Parallel Band Two-Sided Matrix Bidiagonalization for Multicore Architectures *LAPACK Working Note # 209*

Hatem Ltaief[1], Jakub Kurzak[1], and Jack Dongarra[1,2,3⋆]

[1] Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville
[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee
[3] School of Mathematics & School of Computer Science, University of Manchester
{ltaief, kurzak, dongarra}@eecs.utk.edu

Abstract. The objective of this paper is to extend, in the context of multicore architectures, the concepts of *algorithms-by-tiles* [Buttari *et al.*, 2007] for Cholesky, LU, QR factorizations to the family of two-sided factorizations. In particular, the bidiagonal reduction of a general, dense matrix is very often used as a pre-processing step for calculating the singular value decomposition. Furthermore, in the last Top500 list from June 2008, 98% of the fastest parallel systems in the world were based on multicores. The manycore trend has increasingly exacerbated the problem, and it becomes critical to efficiently integrate existing or new numerical linear algebra algorithms suitable for such hardware. By exploiting the concept of *algorithms-by-tiles* in the multicore environment (i.e., high level of parallelism with fine granularity and high performance data representation combined with a dynamic data driven execution), the band bidiagonal reduction presented here achieves 94 Gflop/s on a 12000×12000 matrix with 16 Intel Tigerton 2.4 GHz processors.

1 Introduction

The objective of this paper is to extend, in the context of multicore architectures, the concepts of *algorithms-by-tiles* by Buttari *et al.* [7] for Cholesky, LU, QR factorizations to the family of two-sided factorizations i.e., Hessenberg, Tridiagonalization, Bidiagonalization. In particular, the Bidiagonal Reduction (BRD) of a general, dense matrix is very often used as a pre-processing step for calculating the Singular Value Decomposition (SVD) [14, 28]:

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

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

The necessity of calculating SVDs emerges from various computational science disciplines, e.g., in statistics where it is related to principal component analysis, in signal processing and pattern recognition, and also in numerical weather prediction [10]. The basic idea is to transform the dense matrix A to an upper bidiagonal form B by applying successive distinct transformations from the left (U) as well as from the right (V) as follows:

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

The most commonly used algorithm to perform this two-sided reduction is the Golub-Kahan bidiagonalization [15]. Although this algorithm works for any matrix size, it adds extra floating point operations for rectangular matrices and thus, faster methods such as the Lawson-Hanson-Chan bidiagonalization are preferred [8]. Here, only square matrices are consider, and performance result comparisons of different bidiagonalization algorithms for rectangular matrices will appear in a companion paper.

Also, we only look at the first stage of BRD, which goes from the original dense matrix A to a band bidiagonal matrix B_b, with b being the number of upper-diagonals. The second stage, which annihilates those additional b upper-diagonals, has been studied especially by Lang [21] and is not examined in this paper. This *two-stage* transformation process is also explained by Grosser *et al.* [16]. Although expensive, orthogonal transformations are accepted techniques and commonly used for this reduction because they guarantee stability, as opposed to Gaussian Elimination [28]. The two common transformations are based on Householder reflectors and Givens rotations. Previous work by the authors [22] demonstrates the effectiveness of Householder reflectors over Givens rotations. Therefore, this two-sided band BRD is done by using Householder reflectors.

Furthermore, in the last Top500 list from June 2008 [1], 98% of the fastest parallel systems in the world were based on multicores. The many-core trend has exacerbated the problem even more and it becomes judicious to efficiently integrate existing or new numerical linear algebra algorithms suitable for such hardware. As discussed in [7], a combination of several parameters is essential to match the architecture associated with the cores: (1) fine granularity to reach a high level of parallelism and to fit the cores' small caches; (2) asynchronicity to prevent any global barriers; (3) Block Data Layout (BDL), a high performance data representation to perform efficient memory access; and (4) dynamic data driven scheduler to ensure any enqueued tasks can immediately be processed as soon as all their data dependencies are satisfied. While (1) and (3) represent important items for one-sided and two-sided transformations, (2) and (4) are even more critical for two-sided transformations because of the tremendous amount of tasks generated by the right transformation. Indeed, as a comparison, the algorithmic complexity for the QR factorization is $4/3\, n^3$, while it is $8/3\, n^3$ for the BRD algorithm, with n being the matrix size. On the other hand, previous work done by Kurzak *et al.* [19, 20] show how the characteristics of tiled algorithms

perfectly match even the architectural features of modern multicore processors such as the Cell Broadband Engine processor.

The remainder of this document is organized as follows: Section 2 recalls the standard BRD algorithm. Section 3 describes the implementation of the parallel tiled BRD algorithm. Section 4 outlines the pros and cons of static and dynamic scheduling. Section 5 presents performance results. Comparison tests are run on shared-memory architectures against the state of the art, high performance dense linear algebra software libraries, LAPACK [3] and ScaLAPACK [9]. Section 6 gives a detailed overview of previous projects in this area. Finally, section 7 summarizes the results of this paper and presents the ongoing work.

2 The Standard Bidiagonal Reduction

In this section, we review the original BRD algorithm of a general, dense matrix.

2.1 The Sequential Algorithm

The standard BRD algorithm of $A \in \mathbb{R}^{n \times n}$ based on Householder reflectors combines two factorizations methods, i.e. QR (left reduction) and LQ (right reduction) decompositions. The two phases are written as follows:

Algorithm 1 Bidiagonal Reduction with Householder reflectors

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

Algorithm 1 takes as input a dense matrix A and gives as output the upper bidiagonal decomposition. The reflectors u_j and v_j can be saved in the lower and upper parts of A, respectively, for storage purposes and used later if necessary. The bulk of the computation is located in line 5 and in line 10 in which the reflectors are applied to A from the left and then from the right, respectively. Four flops are needed to annihilate one element of the matrix, which makes the total number of operations for such algorithm $8/3\ n^3$ (the lower order terms are neglected). It is obvious that Algorithm 1 is not efficient as is, especially because it is based on matrix-vector Level-2 BLAS operations. Also, a single

entire column/row is reduced at a time, which engenders a large stride access to memory. The main contribution described in this paper is to transform this algorithm to work on tiles instead to generate, as many as possible, matrix-matrix Level-3 BLAS operations. First introduced by Berry *et al.* in [5] for the reduction of a nonsymmetric matrix to block upper-Hessenberg form and then revisited by Buttari *et al.* in [7], this idea considerably improves data locality and cache reuse.

3 The Parallel Band Bidiagonal Reduction

In this section, we present the parallel implementation of the band BRD algorithm based on Householder reflectors.

3.1 Fast Kernel Descriptions

There are eight overall kernels implemented for the two phases, four for each phase.

For phase 1 (left reduction), the first four kernels are identical to the ones used by Buttari *et al.* [7] for the QR factorization, in which the reflectors are stored in column major form. DGEQRT is used to do a QR blocked factorization using the WY technique for efficiently accumulating the Householder reflectors [26]. The DLARFB kernel comes from the LAPACK distribution and is used to apply a block of Householder reflectors. DTSQRT performs a block QR factorization of a matrix composed of two tiles, a triangular tile on top of a dense square tile. DSSRFB updates the matrix formed by coupling two square tiles and applying the resulting DTSQRT transformations. Buttari *et al.* gives a detailed description of the different kernels [7].

For phase 2 (right reduction), the reflectors are now stored in rows. DGELQT is used to do a LQ blocked factorization using the WY technique as well. DTSLQT performs a block LQ factorization of a matrix composed of two tiles, a triangular tile beside a dense square tile. However, minor modifications are needed for the DLARFB and DSSRFB kernels. These kernels now take into account the row storage of the reflectors.

Moreover, since the right orthogonal transformations do not destroy the zero structure and do not introduce fill-in elements, the computed left and right reflectors can be stored in the lower and upper annihilated parts of the original matrix, for later use. Although the algorithm works for rectangular matrices, for simplicity purposes, only square matrices are considered. Let NBT be the number of tiles in each direction. Then, the tiled band BRD algorithm with Householder reflectors appears as in Algorithm 2. It basically performs a sequence of interleaved QR and LQ factorizations at each step of the reduction.

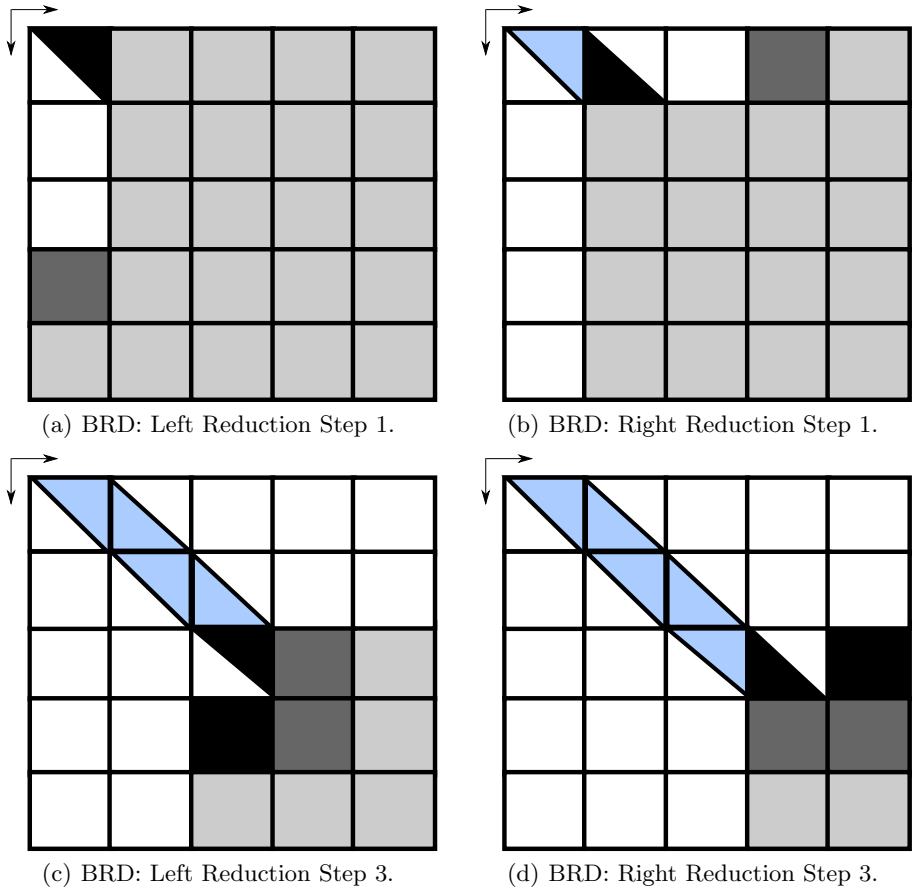
Algorithm 2 Tiled Band BRD Algorithm with Householder reflectors.

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

The characters "L" and "R" stand for Left and Right updates. In each kernel call, the triplets (i, ii, iii) specify the tile location in the original matrix, as in figure 1: (i) corresponds to the reduction step in the general algorithm, (ii) gives the row index and (iii) represents the column index. For example, in figure 1(a), the black tile is the input dependency at the current step, the white tiles are the zeroed tiles, the bright gray tiles are those which need to be processed and finally, the dark gray tile corresponds to DTSQRT(1,4,1). In figure 1(a), the blue tiles represent the final data tiles and the dark gray tile is DLARFB("R",1,1,4). In figure 1(b), the reduction is at step 3 where the dark gray tiles represent DSSRFB("L",3,4,4). In figure 1(c), the dark gray tiles represent DSSRFB("R",3,4,5).

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

The next section describes the number of operations needed to apply this reduction.



(a) BRD: Left Reduction Step 1.

(b) BRD: Right Reduction Step 1.

(c) BRD: Left Reduction Step 3.

(d) BRD: Right Reduction Step 3.

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

3.2 Operation Count

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

Furthermore, by using updating factorization techniques as suggested in [14, 27], the kernels for both implementations can be applied to tiles of the original matrix. Using updating techniques to tile the algorithms have first been proposed by Yip [29] for LU to improve the efficiency of out-of-core solvers, and were recently reintroduced in [17, 23] for LU and QR, once more in the out-of-core context. The cost of these updating techniques is an increase in the

operation count for the whole BRD reduction. However, as suggested in [11–13], by setting up inner-blocking within the tiles during the panel factorizations and the trailing submatrix update, DGEQRT-DGELQT-DTSQRT-DTSLQT kernels and DLARFB-DSSRFB kernels respectively, those extra flops become negligible provided $s << b$, with s being the inner-blocking size (see Buttari *et al.* [7] for further information). This blocking approach has been also described in [17, 24].

In the following part, we present a comparison of two approaches for tile scheduling, i.e., a static and a dynamic data driven execution scheduler that ensures the small kernels (or tasks) generated by Algorithm 2 are processed as soon as their respective dependencies are satisfied.

4 Static Vs Dynamic Scheduling

Two types of schedulers were used, a dynamic one, where scheduling decisions are made at runtime, and a static one, where the schedule is predetermined.

The dynamic scheduling scheme similar to [7] has been extended for the two-sided orthogonal transformations. A Directed Acyclic Graph (DAG) is used to represent the data flow between the nodes/kernels. While the DAG is quite easy to draw for a small number of tiles, it becomes very complex when the number of tiles increases and it is even more difficult to process than the one created by the one-sided orthogonal transformations. Indeed, the right updates impose severe constraints on the scheduler by filling up the DAG with multiple additional edges. The dynamic scheduler maintains a central progress table, which is accessed in the critical section of the code and protected with mutual exclusion primitives (POSIX mutexes in this case). Each thread scans the table to fetch one task at a time for execution. As long as there are tasks with all dependencies satisfied, the scheduler will provide them to the requesting threads and will allow an out-of-order execution. The scheduler does not attempt to exploit data reuse between tasks. The centralized nature of the scheduler is inherently non-scalable with the number of threads. Also, the need for scanning potentially large table window, in order to find work, is inherently non-scalable with the problem size. However, this organization does not cause performance problems for the numbers of threads, problem sizes and task granularities investigated in this paper.

The static scheduler used here is a derivative of the scheduler used successfully in the past to schedule Cholesky and QR factorizations on the Cell processor [18, 20]. The static scheduler imposes a linear order on all the tasks in the factorization. Each thread traverses the tasks space in this order picking a predetermined subset of tasks for execution. In the phase of applying transformations from the right each thread processes one block-column of the matrix; In the phase of applying transformations from the left each thread processes one block-row of the matrix (figure 2). A dependency check is performed before executing each task. If dependencies are not satisfied the thread stalls until they are (implemented by busy waiting). Dependencies are tracked by a progress table, which contains global progress information and is replicated on all threads. Each thread calculates the task traversal locally and checks dependencies by polling

the local copy of the progress table. Due to its decentralized nature, the mechanism is much more scalable and of virtually no overhead. Also, processing of tiles along columns and rows provides for greater data reuse between tasks, to which the authors attribute the main performance advantage of the static scheduler. Since the dynamic scheduler is more aggressive in fetching of tasks, it completes

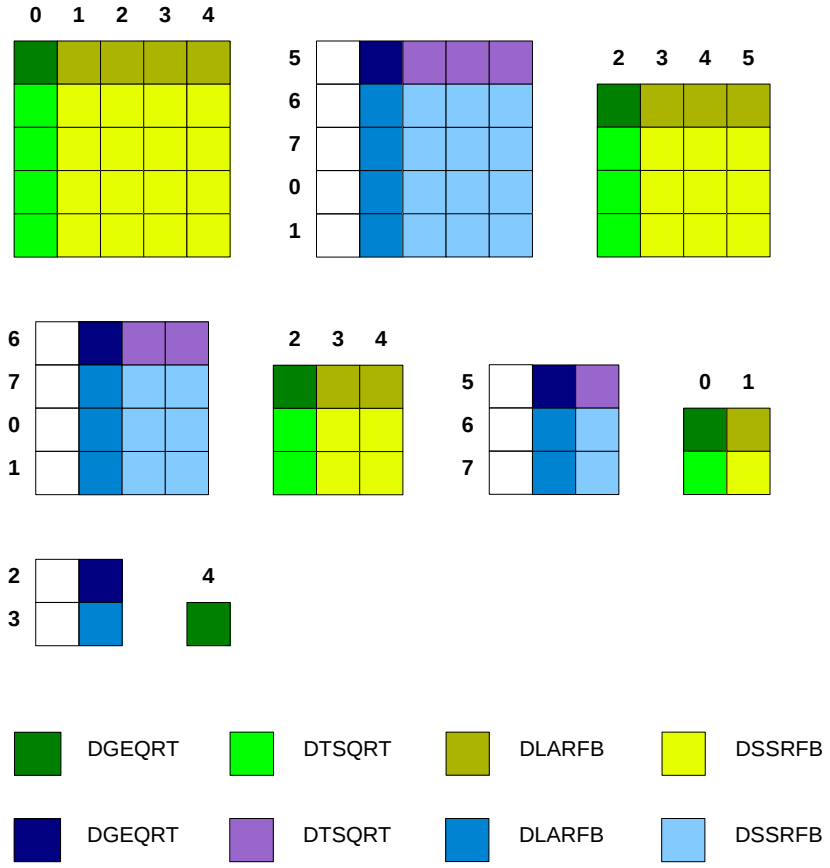


Fig. 2. Task Partitioning with eight cores on a 5×5 tile matrix.

each step of the factorization faster. The static scheduler, on the other hand, takes longer to complete a given step of the factorization, but successfully overlaps consecutive steps achieving the pipelining effect, what leads to very good overall performance (figure 3).

In the next section, we present the experimental results comparing our band BRD implementations with the two schedulers against the state of the art libraries, i.e., LAPACK [3], ScaLAPACK [9] and MKL version 10 [2].

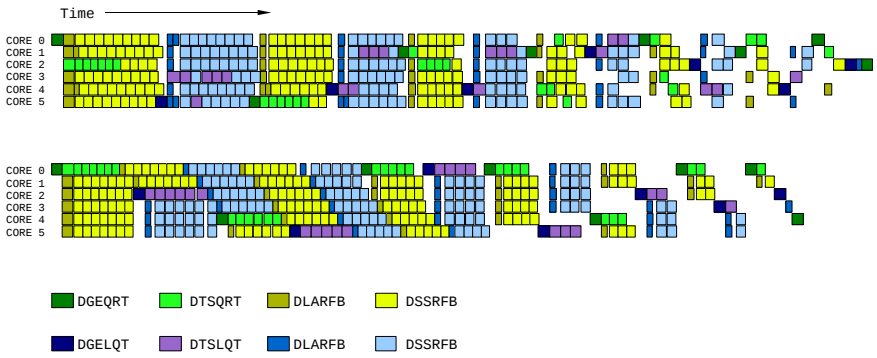


Fig. 3. Scheduler Tracing with six Intel Tigerton 2.4 GHz cores: Top dynamic – Bottom static.

5 Experimental Results

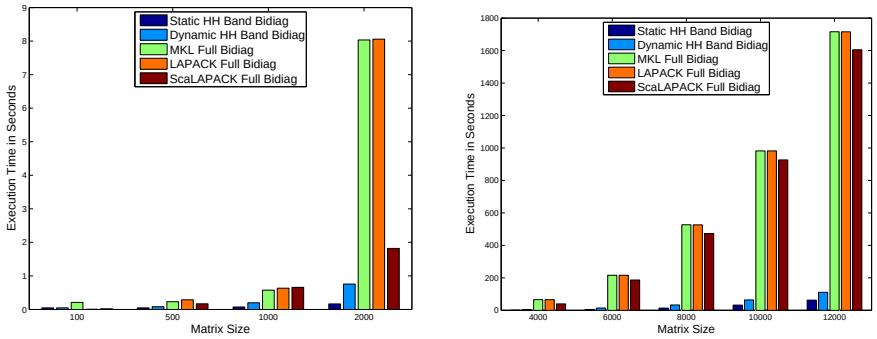
The experiments have been achieved on two different platforms: a quad-socket dual-core Intel Itanium 2 1.6 GHz (eight total cores) with 16GB of memory, and a quad-socket quad-core Intel Tigerton 2.4 GHz (16 total cores) with 32GB of memory. Hand tuning based on empirical data has been performed for large problems to determine the optimal tile size $b = 200$ and inner-blocking size $s = 40$ for the tiled band BRD algorithm. The block sizes for LAPACK and ScaLAPACK have also been hand tuned to get a fair comparison, $b = 32$ and $b = 64$ respectively.

Figure 4 shows the elapsed time in seconds for small and large matrix sizes on the Itanium system with eight cores. The band BRD algorithms based on Householder reflectors with static scheduling is slightly better than with dynamic scheduling. However, both implementations by far outperform the others: for a 12000×12000 problem size, they run approximately 25 x faster than the full BRD of ScaLAPACK, MKL and LAPACK. Figure 5(a) presents the parallel performance in Gflop/s of the band BRD algorithm on the Itanium system. The algorithm with dynamic scheduling runs at 82% of the machine theoretical peak of the system and at 92% of the DGEMM peak. Figure 5(b) zooms in on the three other implementations, and the parallel performance of the full BRD with ScaLAPACK is significantly higher than the full BRD of LAPACK and MKL for small matrix sizes. Also, the performances are almost the same for larger matrix sizes.

The same experiments have been conducted on the Xeon system with 16 cores. Figure 6 shows the execution time in seconds for small and large matrix sizes. Again, both band BRD algorithms almost perform in the same manner. For a 12000×12000 problem size, the band BRD algorithm with dynamic scheduler roughly runs 70 x faster than MKL and LAPACK, and 20 x faster than ScaLAPACK. Figure 7(a) presents the parallel performance in Gflop/s of the

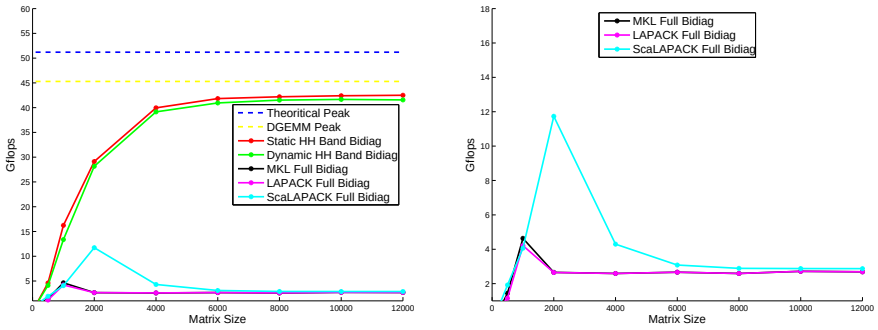
band BRD algorithm. It scales quite well while the matrix size increases, reaching 94 Gflop/s. It runs at 61% of the system theoretical peak and 72% of the DGEMM peak. The zoom-in seen in figure 7(b) highlights the weakness of the full BRD algorithm of MKL, LAPACK and ScaLAPACK. Note: the full BRD of ScaLAPACK is twice as fast as than the full BRD of MKL and LAPACK most likely thanks to the Two-dimensional Block Cyclic Distribution.

The following section briefly comments on the previous work done in BRD algorithms.



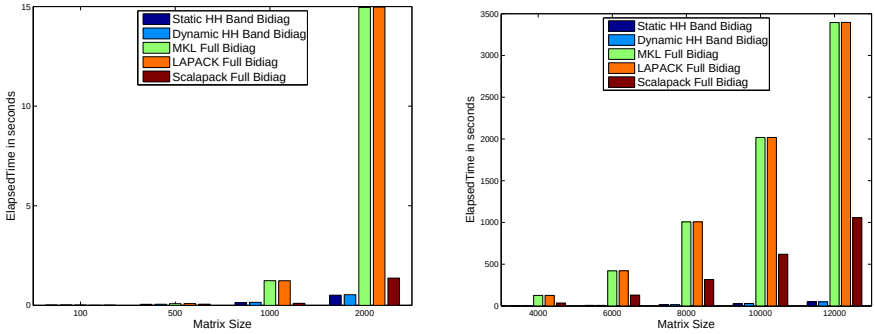
(a) Small Data Size. (b) Large Data Size.

Fig. 4. Elapsed time in seconds for the band bidiagonal reduction on a dual-socket quad-core Intel Itanium2 1.6 GHz with MKL BLAS V10.0.1.



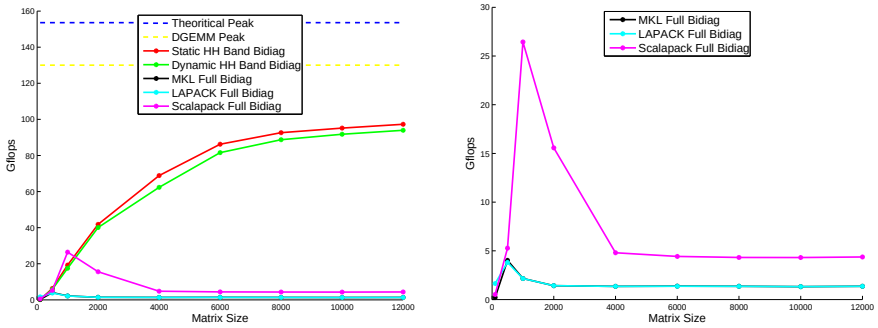
(a) Performance comparisons. (b) Zoom-in.

Fig. 5. Parallel Performance of the band bidiagonal reduction on a dual-socket quad-core Intel Itanium2 1.6 GHz processors with MKL BLAS V10.0.1.



(a) Small Data Size. (b) Large Data Size.

Fig. 6. Elapsed time in seconds for the band bidiagonal reduction on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS V10.0.1.



(a) Performance comparisons. (b) Zoom-in.

Fig. 7. Parallel Performance of the band bidiagonal reduction on a quad-socket quad-core Intel Xeon 2.4 GHz processors with MKL BLAS V10.0.1.

6 Related Work

Grosser and Lang [16] describe an efficient parallel reduction to bidiagonal form by splitting the standard algorithm in two stages, i.e., *dense to banded* and *banded to bidiagonal*, in the context of 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. However, those triangular blocks are then reduced without taking into account their sparsity, which add some extra flops.

Ralha [25] proposed a new approach for the bidiagonal reduction called one-sided bidiagonalization. The main concept is to implicitly tridiagonalize the matrix A^TA by a one-sided orthogonal transformation of A, i.e., $F = A\,V$. 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 could loose its orthogonality properties.

Barlow *et al.* [4] and later, Bosner *et al.* [6], 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 interlaced. 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.

7 Conclusion and Future Work

By exploiting the concepts of *algorithms-by-tiles* in the multicore environment, i.e., high level of parallelism with fine granularity and high performance data representation combined with a dynamic data driven execution, the BRD algorithm with Householder reflectors achieves 94 Gflop/s on a 12000×12000 matrix size with 16 Intel Tigerton 2.4 GHz processors. This algorithm performs most of the operations in Level-3 BLAS. Although the algorithm considerably surpasses in performance of the BRD algorithm of MKL, LAPACK and ScaLAPACK, its main inefficiency comes from the implementation of the kernel operations. The most performance critical, DSSRFB, kernel only achieves roughly 61% of peak for the tile size used ($b = 200$) in the experiments. For comparison a simple call to the DGEMM routine easily crosses 85% of peak. Unlike DGEMM, however, DSSRFB is not a single call to BLAS, but is composed of multiple calls to BLAS in a loop (due to inner blocking), since the inefficiency. DSSRFB could easily achieve similar performance if implemented as a monolithic code and heavily optimized. Finally, this work can be extended to the BRD of any matrix sizes (m, n) by using the appropriate method depending on the ratio between both dimensions.

8 Acknowledgment

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

References

1. http://www.top500.org.
2. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.
3. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, third edition, 1999.

4. J. L. Barlow, N. Bosner, and Z. Drmač. A new stable bidiagonal reduction algorithm. *Linear Algebra and its Applications*, 397(1):35–84, Mar. 2005.
5. M. W. Berry, J. J. Dongarra, and Y. Kim. A highly parallel algorithm for the reduction of a nonsymmetric matrix to block upper-Hessenberg form. LAPACK Working Note 68, Department of Computer Science, University of Tennessee, Knoxville, inst-UT-CS:adr, feb 1994. UT-CS-94-221, February 1994.
6. N. Bosner and J. L. Barlow. Block and parallel versions of one-sided bidiagonalization. *SIAM J. Matrix Anal. Appl.*, 29(3):927–953, 2007.
7. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel Tiled QR Factorization for Multicore Architectures. LAPACK Working Note 191, July 2007.
8. T. F. Chan. An improved algorithm for computing the singular value decomposition. *ACM Transactions on Mathematical Software*, 8(1):72–83, Mar. 1982.
9. J. Choi, J. Demmel, I. Dhillon, J. Dongarra, Ostrouchov, S., A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK, a portable linear algebra library for distributed memory computers-design issues and performance. *Computer Physics Communications*, 97(1-2):1–15, 1996.
10. K. E. Danforth Christopher M. and M. Takemasa. Estimating and correcting global weather model error. *Monthly weather review*, 135(2):281–299, 2007.
11. E. Elmroth and F. G. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. In *Applied Parallel Computing, Large Scale Scientific and Industrial Problems, 4th International Workshop, PARA'98*, Umeå, Sweden, June 14-17 1998. Lecture Notes in Computer Science 1541:120-128. http://dx.doi.org/10.1007/BFb0095328.
12. E. Elmroth and F. G. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. & Dev.*, 44(4):605–624, 2000.
13. E. Elmroth and F. G. Gustavson. High-performance library software for QR factorization. In *Applied Parallel Computing, New Paradigms for HPC in Industry and Academia, 5th International Workshop, PARA 2000*, Bergen, Norway, June 18-20 2000. Lecture Notes in Computer Science 1947:53-63. http://dx.doi.org/10.1007/3-540-70734-4_9.
14. G. H. Golub and C. F. Van Loan. *Matrix Computation*. John Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.
15. Golub, G. H. and Kahan, W. Calculating the singular values and the pseudo inverse of a matrix. *SIAM J. Numer. Anal.*, 2:205–224, 1965.
16. B. Grosser and B. Lang. Efficient parallel reduction to bidiagonal form. *Parallel Comput.*, 25(8):969–986, 1999.
17. B. C. Gunter and R. A. van de Geijn. Parallel out-of-core computation and updating of the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, Mar. 2005.
18. J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equation on the CELL processor using Cholesky factorization. *Trans. Parallel Distrib. Syst.*, 19(9):1175–1186, 2008. http://dx.doi.org/10.1109/TPDS.2007.70813.
19. J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equations on the CELL processor using Cholesky factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1–11, Sept. 2008.
20. J. Kurzak and J. Dongarra. QR Factorization for the CELL Processor. LAPACK Working Note 201, May 2008.
21. B. Lang. Parallel reduction of banded matrices to bidiagonal form. *Parallel Computing*, 22(1):1–18, 1996.

22. 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)*.
23. E. S. Quintana-Ortí and R. A. van de Geijn. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software*, 35(2), July 2008.
24. G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. A. van de Geijn, and F. G. Van Zee. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *PDP*, pages 301–310. IEEE Computer Society, 2008.
25. Rui Ralha. One-sided reduction to bidiagonal form. *Linear Algebra and its Applications*, 358:219–238, Jan 2003.
26. R. Schreiber and C. Van Loan. A storage efficient WY representation for products of householder transformations. *SIAM J. Sci. Statist. Comput.*, 10:53–57, 1989.
27. G. W. Stewart. *Matrix Algorithms Volume I: Matrix Decompositions*. SIAM, Philadelphia, 1998.
28. L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.
29. E. L. Yip. Fortran subroutines for out-of-core solutions of large complex linear systems. *Technical Report CR-159142, NASA*, November 1979.