High-Performance Bidiagonal Reduction using Tile Algorithms on Homogeneous Multicore Architectures

HATEM LTAIEF, KAUST Supercomputing Laboratory
PIOTR LUSZCZEK, University of Tennessee
JACK DONGARRA, University of Tennessee, Oak Ridge National Laboratory, and University of Manchester

This article presents a new high-performance bidiagonal reduction (BRD) for homogeneous multicore architectures. This article is an extension of the high-performance tridiagonal reduction implemented by the same authors [Luszczek et al., IPDPS 2011] to the BRD case. The BRD is the first step toward computing the singular value decomposition of a matrix, which is one of the most important algorithms in numerical linear algebra due to its broad impact in computational science. The high performance of the BRD described in this article comes from the combination of four important features: (1) tile algorithms with tile data layout, which provide an efficient data representation in main memory; (2) a two-stage reduction approach that allows to cast most of the computation during the first stage (reduction to band form) into calls to Level 3 BLAS and reduces the memory traffic during the second stage (reduction from band to bidiagonal form) by using high-performance kernels optimized for cache reuse; (3) a data dependence translation layer that maps the general algorithm with column-major data layout into the tile data layout; and (4) a dynamic runtime system that efficiently schedules the newly implemented kernels across the processing units and ensures that the data dependencies are not violated. A detailed analysis is provided to understand the critical impact of the tile size on the total execution time, which also corresponds to the matrix bandwidth size after the reduction of the first stage. The performance results show a significant improvement over currently established alternatives. The new high-performance BRD achieves up to a 30-fold speedup on a 16-core Intel Xeon machine with a 12000×12000 matrix size against the state-of-the-art open source and commercial numerical software packages, namely LAPACK, compiled with optimized and multithreaded BLAS from MKL as well as Intel MKL version 10.2.

Categories and Subject Descriptors: G.4 [**Mathematics of Computing**]: Mathematical Software—*Parallel and vector implementations*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Bidiagional reduction, tile algorithms, two-stage approach, bulge chasing, data translation layer, high performance kernels, dynamic scheduling

This research reported here was supported in part by the National Science Foundation, the Department of Energy, and Microsoft Research.
Authors' addresses: H. Ltaief, Kaust Supercomputing Laboratory, Thuwal, Saudi Arabia; email: Ltaief@kaust.edu.sa; P. Luszczek and J. Dongarra, Innovative Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996; email: dongarra@eecs.utk.edu; and Luszczek@eecs.uth.edu.

DOI: http://dx.doi.org/10.1145/2450153.2450154

ACM Reference Format:
Ltaief, H., Luszczek, P., and Dongarra, J. 2013. High-performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures. ACM Trans. Math. Softw. 39, 3, Article 16 (April 2013), 22 pages.
DOI: http://dx.doi.org/10.1145/2450153.2450154

1. INTRODUCTION

The bidiagonal reduction (BRD) is an important first step when calculating the singular value decomposition (SVD) of any rectangular dense matrix [Golub and Reinsch 1970; Golub and Van Loan 1996, p. 257; Trefethen and Bau 1997, p. 25]:

$$A = U\Sigma V^T \quad A, \Sigma \in \mathbb{R}^{M\times N}, U \in \mathbb{R}^{M\times M}, V \in \mathbb{R}^{N\times N},$$

where the diagonal matrix Σ contains the singular values and the U and V dense orthogonal matrices are the corresponding left and right singular vectors, respectively. The necessity for calculating SVDs emerges from various computational science areas (e.g., in statistics where it is directly related to the principal component analysis method [Hotelling 1933, 1935]; in signal processing and pattern recognition as an essential filtering tool and in analysis control systems [Moore 1981]). Following the decompositional approach to matrix computation [Stewart 2000], we transform the dense matrix A to an upper bidiagonal form B by applying successive distinct orthogonal transformations [Householder 1958] from the left (X) as well as from the right (Y):

$$B = X^T AY \quad B, A \in \mathbb{R}^{M\times N}, \; X \in \mathbb{R}^{M\times M}, \; Y \in \mathbb{R}^{N\times N}.$$

This reduction step actually represents the most time-consuming phase when computing the singular values. Figure 1 shows the time breakdown between the main phases of SVD calculations for various matrix sizes using LAPACK implementation from the sequential Intel's Math Kernel Library (MKL) version 10.2 on an Intel Xeon core based on Core 2 architecture. The phases are: BRD (labeled *Reduction*), obtaining the singular values (labeled *Divide and Conquer Iteration*) and calculating the corresponding singular vectors from the reduced form using either the dqds algorithm [Fernando and Parlett 1994] (this method is labeled *dqds Iteration*) or using Cuppen's divide-and-conquer algorithm [Jessup and Sorensen 1994; Gu and Eisenstat 1995] (labeled *Divide and Conquer Backtransformation*). Our primary focus is the BRD portion of the computation, which can easily consume over 99% of the time needed to obtain the singular values, and roughly 75% if singular vectors are additionally calculated (see Figure 1). The QR iteration [Demmel and Kahan 1990; Deift et al. 1991] is no longer a method of choice for singular vectors because it takes longer by roughly 50% of the time used by faster methods. The QR method is now deprecated, but is still included in Figure 1 for comparison with the divide-and-conquer method. Because there are two methods in Figure 1 the last set of timing bars extends beyond the 100% mark, and could be easily dismissed if the reader is not interested in performance of the deprecated dqds method. However, for completeness, we indicated the relative time spent in dqds iteration – the timing bars are plotted above the 100% mark for each matrix size. As it was the case for the divide and conquer method, the iteration phase quickly becomes negligible relative to other phases.

With the emergence of multicore architectures, the state-of-the-art numerical libraries faced the problem of diminishing data bandwidth from the new memory design characterized by small data caches associated with each core. The problem of adequate performance has already been addressed in the PLASMA library [Dongarra 2010] in the context of the one-sided factorizations (LU, QR/LQ, and Cholesky) for solving systems of linear equations [Agullo et al. 2009] by redesigning the standard numerical methods using tile algorithms and providing a flexible dynamic runtime system to address productivity of the application development. More recently, the authors [Luszczek et al.

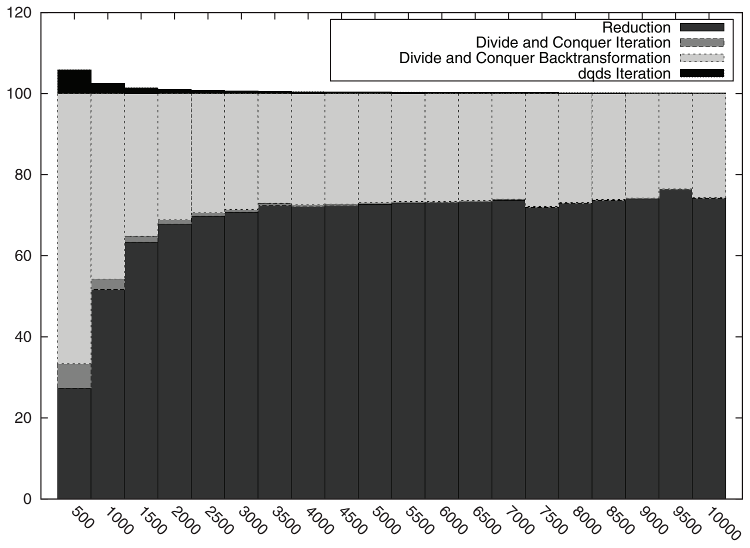


Fig. 1. Breakdown of time between three stages of SVD computation: *Reduction* to bidiagonal form (BRD), various *Iteration* methods to obtain singular values, and *Backtransformation* to obtain both sets of singular vectors using LAPACK's routine implementation from the sequential MKL 10.2 library on an Intel Xeon core based on Core 2 architecture.

2011] implemented a very efficient two-stage tridiagonal reduction (TRD) approach for dense symmetric matrices using tile algorithms on multicore architectures.

This article extends the authors' previous work (one-sided factorizations and TRD) to tackle the BRD case, which presents more challenges due to its increased algorithmic complexity. The standard BRD algorithm interleaves the QR and LQ factorizations and requires $^8\!/_3 N^3$ floating-point operations for an N-by-N matrix: twice the cost of the TRD algorithm. Following a two-stage approach, the matrix is transformed into a band bidiagonal form with a bandwidth of size NB using compute intensive kernels, introduced by Ltaief et al. [2010]. The band form is then further reduced to the required bidiagonal form using a bulge-chasing procedure. This second stage requires the development of new *memory-aware* computational kernels, which reduce memory traffic and memory contention. A dependence translation layer (DTL) allows the mapping of the access pattern (column major) of the bulge-chasing technique onto the tile layout, and helps to define the appropriate data dependencies. The dynamic runtime system called SMPSs [Perez et al. 2008; SMPSs Team 2008] enables scheduling and overlapping tasks generated from both stages, while ensuring that the data dependencies are not violated. Two-stage reduction algorithms for two-sided factorizations are not new approaches, but have recently enjoyed rekindled interests in the community. For instance, it has been used by Bischof et al. [2000] for TRD (SBR toolbox) and Kågström et al. [2008] in the context of Hessenberg and triangular reductions for the generalized eigenvalue problem for dense matrices. The tile bidiagonal reduction that was obtained in this way considerably outperforms the state-of-the-art open-source and commercial numerical libraries.

The bandwidth size of NB, which also corresponds to the tile size in our case, has a critical impact on the overall performance of the BRD algorithm. It has to be adequately chosen, possibly through an auto-tuning approach, so that the performance of either of the stages is not negatively affected.

The remainder of this document is organized as follows: Section 3 recalls the block BRD algorithm as implemented in LAPACK [Anderson et al. 1999] and explains its main deficiencies. Section 4 describes the implementation of the parallel tile BRD algorithm using a two-stage approach. Section 5 outlines the dependence translation layer (DTL). Section 6 has an overview of the different code kernels that are both compute-intensive and memory-efficient. Section 7 presents the performance results. A detailed analysis is provided to understand the critical impact of the bandwidth size on the overall algorithm. Comparison tests are run on shared-memory architectures against the state-of-the-art, high-performance dense linear algebra software libraries, LAPACK [Anderson et al. 1999] (open-source package) and Intel MKL 10.2 [MKL 2011] (commercial package). Finally, Section 8 summarizes the results of this article and presents the ongoing work.

2. RELATED WORK

Grosser and Lang [1999] describe an efficient parallel reduction to bidiagonal form by splitting the standard algorithm in two stages, that is, *dense to banded* and *banded to bidiagonal*, in the context of sparse linear algebra and distributed memory systems. The QR and LQ factorizations are done using a tree approach, where multiple column/row blocks can be reduced to triangular forms at the same time, which can ameliorate the overall parallel performance. Those triangular blocks are then reduced without taking into account their sparsity, which may add some extra flops. Our implementation also uses two stages to obtain the bidiagonal form, but for dense matrices.

Ralha [2003] proposed a new approach for the bidiagonal reduction called one-sided bidiagonalization. The main concept is to implicitly tridiagonalize the matrix $A^T A$ by a one-sided orthogonal transformation of A, that is, $F = AV$. As a first step, the right orthogonal transformation V is computed as a product of Householder reflectors. Then, the left orthogonal transformation U and the bidiagonal matrix B are computed using a Gram-Schmidt QR factorization of the matrix F. This procedure has numerical stability issues, and the matrix U may lose its orthogonality properties. In our implementation we only use Householder reflectors to avoid potential problems with numerical stability and at the same time we overcome the performance bottlenecks that this entails.

Barlow et al. [2005] and later, Bosner and Barlow [2007], further improved the stability of the one-sided bidiagonalization technique by merging the two distinct steps to compute the bidiagonal matrix B. The computation process of the left and right orthogonal transformations is now interleaved. Within a single reduction step, their algorithms simultaneously perform a block Gram-Schmidt QR factorization (using a recurrence relation) and a postmultiplication of a block of Householder reflectors chosen under a special criteria. Again, we refrain from using the Gram-Schmidt factorization in our code, but we introduce interleaving in both stages by means of dynamic scheduling, kernel splitting, and kernel prioritization.

3. THE LAPACK BLOCK BRD ALGORITHM

This section recalls the notion of block algorithms in LAPACK and describes, in particular, the block bidiagonal reduction (BRD).

3.1. Block Algorithms

LAPACK implements block algorithms to solve linear systems of equations as well as eigenvalue problems and singular value decompositions. Block factorization algorithms are characterized by two successive phases: panel factorization and update of the trailing submatrix. During the panel factorization, the transformations are only applied within the panel. The panel factorization is very rich in Level 2 BLAS operations because the transformations are singly applied. Once accumulated within the panel,

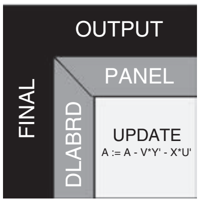


Fig. 2. Panel-update sequence for the LAPACK BRD algorithm.

those transformations are applied to the rest of the matrix (the trailing submatrix) in a blocked manner leading to Level 3 BLAS operations. While the update of the trailing submatrix is compute-bound and very efficient, the panel factorization is memory-bound and may appear to be a bottleneck for some numerical linear algebra algorithms. Last but not least, the parallelism within LAPACK occurs only at the level of the BLAS routines, which follows the expensive fork-join model. Basically, all processing units need to synchronize before and after each call to BLAS kernels.

3.2. LAPACK BRD Algorithm

The BRD algorithm with the TRD and the Hessenberg reduction (HRD) are the three two-sided factorizations. As opposed to one-sided factorizations (i.e., LU, Cholesky, QR/LQ), the computed transformations are applied from the left as well as from the right side of the matrix. In particular, Algorithm 1 and Figure 2 describe the LAPACK BRD algorithm for a rectangular matrix of size M by N with a block size NB (for simplicity, NB divides N). The panel factorization (DLABRD) of the block BRD algorithm interleaves two transformations, that is, left and right Householder-based reductions. The corresponding left and right reflectors are saved in the original matrix A in lieu of the annihilated elements. The accumulation of the left and right transformations (saved in two temporary storages X and Y) necessitates loading into memory the whole unreduced part of the matrix at each single reduction step. The update of the trailing submatrix is then straightforward. Two matrix-matrix multiplications are needed: one to apply the accumulated transformations, X, using the left reflectors (V) and the other one to apply the accumulated transformations, Y, using the right reflectors (U). While the update phase is compute-bound, the panel reduction involves at most Level 2 BLAS operations on large submatrices, which cannot be efficiently parallelized on currently available multicore systems (due to $\theta(n^2)$ floating-point operations (flops) performed on $\theta(n^2)$ floating-point data). The LAPACK BRD algorithm is thus characterized by the presence of a sequential operation (the panel factorization, i.e., DLABRD), which represents a small fraction of the total number of flops performed, ($\theta(n^2)$ flops for a total of $\theta(n^3)$ flops), but limits the scalability of the block BRD reduction on a multicore system when parallelism is only exploited at the level of the BLAS routines. The final computed diagonal and upper or lower diagonal elements are stored in D and E, respectively.

Moreover, this sequence of *Panel-Update* in LAPACK has shown strong limitations on multicore architectures. Indeed, the LAPACK framework is not capable of performing any look-ahead computations, where panel or update tasks from multiple steps can significantly overlap. Although, in practice, look-ahead techniques would be algorithmically possible (e.g., for one-sided factorizations). On the other hand, for two-sided transformations, and the BRD algorithm in particular, the one-stage approach for the reduction to the bidiagonal form necessitates that the panel computational step be atomic because it requires access to the entire trailing submatrix, which thus prevents any look-ahead calculations. The next section describes the concept of tile algorithms

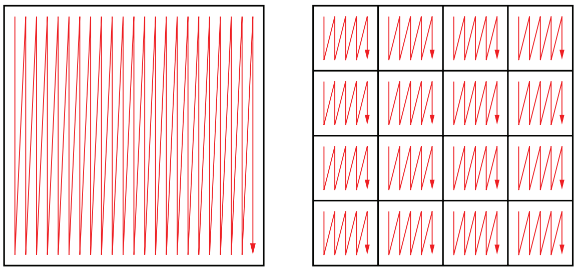


Fig. 3. Translation from LAPACK layout (column-major) to tile data layout.

ALGORITHM 1: LAPACK Blocked Bidiagonal Reduction

```
for $I = 1$ to min(M, N) step NB do
    → {Panel Factorization phase: Reduce rows and columns I:I+NB-1 to bidiagonal form and
    return the matrices X and Y, which are needed to update the unreduced part of the matrix}
    DLABRD(M-I+1, N-I+1, NB,
              A(I,I), LDA,
              D(I), E(I),
              X, LDX, Y, LDY)
    → {Update the trailing submatrix A(I+NB:N, I+NB:N) using an update of the form
    $A := A - V * Y' - X * U'$}
    DGEMM( 'NoTrans', 'Trans',
              M-I-NB+1, N-I-NB+1, NB, -ONE,
              A( I+NB, I ), LDA,
              Y, LDY, ONE,
              A( I+NB, I+NB ), LDA )
    DGEMM( 'NoTrans', 'NoTrans',
              M-I-NB+1, N-I-NB+1, NB, -ONE,
              X, LDX,
              A( I, I+NB ), LDA, ONE,
              A( I+NB, I+NB ), LDA )
end for
```

and explains how these new algorithms are able to supersede block algorithms, especially in the context of the BRD algorithm.

4. THE TWO-STAGE TILE BRD APPROACH

This section recalls the general principles of tile algorithms, as well as the idea behind two-stage approaches, and describes how these core aspects lead to the tile two-stage BRD algorithm.

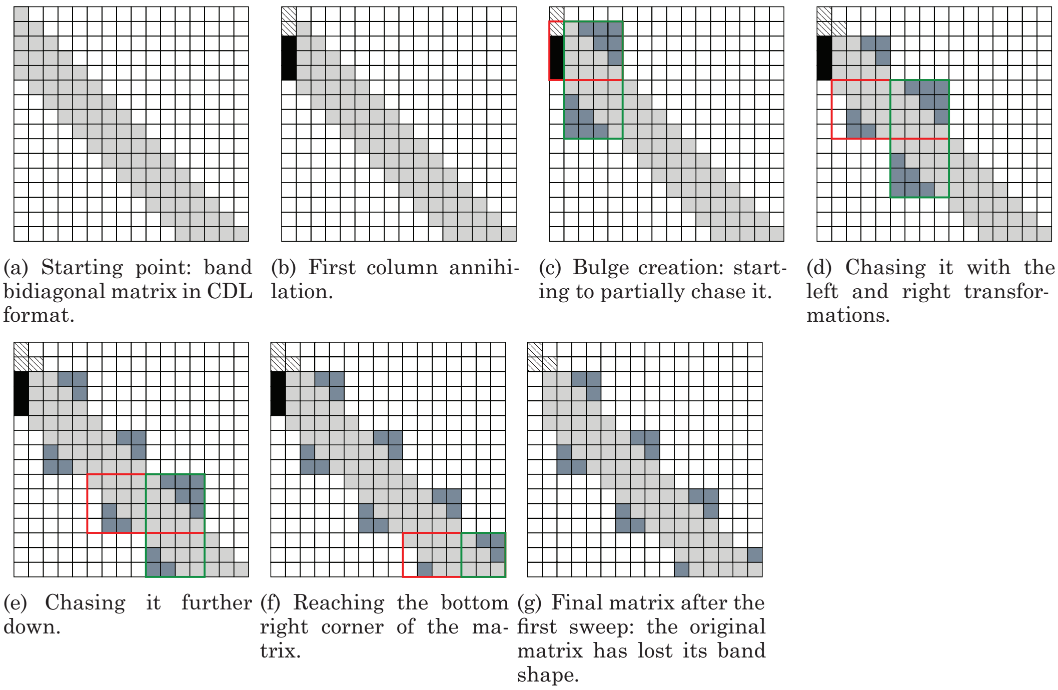
4.1. Tile Algorithms

Tile algorithms [Buttari et al. 2006, 2008, 2009; Dongarra 2010] have already shown promising results for the one-sided factorizations as compared to LAPACK and vendor libraries on multicore architectures [Agullo et al. 2009]. The general idea is to transform the original matrix to *tile data layout* (TDL) [Gustavson 2000] where each data tile is contiguous in memory as in Figure 3. This may demand a complete redesign of the standard numerical algorithm. The panel factorization as well as the update of the trailing submatrix are then decomposed into several fine-grained tasks, which fit the memory of the small core caches better. The parallelism is no longer hidden inside the BLAS routines, but rather is brought to the fore. The whole computation can then be represented with a directed acyclic graph (DAG), where nodes are computational tasks and edges represent the data dependencies among them. Next, it becomes critical to efficiently schedule the sequential fine-grained tasks across the processing units. A dynamic runtime environment system is used to distribute the tasks as soon as the data dependencies are satisfied.

Ltaief et al. [2010] had previously attempted to apply the tile algorithm principles to the BRD algorithm. Although high performance results were attained, the bidiagonal reduction was not complete, and only a partial reduction to *band* bidiagonal was feasible, which is impractical because the full reduction needs to be achieved in order to calculate the singular value decomposition. They have implemented new optimized kernels, which dramatically decrease the overhead of the panel factorization. Indeed, the standard BRD algorithm has been redesigned so that the panel factorization phases now involve only input/output data from the local corresponding tiles (and *not* from the entire unreduced matrix). More details can be found in Section IV C in Ltaief et al. [2010]. However, the obtained band bidiagonal form needs to be further reduced by using the bulge-chasing technique, which is explained in the following section.

4.2. Two-Stage Approach

Two-stage approaches have recently proven to be an interesting solution in achieving high performance in the context of two-sided reductions [Bischof et al. 2000; Kågström et al. 2008; Luszczek et al. 2011]. The first stage consists in reducing the original matrix to band form. The overhead of the Level 2 BLAS operations dramatically decreases, and most of the computation is performed in Level 3 BLAS, which makes this stage run closer to the theoretical peak of the machine. This stage is actually so compute-intensive that Bientinesi et al. [2010] proposed to completely offload it to GPU accelerators to further benefit from the underlying hardware. The second stage further reduces the band matrix to the corresponding compact form. A bulge-chasing procedure using orthogonal transformations annihilates the off-subdiagonal elements columnwise and the off-supdiagonal elements rowwise and hunts down the fill-in elements to the bottom, right corner of the matrix. Originally implemented for TRD [Bischof et al. 2000], this technique has been extended for the BRD algorithm. Figure 4 depicts the execution breakdown of chasing the first column (black elements) on a band bidiagonal matrix of size $N = 16$ and $NB = 4$ with *column-major data layout* (CDL). The red and green rectangles show the left and right transformations, respectively. The dashed elements are the final elements of the bidiagonal structure of the matrix. The dark grey elements represent the fill-in elements left after this first sweep. They will eventually fade out thanks to the subsequent sweeps. It is noteworthy that the introduced bulges are partially destroyed (in fact, only a single column/row per left/right transformations, respectively). If the bulges were destroyed in their entirety instead, the total number of operations would increase and the subsequent sweeps would reintroduce them anew anyway. By only eliminating the necessary parts of the bulges within one sweep, we allow the following sweeps to naturally chase down the leftover bulges. Finally, it may be readily observed that the whole narrow band structure of the matrix has to be traversed in order to annihilate a single column. Each sweep is very low in terms of floating-point operations and involves only small regions around the diagonal. Therefore, the standard bulge-chasing procedure is completely memory-bound, which renders it practically sequential. Although successive sweeps could potentially be pipelined, it might seriously increase the memory bus traffic, as each sweep would be working on different memory regions of the matrix and will not be able to forward the data between the cores' caches. A careful implementation is then paramount to make sure the loaded data into the cores' caches will be reused between subsequent sweeps (See Section 6.3).



(a) Starting point: band bidiagonal matrix in CDL format.

(b) First column annihilation.

(c) Bulge creation: starting to partially chase it.

(d) Chasing it with the left and right transformations.

(e) Chasing it further down.

(f) Reaching the bottom right corner of the matrix.

(g) Final matrix after the first sweep: the original matrix has lost its band shape.

Fig. 4. Execution breakdown of the bulge-chasing procedure on a band bidiagonal matrix of size N = 16 and NB = 4 with *column-major data layout* (CDL) after the first column annihilation (black elements). The red and green rectangles show the left and right transformations, respectively. The dark grey elements represent the fill-in elements, which eventually need to be chased down to the bottom right corner of the matrix. The dashed elements are the final elements of the bidiagonal structure of the matrix.

4.3. Tile Two-Stage BRD Algorithm

The goal of this two-stage tile BRD algorithm presented in this article is to incorporate the strengths of both tile algorithms and the two-stage approach in order to build an efficient framework for reducing a matrix to bidiagonal form.

Implementing the first stage using tile algorithms has already been implemented in Ltaief et al. [2010]. Figure 5 recalls how the band bidiagonal structure is obtained from a 4-by-4 tile matrix. The matrix is reduced to band form by interleaving LQ (right transformations) and QR (left transformations) factorizations. The light gray tiles correspond to transient data, which still need to be reduced. The black and dark gray tiles are being reduced and the dashed tiles are final data tiles. Figures 5(a) and 5(b) only show the last row/column tile annihilation occurring during the right/left reductions of the first step, respectively. Figures 5(c), 5(d), 5(e), and 5(f) represent the partial right/left reduction updates (in dark gray color) of the second/third step, triggered by the annihilation of the corresponding tiles (in black color). Figure 5(g) zeroes out the last tile (fourth step) and Figure 5(h) highlights the final band bidiagonal form. All in all, four interleaved LQ/QR factorization steps are needed to achieve the band bidiagonal form. The authors refer to the paper by Ltaief et al. [2010] for more detailed information.

The bulge-chasing algorithm from the second stage poses its own set of challenges that mostly reside in bridging the disparity of data layouts: TDL from the tile algorithm of the first stage and CDL for the bulge-chasing. Figure 6 shows the extent of this disparity. The bulge-chasing procedure on the tile matrix creates bulges, which could span

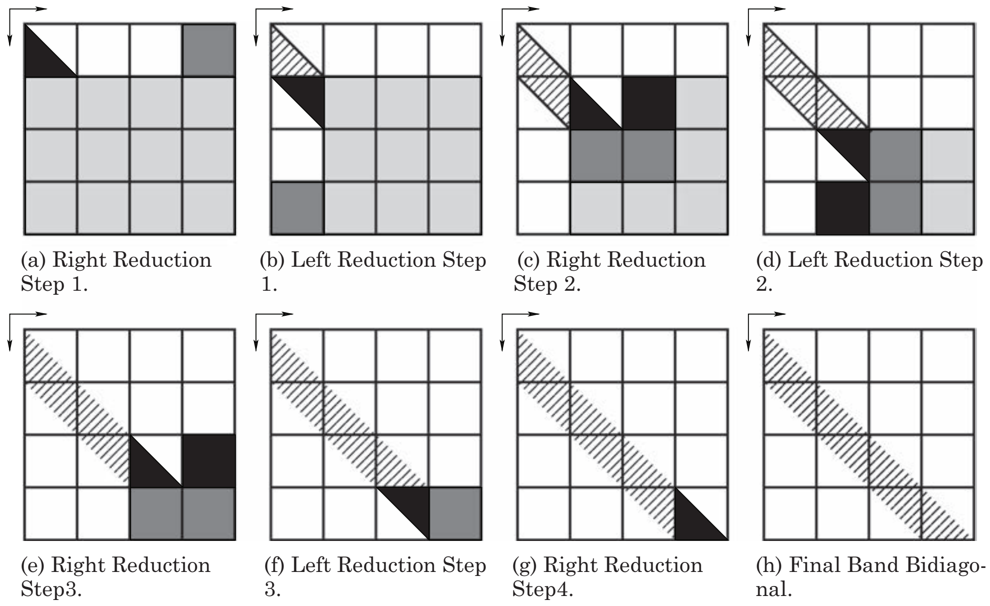

(a) Right Reduction Step 1.
(b) Left Reduction Step 1.
(c) Right Reduction Step 2.
(d) Left Reduction Step 2.
(e) Right Reduction Step3.
(f) Left Reduction Step 3.
(g) Right Reduction Step4.
(h) Final Band Bidiagonal.

Fig. 5. First stage: Reduction to band bidiagonal form applied on a 4x4 tile matrix.

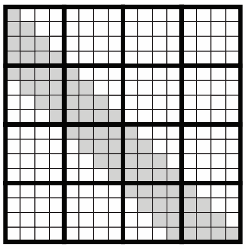
over multiple tiles, and therefore they are no longer contiguous in memory. Obviously, special computational kernels need to be implemented to handle the various cases, depending on the number of tiles involved in the particular task. The new kernels take care of the computational aspect while a layer of abstraction handles the adaptation of the bulge-chasing algorithm's CDL-based formulation to the underlying TDL format. This abstraction layer is important for the two-stage tile BRD algorithm, as it unifies the layout format across both stages and thus avoids costly data reorganization during the process.

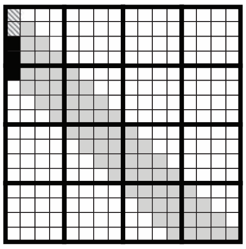
The next section describes the data dependence translation layer in the context of the BRD algorithm.

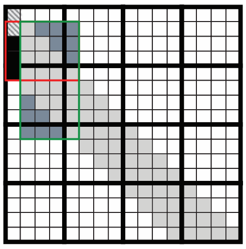
5. DEPENDENCE TRANSLATION LAYER

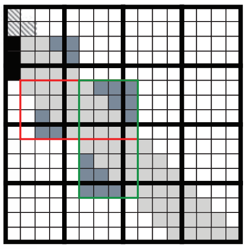
To reiterate the premise explained in the previous sections: the first stage (band reduction) of the BRD reduction fits well with the tile data layout, while the second stage (reduction from band to bidiagonal form) does not. The main reason for the mismatch is the misalignment of algorithmic tiles and storage tiles. The former operates at increments of a tile, and thus can easily be made to match the storage tiles. The latter, on the other hand, works in one column increments, as each column is annihilated by a similarity transformation, and this results in algorithmic tiles spanning one, two or four storage tiles. The four-tile case is shown in Figure 7 where the misaligned tile spans four storage tiles. The translation layer (DTL) we have devised provides a connection between the data access originating from the algorithmic formulation and the memory storage scheme. The abstraction provided by DTL affords the programmer the flexibility of working with a column-major layout, while the data dependences between computational tasks are appropriately propagated to the dynamic scheduling module as though they were specified for tile storage.

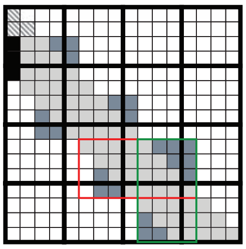
Furthermore, DTL is essential in that it provides the necessary information to the runtime system that allows for overlapping of tasks from both stages. The translation layer supplies the data dependences from the second stage in terms of data used in

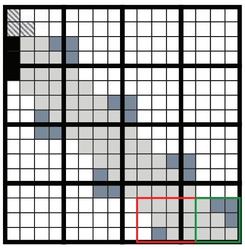

(a) Starting point: band bidiagonal matrix in TDL format.


(b) First column annihilation spanning across two tiles.


(c) Bulge creation: starting to partially chase it. The new bulges span over four tiles.


(d) Chasing it with left and right tile transformations.


(e) Chasing it further down.


(f) Reaching the bottom right corner of the matrix. The left transformation span over two tiles while the right transformation involves only a single tile.

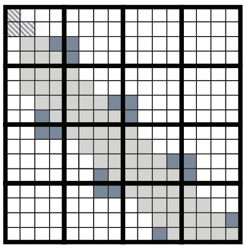

(g) Final tile matrix after the first column annihilation: the tile band structure is restored.

Fig. 6. Execution breakdown of the bulge-chasing procedure on a band bidiagonal matrix of size N = 16 and NB = 4 with tile data layout (TDL) after the first column annihilation (black elements). The red and green rectangles show the left and right transformations, respectively. The dark grey elements represent the fill-in elements, which eventually need to be chased down to the bottom right corner of the matrix. The dashed elements are the final elements of the bidiagonal structure of the matrix.

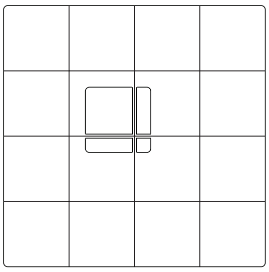


Fig. 7. An access to a misaligned tile is broken down by DTL into four subtiles.

the first stage which allows the runtime to begin scheduling the second stage tasks as soon as a sufficient portion of the first-stage work has finished. This is readily visible in Figure 8 that shows a DAG for reduction of a 6-by-6 matrix with tile size 2. The second-stage tasks (marked in shades of gray) may already be scheduled when less than half of the first stage is done in step 5. The figure shows the tasks ordered in minimum number of steps without violating intertask data dependencies—this represents an

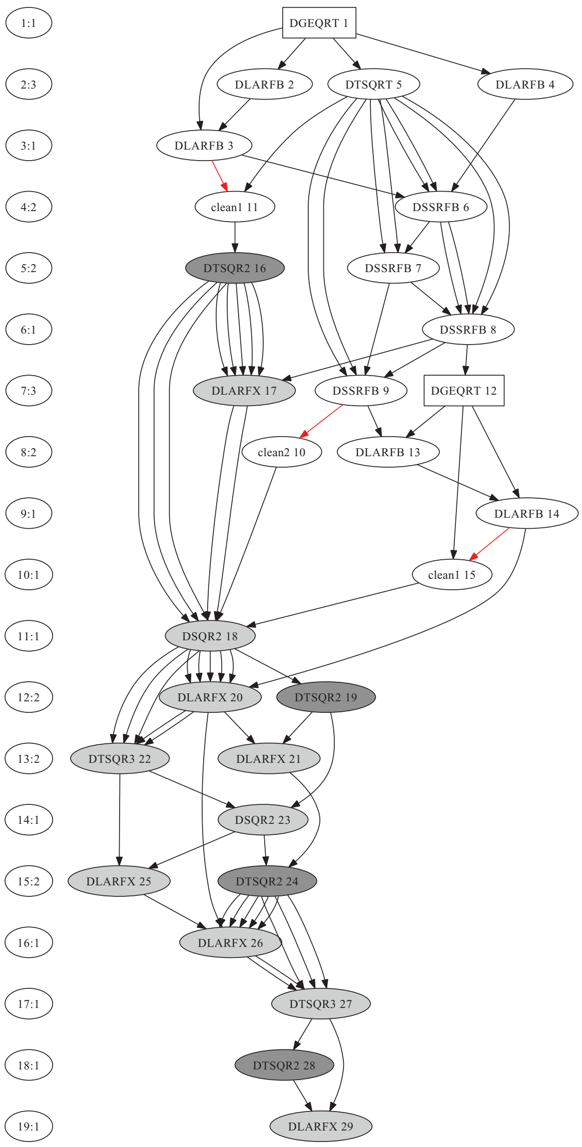


Fig. 8. Optimally scheduled DAG (19 steps) for bidiagonal reduction with the first-stage nodes marked with white and the second with shades of gray (the clean1 and clean2 nodes introduce zeros in some tiles to ensure correct result while using standard kernels). A number next to each node's name represents the order of submission of tasks to runtime scheduling.

optimal schedule that may not always be achieved at runtime due to the heuristic scheduling algorithm.

The operation of DTL may be explained by Figure 7, where the algorithmic tile spans four storage tiles. The flexibility of DTL allows specification of accesses to the matrix by using column-based storage. DTL intercepts these accesses, but first it determines which tiles are affected. Depending on the number of tiles, DTL then selects either the 1-tile kernel, 2-tile kernel, or a 4-tile kernel. Consequently, DTL may also be regarded as a dispatch layer that selects the proper variant of the computational kernel. Our formulation of bulge-chasing guarantees that there are only four possible kernel choices. Once the kernel is selected, it is then submitted to the runtime scheduling module for execution. To keep the data dependences satisfied, the submitted task requests exclusive access to the appropriate number of tiles: 4 tiles in the case of the scenario from Figure 7.

To summarize, the main purpose of DTL (or a "kernel variant selection layer") is to determine how the input and output data interact between the data layouts and then to choose an appropriate variant of the computational kernel. The need for DTL arises from the disparity of data access patterns between the column-oriented bulge-chasing procedure and tile-oriented storage layout.

The next section gives a detailed overview of the high-performance computational kernels.

6. HIGH-PERFORMANCE KERNEL DESCRIPTIONS

This section recalls the computational kernels involved in the first stage and presents the newly developed kernels for the second stage in the context of the two-stage tile BRD algorithm.

6.1. General Kernel Descriptions

All kernels are written in standard C and are composed of successive calls to BLAS routines. The kernels from the first stage are mostly Level 3 BLAS and those from the second stage are based on Level 1 and 2 BLAS. As implemented in LAPACK, these kernels rely on orthogonal transformations using Householder reflectors. Orthogonal transformations are an accepted technique that is commonly used for two-sided reductions because they guarantee numerical stability, as opposed to less computationally expensive elementary transformations, similar to that used in Gaussian elimination [Trefethen and Bau 1997]. Also, as explained in Section 4.1, the partitioning of the panel engenders the orthogonal transformation to be incremental rather than direct, as would happen with block algorithms.

6.2. Compute-Bound Kernels from the First Stage

The kernels described in this section have already been introduced and implemented in Section III A of Ltaief et al. [2010]. Therefore, the purpose of this section is only to make the article self-contained. This stage basically interleaves the QR and LQ factorizations at each step. There are six compute-intensive kernels overall.

—DGEQRT/DGELQT perform, respectively, a QR and an LQ factorizations of a single tile.
—DTSQRT/DTSLQT compute a QR and an LQ factorizations by combining a triangular tile (upper if QR, lower if LQ) with a corresponding full square tile. DTSQRT and DTSLQT are shown in Figure 5(a) and Figure 5(b), respectively.
—DORMQR/DORMLQ apply the orthogonal transformations computed from DGEQRT/DGELQT to the left/right sides, respectively, of the trailing submatrix.
—DTSMQR/DTSMLQ apply the orthogonal transformations computed from DTSQRT/DTSLQT to the left/right sides, respectively, of the trailing submatrix. The left and right applications from DTSMQR/DTSMLQ are laid out, respectively, in Figure 5(c) and Figure 5(d) (the black and dark grey data tiles).

Note that no extra storage is needed to save the Householder reflectors generated from the QR and LQ factorizations. Extra storage is only required to save the triangular

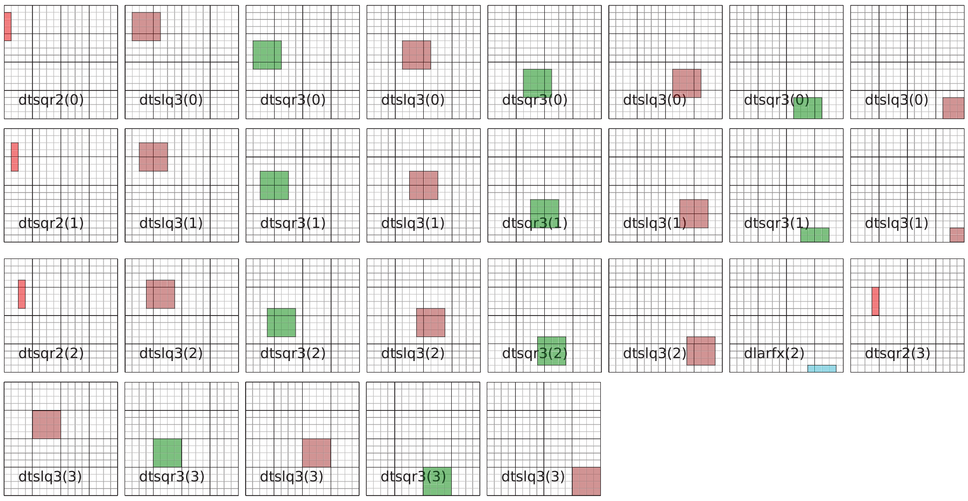


Fig. 9. Second stage: graphical representation of portions of the matrix accessed by the consecutive tasks. The yellow lines represent division of the matrix into individual entries, and the long black lines delineate matrix tiles in the first stage of the tridiagonal reduction and submatrices accessed in the second stage. The red tasks represent the DTSQR2 kernel; the brown tasks identify the DTSLQ3 kernel; the green tasks show the DTSQR3 kernels; and finally, the blue tasks are the DLARFX kernels.

factor T of the block reflectors computed from both factorizations, in order to apply them at once in the trailing submatrix.

6.3. Memory-Bound Kernels from the Second Stage

This second stage is clearly memory-bound, and the new kernels need to take this delicate property into account. There are four kernels overall. Figure 9 shows the execution breakdown of the bulge-chasing procedure for four complete sweeps on a 4-by-4 tile matrix. The figure represents the name of the consecutive tasks along with the reduction step (from 0 to 3) and the corresponding portions of the accessed matrix. Moreover, there are different cases to consider, depending on the region characteristic of the tile matrix being updated (more precisely, the region can span over one, two, or four tiles), and for each case, a particular instance of one of the three general kernels is required. In other words, the higher-level kernel needs to handle the diverse case in a comprehensive manner. Following are some details about the new kernels.

—DTSQR2 (red in Figure 9) is used to annihilate a single column that can only fit on one or two tiles.
—DTSLQ3 (brown in Figure 9) applies the reflectors computed in DTSQR2 to a diagonal block from the left. Then, it reduces the first row of the introduced bulge and immediately applies the corresponding reflectors on the right of the rest of the diagonal tile. The block being affected can span over one, two, or four neighboring tiles, as shown in the execution breakdown in Figure 9.
—DTSQR3 (green in Figure 9) applies the reflectors calculated in DTSLQ3 from the right. It then annihilates the first column of the created bulge and applies those freshly created reflectors to the left within the block. Here, the block being affected can span over one, two, or four neighbored tiles.
—DLARFX (cyan in Figure 9) applies the reflectors computed from DTSLQ3 to the right, and it *always* spans across two tiles.

The bulge-chasing procedure necessitates extra storage to save the generated Householder reflectors from the different bulges, especially if the calculation of the singular vectors is required. Furthermore, since those kernels are called extensively, the whole performance of the second stage relies heavily on an efficient implementation of those routines. All the functions called within the kernels have been inlined up to the level of the BLAS routines. Thus, being memory-bound gives a certain flexibility to reorder and to reorganize the successive computational steps within those kernels in order to optimize for cache reuse and data locality. Last but not least, the tile bulge-chasing procedure increases the DAG's critical path and makes it much more complex with lots of data dependencies spanning both stages and a number of nodes/tasks growing quadratically with the matrix size.

6.4. Algorithmic Complexity

Considering the original dense matrix square, the algorithmic complexity of the standard BRD is $^8\!/_3 N^3$ with N the matrix size. The number of first-stage flops in our tile two-stage BRD is $^8\!/_3\ N \times (N - NB) \times (N - NB)$, since the reduction is only achieved up to the band form. The second stage chases the fill-in elements created by the annihilation of the extra entries during the N sweeps. Each sweep calls the kernels at most $2 \times {}^N\!/_{NB}$ times, and $2 \times NB^2$ flops are performed for each kernel. After removing the lower order terms, the number of flops during the bulge-chasing procedure is then $\sim N \times 2 \times {}^N\!/_{NB} \times 2 \times NB^2 = 4 \times N^2 \times NB$. Therefore, the overall algorithmic complexity of our tile two-stage BRD is roughly the same as the standard BRD.

The next section presents some performance results of the overall two-stage tile BRD algorithm using the high-performance kernels described above. The DTL frame works in association with the dynamic runtime system called SMPSs that is capable of scheduling tasks from both stages simultaneously across the cores of homogeneous multicore architectures as long as data dependences are not violated.

7. PERFORMANCE ANALYSIS AND EXPERIMENTS

This section highlights the parallel performance results achieved by the two-stage tile BRD algorithm. A detailed analysis on the impact of the tile size NB on the overall framework is also discussed.

7.1. Experimental Environment

The experiments were conducted on a 16-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 quad-core quad-socket board. There are two levels of cache. The Level 1 cache, local to each core, is divided into 32 KiB 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 is running Linux 2.6.25 and provides Intel Compilers 11.0 together with the Intel MKL 10.2 vendor library. All the experiments presented below focus on asymptotic performance and were conducted on the maximum amount of cores available on the machine, that is, 16 cores. The two-stage tile BRD algorithm is compared against the equivalent BRD function from the state-of-the-art open-source and commercial numerical libraries that is, LAPACK 3.2 linked with optimized MKL BLAS and Intel MKL V10.2, respectively.

7.2. The Dynamic Runtime System

SMP Superscalar (SMPSs) [Perez et al. 2008; SMPSs Team 2008] is a parallel programming framework developed at the Barcelona Supercomputer Center (Centro Nacional de Supercomputación). SMPSs 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). However, the programmer is not responsible for exposing the structure of the task graph. The task graph is built automatically, based on the information in 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 codes 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). Furthermore, the SMPSs scheduler attempts to exploit locality by scheduling dependent tasks to the same thread, such that output data is reused immediately.

7.3. Modeling Performance with Respect to Tile Size

In order to better understand the behavior of our implementation and how it changes with various matrix and tile sizes, we created a performance model without taking into account intertask dependences, as we assume, for simplicity, perfect scalability of the first stage and bounded scalability for the second one. There are two components in the model:

(1) *computation time* (t_x), which encompasses the floating-point operations performed within each task inside the computation kernel routines; and
(2) *communication time* (t_c), which covers the combined latency of fetching the first cache line of a matrix tile and the runtime scheduler overhead, in addition to the time it takes to communicate the rest of the tile from the main memory to the cache memory close to the computing core.

For a N by N matrix with a tile size NB, time to completion $t(\mathsf{N}, \mathsf{NB})$ for both stages of the reduction is

$$t(\mathsf{N}, \mathsf{NB}) = t_x(\mathsf{N}, \mathsf{NB}) + t_c(\mathsf{N}, \mathsf{NB}).$$

Assuming that α represents the execution rate of floating-point operations, β—bandwidth to the main memory, and γ—latency to the main memory, in the first stage, the individual components of running time are (lower-order terms are omitted for the sake of simplicity of exposition):

$$t_x(\mathsf{N}, \mathsf{NB}) = \frac{1}{\alpha} \times \underbrace{\left(\frac{\mathsf{N}}{\mathsf{NB}}\right)^3}_{\text{number of tasks}} \times \overbrace{\mathsf{NB}^3}^{\text{number of flops per task}} = \frac{1}{\alpha} \times \mathsf{N}^3$$

and

$$\begin{aligned} t_c(\mathsf{N}, \mathsf{NB}) &= \overbrace{\left(\frac{\mathsf{N}}{\mathsf{NB}}\right)^3}^{\text{number of tasks}} \times \left(\frac{1}{\beta} \times \underbrace{\mathsf{NB}^2}_{\text{items to transfer}} + \overbrace{\gamma}^{\text{latency}}\right) \\ &= \mathsf{N}^3 \times \left(\frac{1}{\beta \times \mathsf{NB}} + \frac{\gamma}{\mathsf{NB}^3}\right). \end{aligned}$$

Similarly for the second stage:

$$t_x(\mathsf{N},\mathsf{NB}) = \frac{1}{\alpha} \times \underbrace{\mathsf{N}}_{\text{No. of columns}} \times \overbrace{\frac{\mathsf{N}}{\mathsf{NB}}}^{\text{number of bulges}} \times \underbrace{\mathsf{NB}^2}_{\text{number of flops}} = \frac{1}{\alpha} \times \mathsf{N}^2 \times \mathsf{NB}$$

and

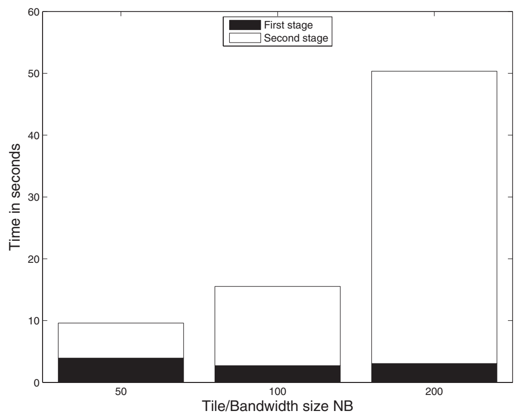
$$\begin{aligned} t_c(\mathsf{N},\mathsf{NB}) &= \underbrace{\mathsf{N}}_{\text{columns}} \times \overbrace{\frac{\mathsf{N}}{\mathsf{NB}}}^{\text{number of bulges}} \times \left(\frac{1}{\beta} \times \underbrace{\mathsf{NB}^2}_{\text{items to transfer}} + \overbrace{\gamma}^{\text{latency}}\right) \\ &= \mathsf{N}^2 \times \left(\frac{\mathsf{NB}}{\beta} + \frac{\gamma}{\mathsf{NB}}\right). \end{aligned}$$

The model clearly indicates that we should expect a drastically different behavior for the first and second stages of the reduction. The first stage benefits from larger tile sizes because the total time decreases for increasing tile size NB. The second stage, on the other hand, needs a particular tile size to achieve an optimal behavior because the communication component of the second stage is a rational function of the form $\mathsf{NB}/\beta + \gamma/\mathsf{NB}$ with a local minimum at $\sqrt{\beta\gamma}$. In other words, the bandwidth demand increases with the increase in width of the matrix band NB, but the effect of latency diminishes with the width because fewer kernels are invoked that need to bear the main memory latency and the scheduler overhead. To apply the model in a more concrete setting, we could assume the memory bandwidth β to be approximately 40 GB/s (a common value for both AMD and Intel chipsets) [Hennessy and Patterson 2012, p. 96] and the latency for most DRAM modules is about 500 cycles with 3 GHz clock frequency [Hennessy and Patterson 2012, p. 112]. This yields $\mathsf{NB}_{\text{optimal}} = \sqrt{40 \times 10^9 * \frac{500}{3\times 10^9}} \approx 80$. In the next section, we turn to experiment to investigate this phenomenon further and obtain a similar optimal NB value from actual runs.

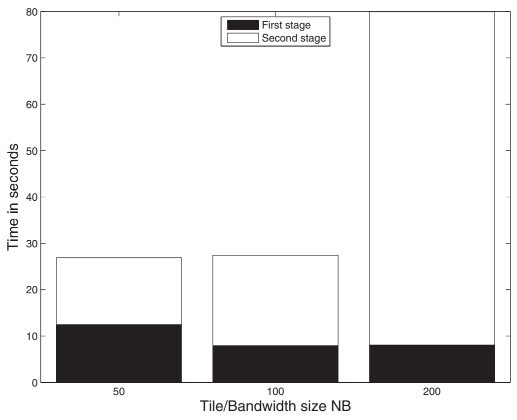
7.4. Tuning the Tile Size Experimentally

Out of the many tunable parameters available for tuning in the BRD code, the tile size NB stands out as the most critical for achieving optimal performance. It determines both the number of tasks and their granularity, and is difficult to tune optimally even for one-sided matrix factorizations [Agullo et al. 2009]. In BRD, a two-sided factorization, with the two-stage approach that we employ, there exists a natural tension between the stages which affects the choice of NB. The computational kernels from the first stage benefit greatly from coarse task granularity, which allows them to run closer to their sequential kernel peak performance. This follows from the compute-intensive nature of the kernels. On the contrary, the kernels of the second stage are mostly memory-bound and rely on data locality to achieve acceptable performance. Therefore, these kernels depend on data reuse and minimization of data being loaded from memory. This is most commonly achieved by a proper arrangement of data access patterns, which in our case can be achieved by memory-friendly scheduling (the runtime scheduler attempts to retain tasks on a single core as long as it helps with data locality) of tasks and having a small NB so that all of the tile data can be retained at the highest level of cache.

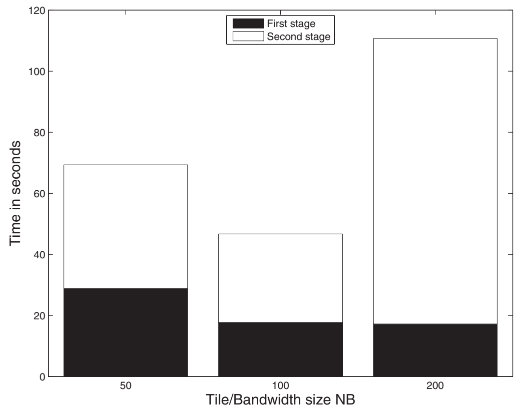
Figure 10 shows the impact of NB on the overall performance of the two-stage BRD with various matrix sizes. For small matrix sizes, that is, 4000 and 6000, the elapsed time increases with the tile size NB. However, for larger matrix sizes, that is, 8000 and 10000, the results are not so straightforward. The elapsed time of the second stage is substantially shorter for tile size NB = 50 than for NB = 100, and even for NB = 200



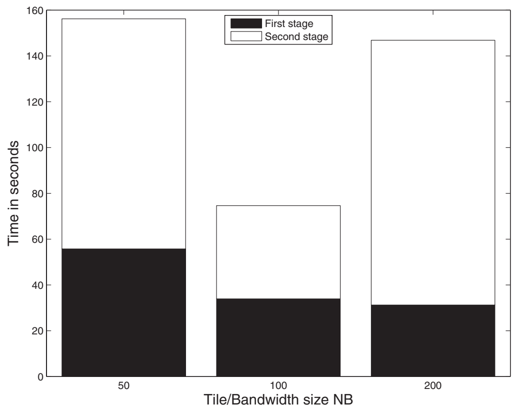
(a) N = 4000.



(b) N = 6000.



(c) N = 8000.



(d) N = 10000.

Fig. 10. Impact of NB on the elapsed time (in seconds) of the two-stage BRD for different matrix sizes.

when the matrix size is 10000. Therefore, our simple assumption made above that a small tile size will benefit the second stage is incorrect, and has to be closely examined for a given matrix size. Intuitively, a large matrix size N and a small tile size NB result in an increased number of data requests to the main memory. On the other hand, performance of the first stage deteriorates as expected for small tile size NB = 50 and is virtually constant for NB = 100 and NB = 200. These simple analyses and experiments lead us to believe that a good default value for a tile size is 100, and this is what we chose for the large-scale experiments. We also backed this choice with a series of performance analysis experiments, as shown below.

Figure 11 shows a detailed study of how the tile size influences the time to run the first and second stages of BRD as well as both stages simultaneously. The matrix size was set to 5040 because this allows us to have over 30 different NB values smaller than 200 (number 5040 has over 30 divisors due to its set of prime factors). The figure clearly indicates the predicted behavior for the first stage as the performance depends proportionally on the tile size. The larger the tile size, the better the performance of the computational kernel. However, the performance of the second stage exhibits a less obvious trend, that is, having a local minimum at 60. Departure from this value causes deterioration in performance. Accordingly, the cumulative performance of both

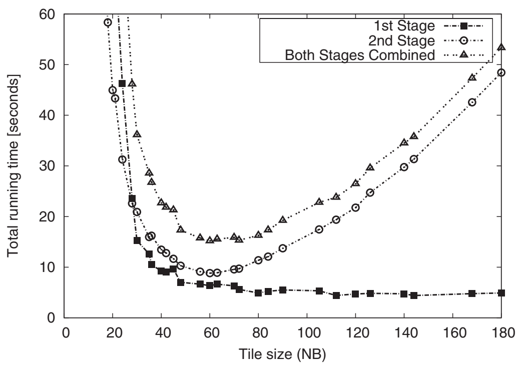


Fig. 11. Running time for first, second, and both stages for various tile sizes NB for matrix size 5040.

Table I. Time Breakdown Among Reduction Stages for Various Matrix Sizes and Potential Improvement if the Tile Size was Chosen Independently

	Shortest running time				
N	1st stage	2nd stage	Sum	Actual	Improvement
		[seconds]		[seconds]	[%]
2520	0.9	2.3	3.2	3.6	10.8%
4000	2.6	5.7	8.3	10.0	16.9%
5040	4.4	8.8	13.2	15.2	12.8%
7560	13.7	21.4	35.0	38.0	7.7%
9240	23.6	32.9	56.5	61.3	8.0%

stages has a similar property. To investigate this further, we chose additional matrix sizes that allow for a wide range of tile sizes and noted the combined performance for both stages. Figure 12 summarizes the results. Two observations are in order. First, for all matrix sizes there exists a locally optimal tile size. Second, an optimal tile size for one matrix size is not optimal for a different matrix size. The former observation makes a case for an autotunig method to be used to choose the optimal tile size [Agullo et al. 2010]. The latter observation raises a question of whether choosing an optimal tile size for each stage independently would benefit performance. Table I attempts to answer this question. The tabulated data shows stage-independent minimums (in the column marked "Sum") and the minimum of the total time. The stage-independent numbers are purely theoretical, as both stages have to share the same tile size. But in our opinion, it is still instructive to perform this "what-if" experiment. The column labeled "Improvement" shows the potential improvement in running time. It turns out that the improvement is not large (at most 17%) and decreases with the matrix size. In other words, choosing the minimum time from both stages is at most 17% faster than choosing the minimum sum.

7.5. Analysis of Algorithmic Variants

The two most common renditions for numerical linear algebra kernels are right- and left-looking [Anderson and Dongarra 1990; Dackland et al. 1992; Yi et al. 2004]. The former is a preferred option when the amount of available parallelism is the limiting factor [Blackford et al. 1997; Choi et al. 1996]. The latter, on the other hand, has much better locality characteristics, especially with respect to write operations, and is the preferred option for out-of-core codes [D'Azevedo and Luszczek 2003]. One of

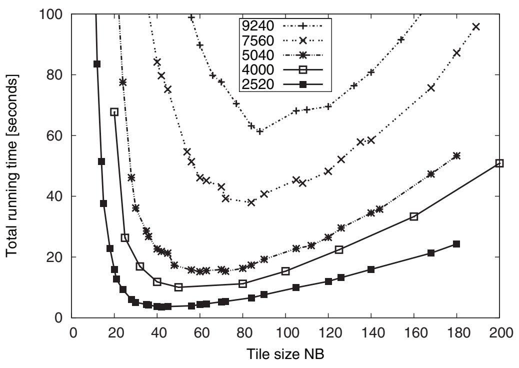


Fig. 12. Combined running time for various tile sizes NB for various matrix sizes.

the consequences of using dynamic DAG scheduling is the loss of fine-grain control in the exact ordering of computations [Haidar et al. 2011]. Despite this loss, we still attempted to investigate the influence of the algorithmic formulation by changing the order in which tasks are submitted to the runtime DAG scheduler. This is a crude approximation of either of the two popular variants, but the performance results still gave us an indication of which is the more important trait in our code: parallelism or locality. It turns out that the former is more desirable, as the right-looking variant consistently outperformed the latter, albeit by a small margin.

7.6. Experimental Results

This section presents the performance results of the overall two-stage BRD algorithm. Figure 13 compares our algorithm (labeled PLASMA with SMPSs scheduler) with the state-of-the-art open-source and commercial numerical libraries that is, multi-threaded LAPACK compiled with optimized MKL BLAS and Intel MKL version 10.2, respectively. It is surprising to see the same curve behaviors for both packages. The performance of both libraries goes up for small matrix sizes but then it just dies off considerably and does not scale while the matrix size increases. Our two-stage BRD approach starts to go beyond both numerical packages at the crossover point $N = 1500$ and outperforms them by far for large matrix sizes reaching up to a 30-fold speed-up on a 12000×12000 matrix size.

8. SUMMARY

This article focuses on a new high-performance two-stage tile bidiagonal reduction (BRD) on homogeneous multicore architectures. Using a two-stage approach on top of tile data layout, the original matrix is first reduced to band form using high-performance compute-intensive kernels and then further reduced to the final condensed form with efficient memory-optimized kernels. A data-dependence translation layer allows us to merge the directed acyclic graphs of tasks from both stages and removes the unnecessary in-between synchronization step. The dynamic runtime system, SMPSs, can then safely schedule the different computational tasks across the processing units and ensure that the data dependences are not violated. In the end, tuning the tile size is important to get the best performance from the two-stage BRD. A brute-force mechanism allows the retrieval of an optimal tile size NB depending on the problem size N. We achieved performance results that exceed those available from any alternative implementation we know. The new high-performance two-stage tile BRD achieves up to 30-fold speed-up on a 16 core Intel Xeon machine with a 12000×12000 matrix size against the state-of-the-art open source and commercial numerical softwares that

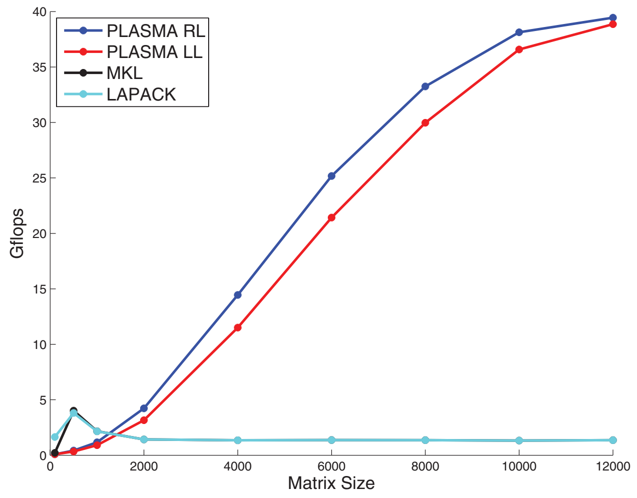


Fig. 13. Performance comparison of the tile two-stage BRD algorithm against Intel MKL version 10.2 and LAPACK 3.2 xGEBRD.

is, multithreaded LAPACK compiled with optimized MKL BLAS and Intel MKL V10.2 (2.5 Gflop/s for both), respectively. Last but not least, it is noteworthy that the overall performance of the two-stage tile BRD algorithm is 40 Gflop/s, and it represents only a small portion of the theoretical peak of the machine, roughly 25%. This level of overall performance exceeds with memory-bound codes (exemplified here by both LAPACK and MKL), even though the second stage of our implementation is memory-bound.

One of the future projects in this direction will be the calculation of the singular vectors. For that, the orthogonal transformations from both stages need to be accumulated into U (left transformations) and V (right transformations). While the accumulation of the reflectors from the first stage is straightforward and can be implemented very efficiently, the second stage is far from being trivial. Indeed, the reflectors created in the first stage can be efficiently accumulated because they are created within a tile all at once. In the second stage, each sweep generates many single reflectors which most of the time span across two tiles, and therefore need to be broken into two subreflectors. Similarly to the first stage, the order of their generations has to be respected during the accumulation step (if singular vectors are required) for numerical correctness purposes. All in all, the exercise of aggregating the multiple short subreflectors during the second stage appears to be challenging compared to the first stage, besides the actual extra flops involved in this procedure compared to the one-stage approach (adding an $O(n^3)$ term in the overall complexity of the algorithm). This is still an open research problem, and the authors are currently looking into removing this bottleneck. Finally, the authors are also investigating how this work can be extended to distributed environment systems within the DPLASMA framework [Bosilca et al. 2011].

REFERENCES

Agullo, E., Dongarra, J., Nath, R., and Tomov, S. 2010. Autotuned dense QR factorization for multicore architectures. Tech. rep. RR-7526, Institut National de Recherche en Informatique et en Automatique (INRIA). arXiv:1102.5328.

Agullo, E., Hadri, B., Ltaief, H., and Dongarra, J. 2009. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. ACM, New York, 1–12.

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S. L., DEMMEL, J. W., DONGARRA, J. J., CROZ, J. D., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. C. 1999. *LAPACK User's Guide* 3rd Ed, SIAM, Philadelphia, PA.

ANDERSON, E. AND DONGARRA, J. J. 1990. Evaluating block algorithm variants in LAPACK. In *Parallel Processing for Scientific Computing*, J. Dongarra et al. Eds., SIAM, Philadelphia, PA., 3–8.

BARLOW, J. L., BOSNER, N., AND DRMAČ , Z. 2005. A new stable bidiagonal reduction algorithm. *Linear Algebra Appl. 397*, 1, 35–84.

BIENTINESI, P., IGUAL, F., KRESSNER, D., AND QUINTANA-ORT'I, E. 2010. Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. *Parallel Process. Appl. Math. 6067*, 387–395.

BISCHOF, C. H., LANG, B., AND SUN, X. 2000. Algorithm 807: The SBR Toolbox—Software for successive band reduction. *ACM Trans. Math. Softw. 26*, 4, 602–616.

BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEVEDO, E. F., DEMMEL, J. W., DHILLON, I. S., DONGARRA, J. J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D. W., AND WHALEY, R. C. 1997. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA.

BOSILCA, G., BOUTEILLER, A., DANALIS, A., FAVERGE, M., HAIDAR, A., THOMAS HERAULT, J. K., LANGOU, J., LEMARINIER, P., LTAIEF, H., LUSZCZEK, P., YARKHAN, AND DONGARRA, J. 2011. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *Proceedings of the 12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-11)*. ACM, New York.

BOSNER, N. AND BARLOW, J. L. 2007. Block and parallel versions of one-sided bidiagonalization. *SIAM J. Matrix Anal. Appl. 29*, 3, 927–953.

BUTTARI, A., DONGARRA, J., KURZAK, J., LANGOU, J., LUSZCZEK, P., AND TOMOV, S. 2006. The impact of multicore on math software. In *Proceedings of the 8th International Workshop on Applied Parallel Computing. State of the Art in Scientific Computing (PARA)*. B. Kågström, et al. Eds., Lecture Notes in Computer Science, vol. 4699 Springer, Berlin, 1–10.

BUTTARI, A., LANGOU, J., KURZAK, J., AND DONGARRA, J. J. 2008. Parallel tiled QR factorization for multicore architectures. *Concurrency Comput. Pract. Exper. 20*, 13, 1573–1590. http://dx.doi.org/10.1002/cpe.1301.

BUTTARI, A., LANGOU, J., KURZAK, J., AND DONGARRA, J. J. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput. Syst. Appl. 35*, 38–53. http://dx.doi.org/10.1016/j.parco.2008.10.002.

CHOI, J., DONGARRA, J. J., OSTROUCHOV, S., PETITET, A., WALKER, D. W., AND WHALEY, R. C. 1996. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Sci. Program. 5*, 173–184.

DACKLAND, K., ELMROTH, E., KØAGSTR OM, B., AND LOAN, C. V. 1992. Design and evaluation of parallel block algorithms: Lu factorization on an IBM 3090 VF/600J. In *Proceedings of the 5th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, PA, 3–10.

D'AZEVEDO, E. AND LUSZCZEK, P. 2003. A framework for check-pointed fault-tolerant out-of-core linear algebra. In *Proceedings of the SIAM Conference on Computational Science and Engineering (CSE03)*. SIAM, Philadelphia, PA.

DEIFT, P., DEMMEL, J. W., LI, L.-C., AND TOMEI, C. 1991. The bidiagonal singular value decomposition and Hamiltonian mechanics. *SIAM J. Numer. Anal. 28*, 5, 1463–1516. (LAPACK Working Note #11).

DEMMEL, J. W. AND KAHAN, W. 1990. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput. 11*, 5, 873–912. (Also LAPACK Working Note #3).

DONGARRA, J. 2010. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version* 2.3. University of Tennessee.

FERNANDO, V. AND PARLETT, B. 1994. Accurate singular values and differential qd algorithms. *Numer. Math. 67*, 191–229.

GOLUB, G. H. AND REINSCH, C. 1970. Singular value decomposition and least squares solutions. *Numer. Math. 14*, 403–420.

GOLUB, G. H. AND VAN LOAN, C. F. 1996. *Matrix Computation* 3rd Ed. Johns Hopkins University Press, Baltimore, MD.

GROSSER, B. AND LANG, B. 1999. Efficient parallel reduction to bidiagonal form. *Parallel Comput. 25*, 8, 969–986.

GU, M. AND EISENSTAT, S. 1995. A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM J. Math. Anal. Appl. 16*, 79–92.

GUSTAVSON, F. G. 2000. New generalized matrix data structures lead to a variety of high-performance algorithms. In *Proceedings of the IFIP WG 2.5 Working Conference on Software Architectures for Scientific Computing Applications*. Kluwer Academic, Amsterdam, 211–234.

HAIDAR, A., LTAIEF, H., YARKHAN, A., AND DONGARRA, J. 2011. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. concurrency and computations: Practice and experience. Tech. rep. UT-CS-11-666, University of Tennessee.

HENNESSY, J. L. AND PATTERSON, D. A. 2012. *Computer Architecture: A Quantitative Approach* 5th Ed. Morgan Kaufmann.

HOTELLING, H. 1933. Analysis of a complex of statistical variables into principal components. *J. Edu. Psych. 24*, 417–441, 498–520.

HOTELLING, H. 1935. Simplified calculation of principal components. *Psychometrica 1*, 27–35.

HOUSEHOLDER, A. S. 1958. Unitary triangularization of a nonsymmetric matrix. *J. ACM 5*, 4. DOI 10.1145/320941.320947.

JESSUP, E. R. AND SORENSEN, D. 1994. A parallel algorithm for computing the singular value decomposition of a matrix. *SIAM J. Matrix Anal. Appl. 15*, 530–548.

KÅGSTRÖM, B., KRESSNER, D., QUINTANA-ORTÍ, E., AND QUINTANA-ORTÍ , G. 2008. Blocked algorithms for the reduction to Hessenberg-triangular form revisited. *BIT Numer. Math. 48*, 563–584.

LTAIEF, H., KURZAK, J., AND DONGARRA, J. 2010. Parallel two-sided matrix reduction to band bidiagonal form on multicore architectures. *IEEE Trans. Parallel Distrib. Syst*. 417–423.

LUSZCZEK, P., LTAIEF, H., AND DONGARRA, J. 2011. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In *Proceedings of IPDPS 2011*. ACM, New York.

MKL. 2011. Intel, Math Kernel Library (MKL). http://www.intel.com/software/products/mkl/. Version 10.2.

MOORE, B. C. 1981. Principal component analysis in linear systems: Controllability, observability, and model reduction. *IEEE Trans. Autom. Control AC-26*, 1.

PEREZ, J., BADIA, R., AND LABARTA, J. 2008. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, Los Alamitos, CA. 142–151.

RUI RALHA. 2003. One-sided reduction to bidiagonal form. *Linear Algebra Appl. 358*, 219–238.

SMPSs Team. 2008. *SMP Superscalar (SMPSs) User's Manual*. Version 2.3.

STEWART, G. W. 2000. The decompositional approach to matrix computation. *Comput. Sci. Eng. 2*, 1, 50–59.

TREFETHEN, L. N. AND BAU, D. 1997. *Numerical Linear Algebra*. SIAM, Philadelphia, PA.

YI, Q., KENNEDY, K., YOU, H., SEYMOUR, K., AND DONGARRA, J. 2004. Automatic blocking of qr and lu factorizations for locality. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Memory System Performance (MSP'04)*. ACM, New York.

Received May 2011; revised March 2012; accepted June 2012