEQRT computes the QR factorization of a sub-diagonal tile. This kernel generates reflectors in the lower part of the sub-diagonal tile (Figure 2(a)).
- DLARFB takes as input the block of reflectors computed from DGEQRT and updates the corresponding tile row (left update, see Figure 2(b)) and column (right update, see Figure 2(c)).
- DTSQRT stacks an upper triangular tile on top of a square tile and computes the QR factorization. It also generates reflectors and stores them in place of the square tile (Figure 2(d)).
- DSSRFB applies the square block of reflectors from DTSQRT to the two tile rows (left updates, see Figure 2(e)) and columns (right updates, see Figure 2(k)).

DGEQRT and DTSQRT handle basically the panel factorization and DLARFB and DSSRFB apply the generated reflectors to the trailing submatrix. More details about those kernels can be found in the literature [21].

The newly implemented kernels appropriately handle the symmetric structure of the matrix. They are described below:

- DSSRFBLR and DSSRFBLRT kernels apply from the left the block of reflectors calculated in DTSQRT by rigorously taking into account the symmetric structure of the matrix (Figure 2(h) and Figures 2(f)-2(g), respectively).

- DSSRFBRL and DSSRFBRLT kernels apply from the right the block of reflectors calculated in DTSQRT by also taking into account the symmetric structure of the matrix (Figure 2(i) and Figure 2(j), respectively).

Those special kernels need temporary buffers to handle the symmetry accordingly and to ensure numerical correctness (Atmp1, Atmp2 and Atmp3) as shown in the overall band TRD algorithm for an NT-by-NT tile matrix (Algorithm 1). The tile band TRD algorithm (two-sided) can be even better characterized conceptually. It is similar to some extent to the tile QR factorization (one-sided). The only difference is that, as soon as the left updates hit the diagonal tiles, they *are reflected* by a 90 degree angle in order to avoid touching the upper matrix. Then, the *reflected* updates are actually the right updates and they continue applying the block of reflectors until the bottom of the matrix is reached. Those *reflected* updates can be seen from Figure 2(f) to Figure 2(j).

Algorithm 1 Tile Band TRD Algorithm with Householder Reflectors.

```
1: for step = 1, 2 to NT−1 do
2:   DGEQRT(A_{step+1,step})
3:   {Left Updates}
4:   DLARFB(A_{step+1,step}, A_{step+1,step+1})
5:   for i = step+1 to NT do
6:     {Right Updates}
7:     DLARFB(A_{step+1,step}, A_{i,step+1})
8:   end for
9:   for k = step+2 to NT do
10:    DTSQRT(A_{step+1,step}, A_{k,step})
11:    {Left Updates}
12:    for j = step+1 to k do
13:      if (j == step+1) then
14:        DSSRFB(A_{step+1,j}, A_{k,j}, Atmp1)
15:      else if (j == k) then
16:        DSSRFBLR(Atmp1, A_{k,j}, Atmp2)
17:      else
18:        DSSRFBLRT(A_{j,step+1}, A_{k,j})
19:      end if
20:    end for
21:    {Right Updates}
22:    for m = step+1 to NT do
23:      if (m == step+1) then
24:        DSSRFB(Atmp2, Atmp3, A_{k,m})
25:      else if (m == k) then
26:        DSSRFBRL(Atmp3, A_{k,m})
27:      else if (m > k) then
28:        DSSRFBRLT(A_{m,step+1}, A_{k,m})
29:      end if
30:    end for
31:  end for
32: end for
```

B. The Second Stage: Reduction to Tridiagonal using the Bulge Chasing Technique

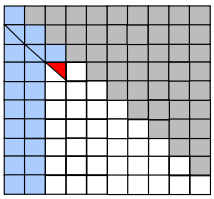
This second stage takes as input a band tridiagonal matrix in LAPACK layout (column-major) and reduces it to the required and final tridiagonal form. The procedure used in this stage is the standard bulge chasing. It consists in annihilating the extra elements column-by-column. For each column annihilated, there is a bulge or a block of fill-in elements created, which needs to be chased down toward the bottom right corner of the matrix. This is what is called a sweep. So, if n is the matrix size, there will be $n-2$ sweeps. These Level 2 BLAS kernels are actually derived from the high performance kernels described in Section V-A. The Level 3 BLAS kernels have been properly unrolled in terms of Level 2 BLAS kernels for better performance, given that the operations are really performed on a small amount of data. The general bulge chasing procedure is presented in Algorithm 2.

Algorithm 2 Standard Bulge Chasing Algorithm of b extra diagonals.

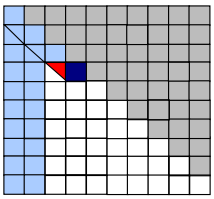
```
1: for k = 1, 2 to N−2 do
2:   {Column annihilation}
3:   DGEQR2(B_{k+1:k+min(b,n−k), k})
4:   {Left and right updates on a diagonal block}
5:   DSYRF(B_{k+1:k+min(b,n−k), k+1:k+min(b,n−k)})
6:   {Chasing the bulge}
7:   for i = k+1 to N with i = +b do
8:     if i+b == n then
9:       {Right update for clean-up}
10:      DLARFX(B_{min(i+b,n), i:min(i+b,n)−1})
11:    end if
12:    if min(i+b,n) < (n−1) || min(i+2*b,n) < (n−1) then
13:      {Column annihilation within the sweep}
14:      DGEQR3(B_{min(i+b,n):min(i+2*b,n)−1, i:min(i+b,n)−1})
15:      {Left and right updates on a diagonal block}
16:      DSYRF(B_{min(i+b,n):min(i+2*b,n)−1, min(i+b,n):min(i+2*b,n)−1})
17:    end if
18:  end for
19: end for
```

We recall the four kernels involved in the standard bulge chasing algorithm:

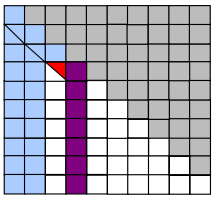
- DGEQR2 generates a single Householder reflector.



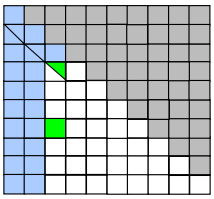
(a) DGEQRT (Panel).



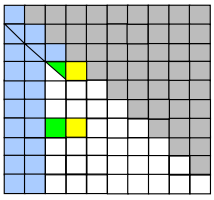
(b) DLARFB (Left update).



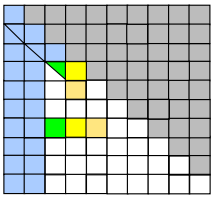
(c) DLARFB (Right update).



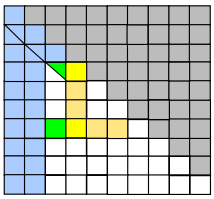
(d) DTSQRT (Panel).



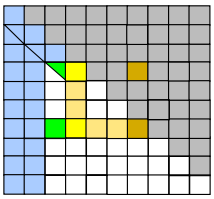
(e) DSSRFB (Left update).



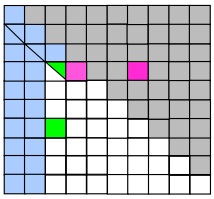
(f) DSSRFBLRT (Reflected update).



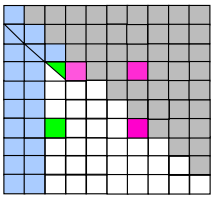
(g) DSSRFBLRT (Reflected update).



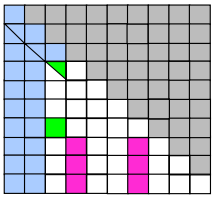
(h) DSSRFBLR (Reflected update).



(i) DSSRFBRL (Reflected update).



(j) DSSRFBRLT (Reflected update).



(k) DSSRFB (Right update).

Figure 2. Execution Breakdown of the Tile TRD Band Reduction at step 3 for a 10-by-10 tile matrix. The light blue part corresponds to the matrix already factorized in band tridiagonal form. The gray part is the untouched area of the original matrix

- DSYRF applies the reflector computed in DGEQR2 to a symmetric diagonal block from the left and right.
- DGEQR3 contains three successive computation steps executed on a data block, which could physically span on four tiles at most. It applies the reflectors calculated in DGEQR2 from the right. It then annihilates the first column of the created bulge and finally applies the new reflectors to the left.
- DLARFX applies the reflectors computed in DGEQR3 from the right during the clean-up phase.

There are mainly two critical issues with this technique that need to be addressed. First, there is an existing mismatch between the original layout (i.e., LAPACK) and the tile layout. Second, the operations involved are completely memory-bound. This engenders significant overheads due to high memory traffic and will eventually occupy the overwhelm bus prohibiting any parallel execution [26].

In the following section, we explain how those issues can be relieved.

VI. Implementation Details

A. Dependence Translation Layer

The first stage of our tridiagonal reduction can readily use standard tile layout, with the first tile occupying the upper left corner of the matrix; the required computational kernels can be reused from the one-sided factorizations. On the other hand, the second stage cannot be easily formulated to obey the standard tile layout; it has a very particular access pattern that most of the time spans the boundaries of the standard tile layout. Figure 3 shows which portion of the matrix is accessed by each of the 25 tasks of the second stage of reduction of a 9×9 matrix with tile size equal 3. The very first operation of the second step is the first column annihilation; the accessed portion of the matrix is marked with a bright red rectangle of dimension 3×1 in the upper left corner of the Figure. This first task already spans two tiles. The next task accesses a portion of the matrix that spans 4 tiles as marked by a brown square of size 3×3. There are some tasks that access only two tiles or even a single tile. Such irregularity of data accesses would be very hard to accommodate programmatically in a standard tile layout, or any tile layout for that matter, as the boundaries of the accessed portions of the matrix move by one at each iteration of the second stage. One solution

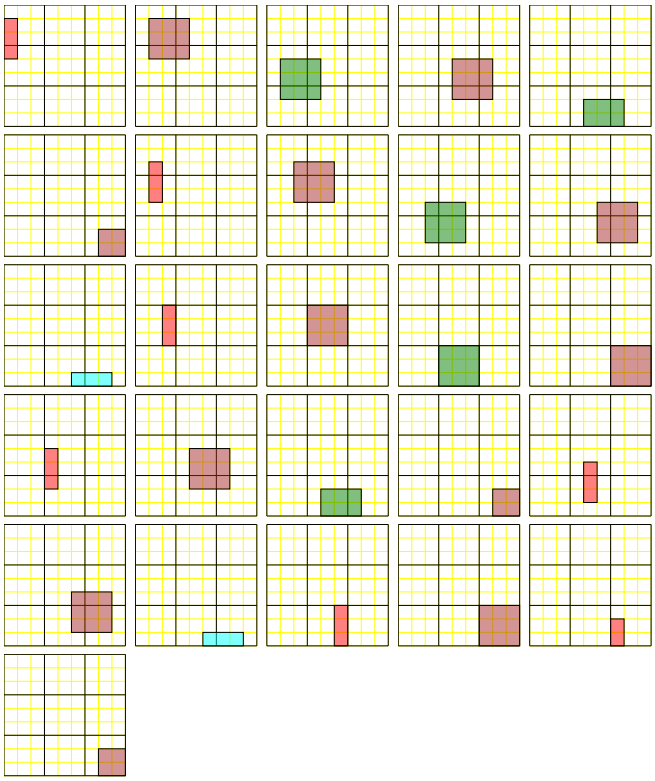


Figure 3. Graphical representation of portions of the matrix accessed by the consecutive task of the second stage of the reduction. The yellow lines represent division of the matrix into individual entries and the long black lines delineate matrix tiles in the first stage of the tridiagonal reduction and submatrices accessed in the second stage. The red tasks represent the DGEQR2 kernel, the brown tasks identify the DSYRF kernel, the green tasks shows the DGEQR3 kernels and finally, the blue tasks are the DLARFX kernels.

could be to translate the matrix from tile storage to a standard column-major storage (as it is used by LAPACK and Fortran) but then we face the performance penalty of translation and the hardship of parallelizing the code by hand, as the current DAG schedulers only address tile storage format. We chose to combine the advantages of both approaches (use our existing tile formulation on top of column-major storage and take advantage of automatic parallelization) by introducing a Dependence Translation Layer (DTL).

The translation layer bridges the column-major layout with the standard tile storage and creates proper dependencies that could be fed to either the SMPSs or QUARK runtimes. Our original code that works on top of column-major storage remains unchanged except for names of the functions that implement the computational kernels; instead of the kernels, wrapper functions are called, which perform the translation. Each wrapper has two parts. The first part extracts the information as to which and what portions of tiles are being accessed while the second part schedules the appropriate kernel for execution by the DAG runtime scheduler. The first part of each wrapper in the translation layer is common to all kernels and could be reused throughout the code – a win from the software engineering stand point. The second part of each wrapper is unique and requires manual intervention – a small price to pay considering the daunting task of rewriting the whole second stage.

The visualization of the second stage tasks from Figure 3 is a very useful tool in developing the DTL. Let's use 9×9 matrix from the upper left corner of the Figure. It shows the annihilation of the first column of the band tridiagonal form; it spans two tiles. The DTL will take this information and schedule for execution a kernel that can generate a Householder reflector suitable for annihilation split across tiles. The DTL will also mark the two tiles as being read and overwritten in order for the runtime to correctly account for data dependencies and preserve the proper order of execution.

B. Tile Bulge Chasing Procedure

After integrating the DTL framework into the second stage, the original bulge chasing kernels presented in Section V-B need to be broken into smaller successive tasks to match the area spanning across the physical tiles. This is part of the second step explained in the Section VI-A. As shown in Figure 3, there are different cases to consider depending on this area, and for each case, a particular kernel is required.

Below are some details about the new kernels:

- DGEQR2 (red in Figure 3) becomes DTSQR2 when the single column to be annihilated resides on two tiles and stays as is if the column fits in a single tile.
- DSYRF (brown in Figure 3) applies the reflectors computed in DGEQR2 to a symmetric diagonal tile from the left and right, if the targeted area fits in a single tile. Also, DSYRF applies the reflectors computed in DTSQR2 to a symmetric diagonal tile from the left and right if the region extends across four tiles.
- DGEQR3 (green in Figure 3) applies successively from the right the reflectors calculated in DGEQR2 or DTSQR2, depending on whether the area fits in one or two/four tiles, respectively. It then annihilates the first column of the bulge by calling DGEQR2/DTSQR2 and applies those freshly

created reflectors to the left within the tile by executing DLARFX or DSSRFX, respectively.
- DLARFX (cyan in Figure 3) applies the reflectors from DTSQR2, as it *always* spans across two tiles.

The tile bulge chasing technique makes the DAGs for two-sided factorizations much more complex with a number of nodes/tasks growing exponentially with the matrix size. The next section describes SMPSs framework, a runtime system for task scheduling across homogeneous multicore architectures.

C. Runtime System: SMPSs/QUARK

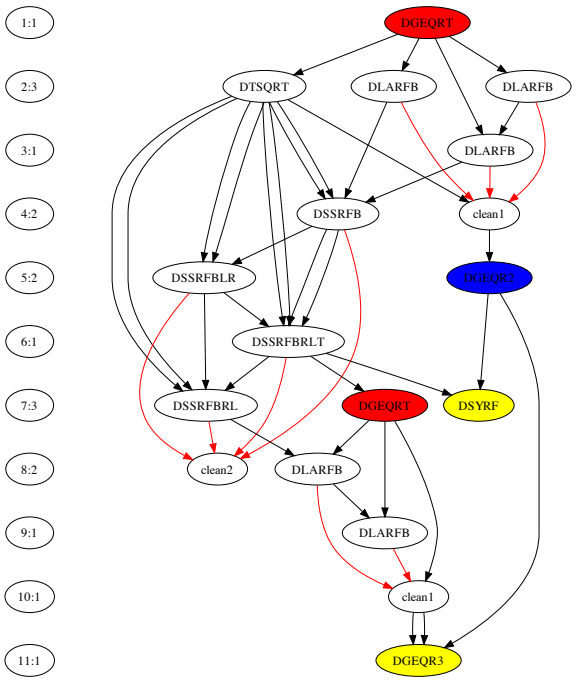


Figure 4. DAG for tridiagonal reduction: yellow nodes belong to the second stage and the blue node marks the beginning of the second stage. The ovals on the left denote the step and the number of tasks that can be executed in parallel in that step.

DAGs have a long history [27] of expressing parallelism and task dependencies in distributed systems. Previously, they have often been used in grids and peer-to-peer systems to schedule large grain tasks, mostly from a central coordinator organizing the different task executions and data movements. A whole taxonomy of DAGs have been used in grid environments [28], [29].

More recently, many projects have proposed to use them as an approach to address the challenge of harnessing the computing potential of multicore computers, especially in the linear algebra field. It has been demonstrated that DAGs enable the scheduling of tasks for tile algorithms on multi-core CPUs [30], [31], reaching performances inaccessible to traditional approaches for the same problem sizes. Such an approach can also be used to address hybrid architectures [32], with computers augmented with accelerators such as GPUs. Using a task description language to define codelets enables the execution of same tasks on different hardware [33], and DAGs may be used to schedule tasks on heterogeneous computers [34].

There are at least three approaches to building and managing the DAG during the execution. (1) First read a concise representation of the DAG (in XML) and unroll it in memory before scheduling it [29]. (2) Modify the sequential code with pragmas to isolate tasks that will be run as an atomic entity and run the sequential code to discover the DAG [10], [32], [35]. Optionally, these execution engines use bounded buffers of tasks to limit the impact of the unrolling operation in memory. (3) The third approach consists of using the concise representation of the DAG in memory, to avoid most of the impact of unrolling it at runtime. Using structures like Parameterized Task Graph (*PTG*) [36], the memory used for DAG representation is linear in the number of task types and totally independent of the total number of tasks.

Clearly, using a DAG for scientific computations is not new. The novelty of our approach lies in the use of DAG scheduling for a two-sided factorization. A DAG for a small 3×3 matrix is shown in Figure 4. Because we apply a DAG for both stages of the tridiagonal reduction, the nodes in the DAG do not represent tasks of equal computational load. The white and red nodes represent the first stage, the reduction to band tridiagonal form, and are rich in Level 3 BLAS operations; they operate close to peak performance of the hardware. The blue node begins the second stage, the reduction to tridiagonal form, which is carried out by the yellow nodes; they are by far bound by the available memory bandwidth. This presents a potential challenge for DAG schedulers; the first stage is, to an extent, oblivious to data locality, while for the second stage data locality is of the utmost importance. We conducted our tests with both SMPSs and QUARK (part of PLASMA) schedulers and found that indeed data locality is the most

crucial aspect that can help performance of the reduction.

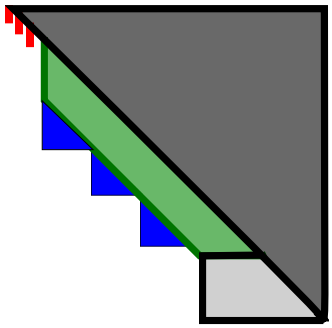


Figure 5. Overlapping between the two stages: the dark gray area remain untouched since the matrix is symmetric. The red elements belong to the final tridiagonal structure. The green band still needs to be reduced after chasing down the blue bulges (second stage tasks). The light gray area is the unreduced part (first stage tasks).

By overlapping the execution of the first and second stage of the reduction, we are able to have about 10% of the matrix reduced to tridiagonal form by the time the first stage finishes. The overlap may be observed in Figure 4, as the second stage begins with a blue node at step 5 and the first stage doesn't end until step 10. Figure 5 presents another way to show how the overlapping actually happens. The Figure represents the physical state of the matrix at a given time during the reduction. The upper triangular (dark gray) remains untouched due to the symmetric structure of the matrix. The top left corner of the matrix (red) has already reached the tridiagonal form. The middle of the matrix contains the band form (green) which will eventually be reduced after chasing down the fill-in elements (blue). The bottom right of the matrix (light gray) is still unreduced. The difficult part of overlapping the stages is formulating the second stage reduction in tile form. The execution DAG engines perform the dependence tracking and parallel scheduling that lead to the overlap.

D. Looking Variants

One thing that a DAG of tasks does not convey is which variant of a given algorithm it represents: left-looking or right-looking [37]. The DAG is the same for either variant and it is the order of visiting the DAG's nodes during execution that determines which variant is used. In essence, it is up to the scheduling engine to choose the node visitation order and thus pick the variant. Left-looking variants of many linear algebra algorithms are known to possess better data locality properties which is essential for the second stage of the tridiagonal reduction. However, it is conceptually much easier to code the right-looking variant. First annihilate the first column and chase the resulting fill-in (bulge) down to the bottom right corner of the matrix, then proceed with the second column and so on. Naturally then, we started our experiments with the right-looking variant and had hoped for the DAG scheduling runtime to help in introducing data locality. Note that locality-aware scheduling requires looking ahead in the stream of tasks and picking those for execution whose data is still in cache – the right-looking variant discards data from cache quickly and moves down to the bottom right of the matrix without any reuse. By means of profiling and tracing, we were able to observe inefficiency in scheduling; the tasks were taking longer than expected, which indicated to us a higher rate of cache misses resulting from the lack of locality. We concluded that the insertion order of tasks into the DAG scheduling engine is important and thus proceeded with implementing the left-looking variant of the second stage of the reduction. Due to complicated data dependencies, deriving the left-looking variant from the right-looking one is not as straightforward as interchanging the loops. However, from an understanding of the algorithm and by looking at Figure 3, we derive the left-looking variant by changing the function that implements the right-looking variant into a generator, i.e., a function that returns multiple values. Our generator returns tasks to be scheduled by the DAG runtime, and we keep a number of generators active as there are columns to annihilate. The generator for annihilation of column $k+1$ is allowed to produce a task if the generator for column k is at least two steps behind it in the number of tasks it has produced. This simple rule allowed for a productive implementation of the left-looking variant and resulted in performance improvement, as shown later in Section VII-D.

VII. Experimental Results

A. Hardware Description

The experiments have been performed on a quad-socket quad-core machine based on an Intel Xeon EMT64 E7340 processor operating at 2.4 GHz. The theoretical peak is equal to 9.6 Gflop/s per core or 153.2 Gflop/s for the whole node, which is composed of 16 cores. There are two levels of cache. The Level 1 cache, local to each core, is divided into 32 KB of instruction cache and 32 KiB of data cache. Each quad-core processor is composed of two dual-core Core2 architectures, the Level 2 cache has 2×4 MB per socket (each

dual-core shares 4 MB). The effective bus speed is 1066 MHz per socket leading to a bandwidth of 8.5 GB/s (per socket). The machine was running Linux 2.6.25 and provided Intel Compilers 11.0 together with the MKL 10.1 vendor library. All the experiments presented below focus on asymptotic performance and were conducted on the maximum amount of cores available on the machine, i.e., 16 cores.

B. Software Description

There is a number of software packages that implement tridiagonal reduction. For comparison we used as many as we were aware of, and here we briefly describe each one in turn.

LAPACK [11] is a library of Fortran 77 subroutines for solving those most commonly occurring problems in dense matrix computations. It has been designed to be efficient on a wide range of modern high-performance computers. The name LAPACK is an acronym for Linear Algebra PACKage. LAPACK can solve systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems. LAPACK can also handle many associated computations, such as matrix factorizations or estimating condition numbers.

ScaLAPACK [12], [13] is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers and networks of workstations supporting PVM [38] and/or MPI [39]–[42]. It is a continuation of the LAPACK project, which designed and produced analogous software for workstations, vector supercomputers, and shared-memory parallel computers. The other extension to LAPACK is that ScaLAPACK uses a two-dimensional block cyclic distribution, which improves the memory locality.

MKL (Math Kernel Library) [15] is commercial software from Intel that is a highly optimized programming library. It includes a comprehensive set of mathematical routines implemented to run well on multicore processors. In particular, MKL includes a LAPACK-equivalent interface that allows for easy swapping the reference LAPACK implementation for the MKL one by simply changing the linking parameters. By the same token, the SBR Toolbox interface is also available in MKL. We tested both versions 10 and 11 of MKL and found negligible difference in performance of the routines relevant for this document.

C. Tuning

The performance of *tile algorithms* strongly depends on tunable execution parameters of the outer and the inner blocking sizes [25]. In the first stage of the tridiagonal reduction, the outer block size, nb, trades off parallelization granularity and scheduling flexibility for single core utilization. The inner block size, ib, trades off memory load with extra flops due to redundant calculations. The second stage requires tuning of only the outer block size – nb. Large values of nb tend to decrease opportunities for increase in data locality. And small values of nb increase the overhead due to kernel startup cost. Both stages are connected because they both use the same data layout, which determines the value of nb. This creates an unusually large set of constraints on the optimal value of nb. We have established experimentally the optimal range for nb values to be between 100 and 200 with only slight changes in the resulting performance. As exhaustive tuning is beyond the scope of this paper, we simply chose value 100 for nb and 25 for ib.

D. Performance

Figure 6 draws the performance comparisons between the right and left looking variants. As described in Section VI-D, the right looking variant requires the traverse of the whole diagonal band of the matrix to annihilate a single column. In other words, the whole matrix has to be loaded in cache to annihilate a single column, which can not be an option.

On the contrary, the left looking variant allows for making sure *all* subsequent updates from column annihilations are applied at once to an area around the diagonal before sliding to the next area. This results in improved cache reuse, which is paramount when it comes to enhancing memory-bound operations. And in terms of performance numbers, this is expressed by a gain of 10% to 15% compared to the right looking variant.

Figure 7 shows the performance comparisons of the tile TRD left looking variant against the state-of-the-art numerical libraries. LAPACK/ScaLAPACK TRD and MKL TRD use a one-stage approach to reduce a general dense matrix to tridiagonal form. The standard SBR TRD implements a two-stage approach, which requires the change of the layout in-between, going from column-major to band layout. MKL SBR TRD is the vendor optimized version of the SBR TRD.

The performance for LAPACK TRD is very low, around 2.5 Gflop/s. This again proves the scala-

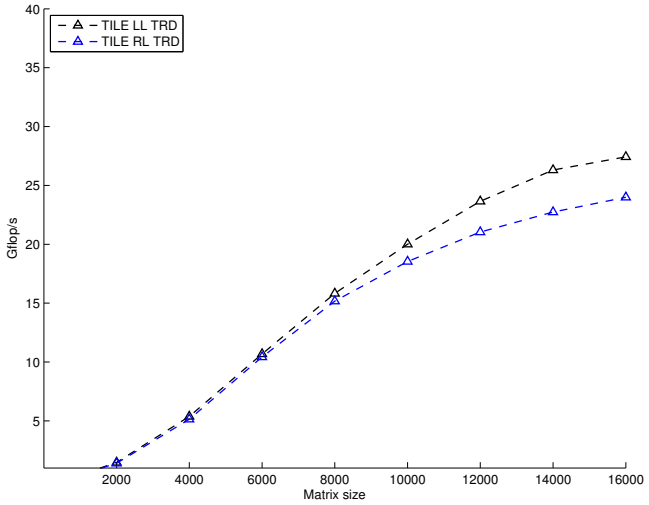


Figure 6. Right Looking and Left Looking variants of the Tile TRD.

bility limitations of block algorithms on multicore architectures. Thanks to a better memory locality, ScaLAPACK TRD performs better for small sizes but then the performance does not scale as the matrix size increases. MKL TRD presents the same performance behavior. The two-stage approach implemented in SBR TRD only scales for small matrix sizes, similar to the one-stage approach. There are mainly three reasons: (1) the first stage follows the strategies of block algorithms with parallelism occurring at the level of the BLAS, (2) the transformation of the layout between both stages and (3) the right looking variant of the bulge chasing procedure. In particular, (3) really becomes a bottleneck when the matrix gets larger. However, the vendor optimized MKL SBR TRD scales for large matrix sizes. The second stage has been most probably enhanced.

Our tile LL TRD approach is clearly not suited for small matrix sizes. And this can be mainly explained by the overhead of chasing the bulge (second stage) on top of tile layout. The real gain over most of the other libraries happens around the crossover point $n = 6000$, where the first stage exhibits very good performance thanks to the tile algorithm paradigm. Our approach continues to scale as the matrix gets larger thanks to the left looking variant of the second stage. The possible overlap between both stages (up to 10%) also helps toward that direction. For $n = 16000$, the tile LL TRD runs roughly at 28 Gflop/s, which is 10 times faster than multithreaded LAPACK with optimized MKL BLAS and 2.5 times faster than the vendor optimized MKL SBR TRD.

Although the system has only 16 cores, we envision this two-stage approach to be very scalable as the number of cores increases because it is really optimized for memory accesses.

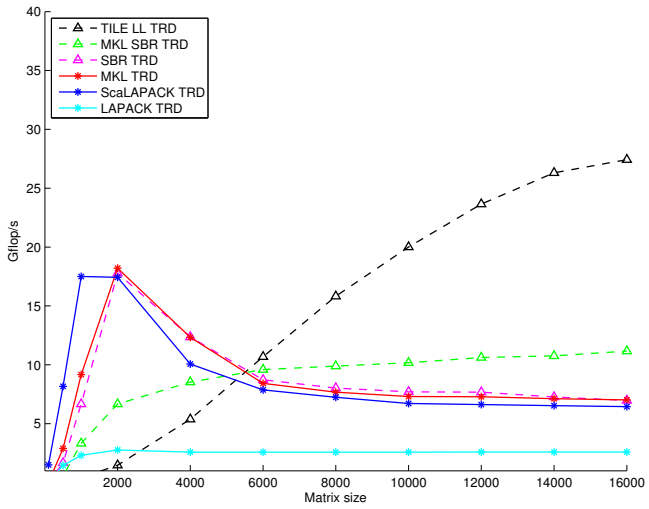


Figure 7. Tile TRD performance comparisons on Intel Xeon 2.4GHz with MKL Blas 10.1

VIII. Summary

This paper describes a novel implementation, which enables efficient reduction of a symmetric matrix to the tridiagonal form – the first algorithmic step toward computing one of the most important linear algebra algorithms i.e., the spectral decomposition of a dense symmetric matrix. Using a two-stag approach, this implementation is particularly well suited for homogeneous multicore architectures. It relies on the tile formulation of the reduction algorithm and a combination of performance enhancing techniques: (1) a left-looking variant of the bulge chasing procedure that enhances data locality and reduces memory traffic, (2) a dependence translation layer that eases the burden of data layout translation or reprogramming of existing code, and (3) a dynamic runtime system which allows for scheduling and overlapping tasks generated from both reduction stages. The combined effect of these factors allows our implementation to exhibit weak scaling (currently hard to achieve) and vastly outperforms the state-of-the-art numerical libraries (an up to 10-fold improvement) for medium to large matrix sizes. Due to the memory-bound nature of the algorithm, it is noteworthy to mention that the overall performance achieved still represents only a relatively small fraction of the theoretical peak of the machine (20%).

The authors are currently investigating how the Q matrix, which contains all the accumulated transformations from both stages, may be efficiently computed. This is a relevant issue because the Q matrix is required during the back transformation step that is performed in the case when the eigenvectors are needed.

Given the significance of the eigenvalue problem to various areas of science, a straightforward extension to this work will be the generalized symmetric eigenvalue problem. To accommodate this, another stage will have to be implemented, which necessitates the integration of the Cholesky factorization. This new stage can be seen as a preprocessing step to the two-stage approach presented in this paper. These three stages can then be overlapped thanks to the dependence translation layer and the dynamic runtime system.

Finally, the authors are also looking at how this work can be extended to address the Hessenberg and bidiagonal reductions. Preliminary results have already been obtained for the first stage of each of the reductions [43].

References

[1] H. W. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon, *TOP500 Supercomputer Sites*, 35th ed., June 2010, (The report can be downloaded from http://www.netlib.org/benchmark/top500.html).

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," Electrical Engineering and Computer Sciences University of California at Berkeley,, Tech. Rep. Technical Report No. UCB/EECS-2006-183, December 18 2006.

[3] *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Archtectures, Version 2.2*, University of Tennessee, November 2009.

[4] "The FLAME project," April 2010, http://z.cs.utexas.edu/wiki/flame.wiki/FrontPage.

[5] G. Golub and C. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

[6] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. Philadelphia, PA: SIAM, 1997, http://www.siam.org/books/OT50/Index.htm.

[7] N. Rösch, S. Krüger, V. Nasluzov, and A. Matveev, "ParaGauss: The Density Functional Program ParaGauss for Complex Systems in Chemistry," in *High Performance Computing in Science and Engineering, Garching 2004, part III*. Springer, 2005, pp. 285–296, dOI: 10.1007/3-540-28555-5-25.

[8] R. Grimes, H. Krakauer, J. Lewis, H. Simon, and S.-H. Wei, "The solution of large dense generalized eigenvalue problems on the Cray X-MP/24 with SSD," *J. Comput. Phys.*, vol. 69, pp. 471–481, April 1987. [Online]. Available: http://portal.acm.org/citation.cfm?id=32855.32865

[9] R. M. Martin, "Electronic structure: Basic theory and practical methods," *Cambridge University Press*, 2008.

[10] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Cluster Computing, 2008 IEEE International Conference on*, 29 2008-oct. 1 2008, pp. 142 –151.

[11] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen, *LAPACK User's Guide*, 3rd ed. Philadelphia: Society for Industrial and Applied Mathematics, 1999.

[12] L. S. Blackford, J. Choi, A. Cleary, E. F. D'Azevedo, J. W. Demmel, I. S. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia: Society for Industrial and Applied Mathematics, 1997.

[13] J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. C. Whaley, "The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines," *Scientific Programming*, vol. 5, pp. 173–184, 1996.

[14] C. H. Bischof, B. Lang, and X. Sun, "Algorithm 807: The SBR Toolbox—software for successive band reduction," *ACM Trans. Math. Softw.*, vol. 26, no. 4, pp. 602–616, 2000.

[15] "Intel, Math Kernel Library (MKL)," http://www.intel.com/software/products/mkl/.

[16] T. A. Davis and S. Rajamanickam, "PIRO BAND, Pipelined Plane Rotations for Blocked Band Reduction."

[17] B. Kågström, D. Kressner, E. Quintana-Orti, and G. Quintana-Orti, "Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisited," *BIT Numerical Mathematics*, vol. 48, pp. 563–584, 2008.

[18] S. Tomov, R. Nath, and J. Dongarra, "Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing," *Parallel Computing*, vol. DOI information: 10.1016/j.parco.2010.06.001, 2010.

[19] P. Bientinesi, F. Igual, D. Kressner, and E. Quintana-Orti, "Reduction to Condensed Forms for Symmetric Eigenvalue Problems on Multi-core Architectures," *Parallel Processing and Applied Mathemetics*, pp. 387–395, 2010.

[20] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, "Parallel Tiled QR Factorization for Multicore Architectures," *Concurrency Computat.: Pract. Exper.*, vol. 20, no. 13, pp. 1573–1590, 2008, http://dx.doi.org/10.1002/cpe.1301 DOI: 10.1002/cpe.1301.

[21] ——, "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures," *Parellel Comput. Syst. Appl.*, vol. 35, pp. 38–53, 2009, http://dx.doi.org/10.1016/j.parco.2008.10.002 DOI: 10.1016/j.parco.2008.10.002.

[22] E. S. Quintana-Ortí and R. A. van de Geijn, "Updating an LU Factorization with Pivoting," *ACM Trans. Math. Softw.*, vol. 35, no. 2, p. 11, 2008, http://doi.acm.org/10.1145/1377612.1377615 DOI: 10.1145/1377612.1377615.

[23] J. Kurzak, A. Buttari, and J. J. Dongarra, "Solving systems of linear equation on the CELL processor using Cholesky factorization," *Trans. Parallel Distrib. Syst.*, vol. 19, no. 9, pp. 1175–1186, 2008, http://dx.doi.org/10.1109/TPDS.2007.70813 DOI: TPDS.2007.70813.

[24] J. Kurzak and J. J. Dongarra, "QR factorization for the CELL processor," *Scientific Programming*, vol. 17, pp. 1–12, 2008, http://dx.doi.org/10.3233/SPR-2008-0268 DOI: 10.3233/SPR-2008-0268.

[25] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra, "Comparative study of one-sided factorizations with multiple software packages on multicore hardware," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[26] H. Ltaief, J. Kurzak, and J. Dongarra, "Parallel block hessenberg reduction using algorithms-by-tiles for multicore architectures revisited," *UT-CS-08-624 (also LAPACK Working Note 208)*, 2008.

[27] J. A. Sharp, Ed., *Data flow computing: theory and practice*. Ablex Publishing Corp, 1992.

[28] J. Yu and R. Buyya, "A Taxonomy of Workflow Management Systems for Grid Computing," *Journal of Grid Computing*, 2005.

[29] O. Delannoy, N. Emad, and S. Petiton, "Workflow global computing with yml," in *7th IEEE/ACM International Conference on Grid Computing*, september 2006.

[30] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, "The Impact of Multicore on Math Software," in *Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA*, ser. Lecture Notes in Computer Science, B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, Eds., vol. 4699. Springer, 2006, pp. 1–10.

[31] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 116–125.

[32] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. 180, 2009.

[33] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A hybrid multi-core parallel programming environment," in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[34] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par 2009 Euro-par'09 Proceedings*, ser. LNCS, Delft Pays-Bas, 2009. [Online]. Available: http://hal.inria.fr/inria-00384363/en/

[35] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11, http://doi.acm.org/10.1145/1654059.1654079 DOI: 10.1145/1654059.1654079.

[36] M. Cosnard and E. Jeannot, "Automatic Parallelization Techniques Based on Compact DAG Extraction and Symbolic Scheduling," *Parallel Processing Letters*, vol. 11, pp. 151–168, 2001. [Online]. Available: http://dx.doi.org/10.1142/S012962640100049X http://hal.inria.fr/inria-00000278/en/

[37] Q. Yi, K. Kennedy, H. You, K. Seymour, and J. Dongarra, "Automatic blocking of QR and LU factorizations for locality," in *2nd ACM SIGPLAN Workshop on Memory System Performance (MSP 2004)*, 2004.

[38] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: MIT Press, 1994.

[39] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. J. Dongarra, *MPI: The Complete Reference*. Cambridge, MA: MIT Press, 1996.

[40] M. P. I. Forum, "MPI: A Message-Passing Interface Standard," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 8, 1994.

[41] ——, "MPI: A Message-Passing Interface Standard (version 1.1)," 1995, available at: http://www.mpi-forum.org/.

[42] ——, "MPI-2: Extensions to the Message-Passing Interface," 18 Jul. 1997, available at http://www.mpi-forum.org/docs/mpi-20.ps.

[43] H. Ltaief, J. Kurzak, J. Dongarra, and R. M. Badia, "Scheduling two-sided transformations using tile algorithms on multicore architectures," *Sci. Program.*, vol. 18, no. 1, pp. 35–50, 2010.