Bidiagonalization and R-Bidiagonalization: Parallel Tiled Algorithms, Critical Paths and Distributed-Memory Implementation

Mathieu Faverge*§, Julien Langou†, Yves Robert‡§, Jack Dongarra§¶
*Bordeaux INP, CNRS, Inria et Université de Bordeaux, France
†University of Colorado Denver, USA
‡Laboratoire LIP, École Normale Supérieure de Lyon, France
§University of Tennessee Knoxville, USA
¶University of Manchester, UK

***Abstract*—We study tiled algorithms for going from a "full" matrix to a condensed "band bidiagonal" form using orthogonal transformations: (i) the tiled bidiagonalization algorithm BIDIAG, which is a tiled version of the standard scalar bidiagonalization algorithm; and (ii) the R-bidiagonalization algorithm R-BIDIAG, which is a tiled version of the algorithm which consists in first performing the QR factorization of the initial matrix, then performing the band-bidiagonalization of the R-factor. For both BIDIAG and R-BIDIAG, we use four main types of reduction trees, namely FLATTS, FLATTT, GREEDY, and a newly introduced auto-adaptive tree, AUTO. We provide a study of critical path lengths for these tiled algorithms, which shows that (i) R-BIDIAG has a shorter critical path length than BIDIAG for tall and skinny matrices, and (ii) GREEDY based schemes are much better than earlier proposed algorithms with unbounded resources. We provide experiments on a single multicore node, and on a few multicore nodes of a parallel distributed shared-memory system, to show the superiority of the new algorithms on a variety of matrix sizes, matrix shapes and core counts.**

Keywords: bidiagonalization, R-bidiagonalization, critical path, greedy algorithms, auto-adaptive reduction tree.

I. INTRODUCTION

This work is devoted to the design and comparison of tiled algorithms for the bidiagonalization of large matrices. Bidiagonalization is a widely used kernel that transforms a full matrix into bidiagonal form using orthogonal transformations. In many algorithms, the bidiagonal form is a critical step to compute the singular value decomposition (SVD) of a matrix. The necessity of computing the SVD is present in many computational science and engineering areas. Based on the Eckart–Young theorem [14], we know that the singular vectors associated with the largest singular values represent the best way (in the 2-norm sense) to approximate the matrix. This approximation result leads to many applications, since it means that SVD can be used to extract the "most important" information of a matrix. We can use the SVD for compressing data or making sense of data. In this era of Big Data, we are interested in very large matrices. To reference one out of many application, SVD is needed for principal component analysis (PCA) in Statistics, a widely used method in applied multivariate data analysis.

We consider algorithms for going from a "full" matrix to a condensed "band bidiagonal" form using orthogonal transformations. We use the framework of "algorithms by tiles". Within this framework, we study: (i) the tiled bidiagonalization algorithm BIDIAG, which is a tiled version of the standard scalar bidiagonalization algorithm; and (ii) the R-bidiagonalization algorithm R-BIDIAG, which is a tiled version of the algorithm which consists in first performing the QR factorization of the initial matrix, then performing the band-bidiagonalization of the R-factor. For both bidiagonalization algorithms BIDIAG and R-BIDIAG, we use HQR-based reduction trees, where HQR stands for the Hierarchical QR factorization of a tiled matrix [12]. Considering various reduction trees gives us the flexibility to adapt to matrix shape and machine architecture. In this work, we consider many types of reduction trees. In shared memory, they are named FLATTS, FLATTT, GREEDY, and a newly introduced auto-adaptive tree, AUTO. In distributed memory, they are somewhat more complex and take into account the topology of the machine. The main contributions are the following:

- The design and comparison of the BIDIAG and R-BIDIAG tiled algorithms with many types of reduction trees. There is considerable novelty in this. Previous work [22], [23], [25], [27] on tiled bidiagonalization has only considered one type of tree (FLATTS tree) with no R-BIDIAG. Previous work [26] has considered GREEDY trees for only half of the steps in BIDIAG and does not consider R-BIDIAG. This paper is the first to study R-BIDIAG for tiled bidiagonalization algorithm. and to study GREEDY trees for both steps of the tiled bidiagonalization algorithm.
- A detailed study of critical path lengths for FLATTS, FLATTT, GREEDY with BIDIAG and R-BIDIAG (so six different algorithms in total), which shows that: (i) The newly-introduced GREEDY based schemes (BIDIAG and R-BIDIAG) are much better than earlier proposed variants with unbounded resources and no communication: for matrices of $p \times q$ tiles,

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.46

$p \geq q$, their critical paths have a length $\Theta(q \log_2(p))$ instead of $\Theta(pq)$ for FLATTS and FLATTT; (ii) BIDIAGGREEDY has a shorter critical path length than R-BIDIAGGREEDY for square matrices; it is the opposite for tall and skinny matrices, and the asymptotic ratio is $\frac{1}{1+\frac{\alpha}{2}}$ for tiled matrices of size $p \times q$ when $p = \beta q^{1+\alpha}$, with $0 \leq \alpha < 1$

• Implementation of our algorithms in DPLASMA [15], which runs on top of the PARSEC runtime system [4], and which enables parallel distributed experiments on multicore nodes. All previous tiled bidiagonalization study [22], [23], [25]–[27] were limited to shared memory implementation.

• Experiments on a single multicore node, and on a few multicore nodes of a parallel distributed shared-memory system, show the superiority of the new algorithms on a variety of matrix sizes, matrix shapes and core counts. AUTO outperforms its competitors in almost every test case, hence standing as the best algorithmic choice for most users.

The rest of the paper is organized as follows. Section II provides a detailed overview of related work. Section III describes the BIDIAG and R-BIDIAG algorithms with the FLATTS, FLATTT and GREEDY trees. Section IV is devoted to the analysis of the critical paths of all variants. Section V outlines our implementation, and introduces the new AUTO reduction tree. Experimental results are reported in Section VI. Conclusion and hints for future work are given in Section VII.

II. RELATED WORK

This section surveys the various approaches to compute the singular values of a matrix, and positions our new algorithm with respect to existing numerical software kernels.

Computing the SVD. Computing the SVD of large matrices in an efficient and scalable way, is an important problem that has gathered much attention. The matrices considered here are rectangular m-by-n, with $m \geq n$. We call GE2VAL the problem of computing (only) the singular values of a matrix, and GESVD the problem of computing the singular values and the associated singular vectors.

From full to bidiagonal form. Many SVD algorithms first reduce the matrix to bidiagonal form with orthogonal transformations (GE2BD step), then process the bidiagonal matrix to obtain the sought singular values (BD2VAL step). These two steps (GE2BD and BD2VAL) are very different in nature. GE2BD can be done in a known number of operations and has no numerical difficulties. On the other hand, BD2VAL requires the convergence of an iterative process and is prone to numerical difficulties. This paper mostly focuses on GE2BD: reduction from full to bidiagonal form. Clearly, GE2BD+BD2VAL solves GE2VAL: computing (only) the singular value of a matrix. If the singular vectors are desired (GESVD), one can also compute them by accumulating the "backward" transformations; in this example, this would consist in a VAL2BD step followed by a BD2GE step. Golub and Kahan [17] provides a singular value solver based on an initial reduction to bidiagonal form. In [17, Th. 1], the GE2BD step is done using a QR step on the first column, then an LQ step on the first row, then a QR step on the second column, etc. The steps are done one column at a time using Householder transformation. This algorithm is implemented as a Level-2 BLAS algorithm in LAPACK as `xGEBD2`. For an m-by-n matrix, the cost of this algorithm is (approximately) $4mn^2 - \frac{4}{3}n^3$.

Level 3 BLAS for GE2BD. Dongarra, Sorensen and Hammarling [13] explains how to incorporate Level-3 BLAS in LAPACK `xGEBD2`. The idea is to compute few Householder transformations in advance, and then to accumulate and apply them in block using the WY transform [2]. This algorithm is available in LAPACK (using the compact WY transform [29]) as `xGEBRD`. Großer and Lang [19, Table 1] explain that this algorithm performs (approximately) 50% of flops in Level 2 BLAS (computing and accumulating Householder vectors) and 50% in Level 3 BLAS (applying Householder vectors). In 1995, Choi, Dongarra and Walker [10] presents the SCALAPACK version, `PxGEBRD`, of the LAPACK `xGEBRD` algorithm of [13].

Multi-step approach. Further improvements for GE2BD (detailed thereafter) are possible. These improvements rely on combining multiple steps. These multi-step methods will perform in general much better for GE2VAL (when only singular values are sought) than for GESVD (when singular values and singular vectors are sought). When singular values and singular vectors are sought, all the "multi" steps have to be performed in "reverse" on the singular vectors adding a non-negligible overhead to the singular vector computation.

Preprocessing the bidiagonalization with a QR factorization (preQR step). Chan [9] explains that, for tall-and-skinny matrices, in order to perform less flops, one can pre-process the bidiagonalization step (GE2BD) with a QR factorization. In other words, Chan propose to do preQR(m,n)+GE2BD(n,n) instead of GE2BD(m,n). A curiosity of this algorithm is that it introduces nonzeros where zeros were previously introduced; yet, there is a gain in term of flops. Chan proves that the crossover points when preQR(m,n)+GE2BD(n,n) performs less flops than GE2BD(m,n) is when m is greater than $\frac{5}{3}n$. Chan also proved that, asymptotically, preQR(m,n)+GE2BD(n,n) will perform half the flops than GE2BD(m,n) for a fixed n and m going to infinity. If the singular vectors are sought, preQR has more overhead: (1) the crossover point is moved to more tall-and-skinny matrices, and there is less gain; also (2) there is some complication as far as storage goes.

Two-step approach: GE2BND+BND2BD. In 1999, Großer and Lang [19] studied a two-step approach for GE2BD: (1) go from full to band (GE2BND), (2) then go from band to bidiagonal (BND2BD). In this scenario, GE2BND has most of the flops and performs using Level-3 BLAS kernels; BND2BD is not using Level-3 BLAS but it executes much less flops and operates on a smaller data footprint that might fit better in cache. There is a trade-off for the bandwidth to be chosen. If the bandwidth is too small, then the first step (GE2BND) will have the same issues as GE2BD. If the bandwidth is too large, then the second step BND2BD will have many flops and dominates the run time.

Tiled Algorithms for the SVD. In the context of massive parallelism, and of reducing data movement, many dense linear algebra algorithms operates on *tiles* of the matrix, and tasks are scheduled thanks to a runtime. In the context of the SVD, tiled algorithms naturally leads to band bidiagonal form. Ltaief, Kurzak and Dongarra [25] present a tiled algorithm for GE2BND (to go from full to band bidiagonal form). Ltaief, Luszczek, Dongarra [27] add the second step (BND2BD) and present a tiled algorithm for GE2VAL using GE2BND+BND2BD+BD2VAL. Ltaief, Luszczek, and Dongarra [26] improve the algorithm for tall and skinny matrices by using "any" tree instead of flat trees in the QR steps. Haidar, Ltaief, Luszczek and Dongarra [23] improve the BND2BD step of [27]. Finally, in 2013, Haidar, Kurzak, and Luszczek [22] consider the problem of computing singular vectors (GESVD) by performing GE2BND+BND2BD+BD2VAL+VAL2BD+BD2BND+BND2GE. They show that the two-step approach (from full to band, then band to bidiagonal) can be successfully used not only for computing singular values, but also for computing singular vectors.

BND2BD step. The algorithm in LAPACK for BND2BD is xGBBRD. In 1996, Lang [24] improved the sequential version of the algorithm and developed a parallel distributed algorithm. Recently, PLASMA released an efficient multi-threaded implementation [23], [27], and Rajamanickam [28] also worked on this step.

BD2VAL step. Much research has been done on this kernel. Much software exists. In LAPACK, to compute the singular values and optionally the singular vectors of a bidiagonal matrix, the routine `xBDSQR` uses the Golub-Kahan QR algorithm [17]; the routine `xBDSDC` uses the divide-and-conquer algorithm [20]; and the routine `xBDSVX` uses bisection and inverse iteration algorithm. Recent research was trying to apply the MRRR (Multiple Relatively Robust Representations) method [31] to the problem.

BND2BD+BD2VAL steps in this paper. This paper focuses neither on BND2BD nor BD2VAL. As far as we are concerned, we can use any of the methods mentioned above. The faster these two steps are, the better for us. For this study, during the experimental section, for BND2BD, we use the PLASMA multi-threaded implementation [23], [27] and, for BD2VAL, we use LAPACK `xBDSQR`.

III. Tiled Bidiagonalization Algorithms

A. QR factorization

Tiled algorithms are expressed in terms of tile operations rather than elementary operations. Each tile is of size $n_b \times n_b$, where n_b is a parameter tuned to squeeze the most out of arithmetic units and memory hierarchy. Typically, n_b ranges from 80 to 200 on state-of-the-art machines [1]. Consider a rectangular tiled matrix A of size $p \times q$. The actual size of A is thus $m \times n$, where $m = pn_b$ and $n = qn_b$. In Algorithm 1, k is the step, and also the panel index, and $elim(i, piv(i,k), k)$ is an orthogonal transformation that combines rows i and $piv(i,k)$ to zero out the tile in position (i,k). To implement $elim(i, piv(i,k), k)$, one can use six different kernels, whose costs are given in Table I. In this table, the unit of time is the time to perform $\frac{n_b^3}{3}$ floating-point operations. There are two main possibilities. The first version eliminates tile (i,k) with the *TS (Triangle on top of square)* kernels, while the second version uses *TT (Triangle on top of triangle)* kernels. In a nutshell, *TT* kernels allow for more parallelism, using several eliminators per panel simultaneously, but they reach only a fraction of the performance of *TS* kernels. See [6], [12] or the extended version of this work [16] for details. There are many algorithms to compute the QR factorization of A, and we refer to [6] for a survey. We use the three following variants:

- **FlatTS**: This algorithm with *TS* kernels is the reference algorithm used in [7], [8]. At step k, the pivot row is always row k, and we perform the eliminations $elim(i,k,k)$ in sequence, for $i = k+1$, $i = k+2$ down to $i = p$.
- **FlatTT**: This algorithm is the counterpart of the FlatTS algorithm with *TT* kernels. It uses exactly the same elimination operations, but with different kernels.
- **Greedy**: This algorithm is asymptotically optimal, and turns out to be the most efficient on a variety of platforms [5], [12]. It eliminates many tiles in parallel at each step, using a reduction tree (see [6] for a detailed description).

Algorithm 1: $QR(p,q)$ algorithm for a tiled matrix of size (p,q).

```
for k = 1 to min(p, q) do
    Step k, denoted as QR(k):
    for i = k + 1 to p do
        elim(i, piv(i, k), k)
```

Algorithm 2: Step $LQ(k)$ for a tiled matrix of size $p \times q$.

```
Step k, denoted as LQ(k):
for j = k + 1 to q do
    col-elim(j, piv(j, k), k)
```

Operation	Panel		Update	
	Name	Cost	Name	Cost
Factor square into triangle	$GEQRT$	4	$UNMQR$	6
Zero square with triangle on top	$TSQRT$	6	$TSMQR$	12
Zero triangle with triangle on top	$TTQRT$	2	$TTMQR$	6

Table I: Kernels for tiled QR. The unit of time is $\frac{n_b^3}{3}$, where n_b is the blocksize.

B. Bidiagonalization

Consider a rectangular tiled matrix A of size $p \times q$, with $p \geq q$. The bidiagonalization algorithm BiDiag proceeds as the QR factorization, but interleaves one step of LQ factorization between two steps of QR factorization (see Figure 1). More precisely, BiDiag executes the sequence

$$QR(1); LQ(1); QR(2); LQ(2) \ldots QR(q-1); LQ(q-1); QR(q)$$

where $QR(k)$ is the step k of the QR algorithm (see Algorithm 1), and $LQ(k)$ is the step k of the LQ algorithm. The latter is a right factorization step that executes the column-oriented eliminations shown in Algorithm 2.

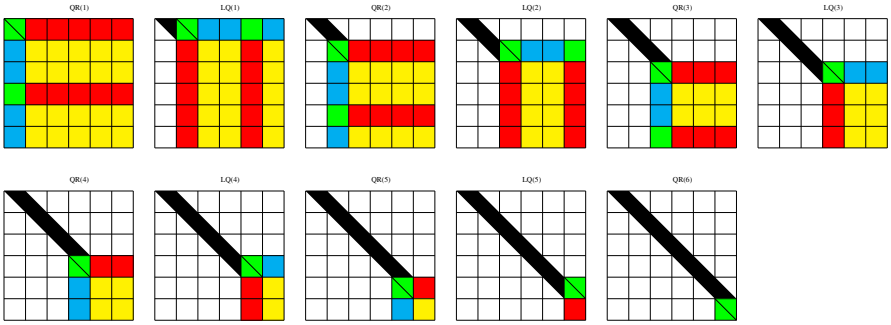


Figure 1: Snapshots of the bidiagonalization algorithm BIDIAG.

In Algorithm 2, *col-elim*$(j, piv(k,j), k)$ is an orthogonal transformation that combines columns j and $piv(k,j)$ to zero out the tile in position (k,j). It is the exact counterpart to the row-oriented eliminations $elim(i, piv(i,k), k)$ and be implemented with the very same kernels, either *TS* or *TT*.

C. R-Bidiagonalization

When p is much larger than q, R-bidiagonalization should be preferred, if minimizing the operation count is the objective. This R-BIDIAG algorithm does a QR factorization of A, followed by a bidiagonalization of the upper square $q \times q$ matrix. In other words, given a rectangular tiled matrix A of size $p \times q$, with $p \geq q$, R-BIDIAG executes the sequence

$$QR(p,q); LQ(1); QR(2); LQ(2); QR(3) \ldots LQ(q-1); QR(q)$$

Let $m = pn_b$ and $n = qn_b$ be the actual size of A (element wise). The number of arithmetic operations is $4n^2(m - \frac{n}{3})$ for BIDIAG and $2n^2(m+n)$ for R-BIDIAG [18, p.284]. These numbers show that R-BIDIAG is less costly than BIDIAG whenever $m \geq \frac{5n}{3}$, or equivalently, whenever $p \geq \frac{5q}{3}$. One major contribution of this paper is to provide a comparison of BIDIAG and R-BIDIAG in terms of parallel execution time, instead of operation count.

IV. CRITICAL PATHS

In this section, we compute exact or estimated values of the critical paths of the BIDIAG and R-BIDIAG algorithms with the FLATTS, FLATTT, and GREEDY trees.

A. Bidiagonalization

To compute the critical path, given the sequence executed by BIDIAG, we first observe that there is no overlap between two consecutive steps $QR(k)$ and $LQ(k)$. To see why, consider w.l.o.g. the first two steps $QR(1)$ and $LQ(1)$ on Figure 1. Tile $(1,2)$ is used at the end of the $QR(1)$ step to update the last row of the trailing matrix (whichever it is). In passing, note that all columns in this last row are updated in parallel, because we assume unlimited resources when computing critical paths. But tile $(1,2)$ is the first tile modified by the $LQ(1)$ step, hence there is no possible overlap. Similarly, there is no overlap between two consecutive steps $LQ(k)$ and $QR(k+1)$. Consider steps $LQ(1)$ and $QR(2)$ on Figure 1. Tile $(2,2)$ is used at the end of the $LQ(1)$ step to update the last column of the trailing matrix (whichever it is), and it is the first tile modified by the $QR(1)$ step.

As a consequence, the critical path of BIDIAG is the sum of the critical paths of each step. From [5], [6], [12] we have the following values for the critical path of one QR step applied to a tiled matrix of size (u,v):

FLATTS

$$QR - FTS_{1step}(u,v) = \begin{cases} 4 + 6(u-1) & \text{if } v = 1, \\ 4 + 6 + 12(u-1) & \text{otherwise.} \end{cases}$$

FLATTT

$$QR - FTT_{1step}(u,v) = \begin{cases} 4 + 2(u-1) & \text{if } v = 1, \\ 4 + 6 + 6(u-1) & \text{otherwise.} \end{cases}$$

GREEDY

$$QR - GRE_{1step}(u,v) = \begin{cases} 4 + 2\lceil \log_2(u) \rceil & \text{if } v = 1, \\ 4 + 6 + 6\lceil \log_2(u) \rceil & \text{otherwise.} \end{cases}$$

The critical path of one LQ step applied to a tiled matrix of size (u,v) is $LQ_{1step}(u,v) = QR_{1step}(v,u)$. Finally, in the BIDIAG algorithm, the size of the matrix for step $QR(k)$ is $(p-k+1, q-k+1)$ and the size of the matrix for step $LQ(k)$ is $(p-k+1, q-k)$. We derive the following values:

- **FLATTS**: $\text{BIDIAGFLATTS}(p,q) = 12pq - 6p + 2q - 4$
- **FLATTT**: $\text{BIDIAGFLATTT}(p,q) = 6pq - 4p + 12q - 10$
- **GREEDY**: $\text{BIDIAGGREEDY}(p,q) = \sum_{k=1}^{q-1}(10 + 6\lceil \log_2(p+1-k) \rceil) + \sum_{k=1}^{q-1}(10 + 6\lceil \log_2(q-k) \rceil) + (4 + 2\lceil \log_2(p+1-q) \rceil$

If q is a power of two, we derive that $\text{BIDIAGGREEDY}(q,q) = 12q\log_2(q) + 8q - 6\log_2(q) - 4$. If both p and q are powers of two, with $p > q$, we obtain $\text{BIDIAGGREEDY}(p,q) = 6q\log_2(p) + 6q\log_2(q) + 14q - 4\log_2(p) - 6\log_2(q) - 10$. For the general case, see [16] for the exact but complicated formula. Simpler bounds are obtained by rounding down and up the ceiling function in the logarithms [16]. Here, we content ourselves with an asymptotical analysis for large matrices. Take $p = \beta q^{1+\alpha}$, with $0 \leq \alpha$. We obtain that

$$\lim_{q \to \infty} \frac{\text{BIDIAGGREEDY}(\beta q^{1+\alpha}, q)}{(12 + 6\alpha) q \log_2(q)} = 1 \qquad (1)$$

Equation (1) shows that BIDIAGGREEDY is an order of magnitude faster than FLATTS or FLATTT. For instance when $\alpha = 0$, hence $p = \beta q$ are proportional (with $\beta \geq 1$), we have $\text{BIDIAGFLATTS}(\beta q, q) = 12\beta q^2 + O(q)$, $\text{BIDIAGFLATTT}(\beta q, q) = 6\beta q^2 + O(q)$, and $\text{BIDIAGGREEDY}(\beta q, q) = 12q\log_2(q) + O(q)$.

In fact, we have derived a stronger result: the optimal critical path of BIDIAG(p,q) with $p = \beta q^{1+\alpha}$ is asymptotically equivalent to $(12+6\alpha)q\log_2(q)$, regardless of the reduction tree used for each QR and LQ step: this is because GREEDY is optimal (up to a constant) for each step [5], hence BIDIAGGREEDY is optimal up to a linear factor in q, hence asymptotically optimal.

B. R-Bidiagonalization

Computing the critical path of R-BIDIAG is more difficult than for BIDIAG, because kernels partly overlap. For example, there is no need to wait for the end of the (left) QR factorization to start the first (right) factorization step $LQ(1)$. In fact, this step can start as soon as the first step $QR(1)$ is over because the first row of the matrix is no longer used throughout the whole QR factorization at this point. However, the interleaving of the following kernels gets quite intricate. Since taking it into account, or not, does not change the higher-order terms, in the following we simply sum up the values obtained without overlap, adding the cost of the QR factorization of size (p,q) to that of the bidiagonalization of the top square (q,q) matrix, and subtracting step $QR(1)$ as discussed above.

Due to lack of space, we refer to [16] for critical path values of R-BIDIAG (p,q) with FLATTS and FLATTT. Here, we concentrate on the most efficient tree GREEDY. The key result is the following: combining [5, Theorem 3.5] with [11, Theorem 3] we derive that the cost $QR-GRE$ of the QR factorization with GREEDY is $QR-GRE(p,q) = 22q+o(q)$ whenever $p = o(q^2)$. This leads to R-BIDIAGGREEDY$(p,q) \le (22q+o(q)) + (12q\log_2(q) + (20-12\log_2(e))q + o(q)) - o(q) = 12q\log_2(q) + (42-12\log_2(e))q + o(q)$ whenever $p = o(q^2)$.

Again, we are interested in the asymptotic analysis of R-BIDIAGGREEDY, and in the comparison with BIDIAG. In fact, when $p = o(q^2)$, say $p = \beta q^{1+\alpha}$, with $0 \le \alpha < 1$, the cost of the QR factorization $QR(p,q)$ is negligible in front of the cost of the bidiagonalization BIDIAGGREEDY(q,q), so that R-BIDIAGGREEDY(p,q) is asymptotically equivalent to BIDIAGGREEDY(q,q), and we derive that:

$$\lim_{q\to\infty} \frac{\text{BIDIAGGREEDY}(\beta q^{1+\alpha}, q)}{\text{R-BIDIAGGREEDY}(\beta q^{1+\alpha}, q)} = 1 + \frac{\alpha}{2} \qquad (2)$$

Asymptotically, BIDIAGGREEDY is at least as costly (with equality is p and q are proportional) and at most 1.5 times as costly as R-BIDIAGGREEDY (the maximum ratio being reached when $\alpha = 1-\varepsilon$ for small values of ε.

Just as before, R-BIDIAGGREEDY is asymptotically optimal among all possible reduction trees, and we have proven the following result, where for notation convenience we let BIDIAG(p,q) and R-BIDIAG(p,q) denote the optimal critical path lengths of the algorithms::

Theorem 1. *For $p = \beta q^{1+\alpha}$, with $0 \le \alpha < 1$:*

$$\lim_{q\to\infty} \frac{\text{BIDIAG}(p,q)}{(12+6\alpha)q\log_2(q)} = 1, \lim_{q\to\infty} \frac{\text{BIDIAG}(p,q)}{\text{R-BIDIAG}(p,q)} = 1 + \frac{\alpha}{2}$$

When p and q are proportional ($\alpha = 0$, $\beta \ge 1$), both algorithms have same asymptotic cost $12qlog_2(q)$. On the contrary, for very elongated matrices with fixed $q \ge 2$, the ratio of the critical path lengths of BIDIAG and R-BIDIAG gets high asymptotically: the cost of the QR factorization is equivalent to $6\log_2(p)$ and that of BIDIAG(p,q) to $6q\log_2(p)$. Since the cost of BIDIAG(q,q) is a constant for fixed q, we get a ratio of q. Finally, to give a more practical insight, we provide detailed comparisons of all schemes in [16].

C. Switching from BIDIAG to R-BIDIAG

For square matrices, BIDIAG is better than R-BIDIAG. For tall and skinny matrices, this is the opposite. For a given q, what is the ratio $\delta = p/q$ for which we should switch between BIDIAG and R-BIDIAG? Let δ_s denote this crossover ratio. The question was answered by Chan [9] when considering the operation count, showing that the optimal switching point between BIDIAG and R-BIDIAG when singular values only are sought is $\delta = \frac{5}{3}$. We consider the same question but when critical path length (instead of number of flops) is the objective function. We provide some experimental data in [16], focusing on BIDIAGGREEDY R-BIDIAGGREEDY and writing some code snippets that explicitly compute the critical path lengths for given p and q, and find the intersection for a given q. Altogether, we find that δ_s is a complicated function of q. oscillating between 5 and 8.

V. IMPLEMENTATION

To evaluate experimentally the impact of the different reduction trees on the performance of the GE2BND and GE2VAL algorithms, we have implemented both the BIDIAG and R-BIDIAG algorithms in the DPLASMA library [15], which runs on top of the PARSEC runtime system [4]. PARSEC is a high-performance fully-distributed scheduling environment for generic data-flow algorithms. It takes as input a problem-size-independent, symbolic representation of a Directet Acyclic Graph (DAG) in which each node represents a task, and each edge a dependency, or data movement, from one task to another. PARSEC schedules those tasks on distributed parallel machine of multi-cores, potentially heterogeneous, while complying with the dependencies expressed by the programmer. At runtime, task executions trigger data movements, and create new ready tasks, following the dependencies defined by the DAG representation. The runtime engine is responsible for actually moving the data from one machine (node) to another, if necessary, using an underlying communication mechanism, like MPI. Tasks that are ready to compute are scheduled according to a data-reuse heuristic: each core will try to execute close successors of the last task it ran, under the assumption that these tasks require data that was just touched by the terminated one. This policy is tuned by the user through a priority function: among the tasks of a given core, the choice is done following this function. To balance load between the cores, tasks of a same cluster in the algorithm (reside on a same shared memory machine) are shared between the computing cores, and a NUMA-aware job stealing policy is

implemented. The user is then responsible only to provide the algorithm, the initial data distribution, and potentially the task distribution. The last one is usually correlated to the data distribution when the (default) owner-compute rule is applied. In our case, we use a $2D$ block-cyclic data distribution as used in the SCALAPACK library, and we map the computation together with the data. A full description of PARSEC can be found in [4].

The implementation of the BIDIAG and R-BIDIAG algorithms have then been designed as an extension of our previous work on HQR factorization [12] within the DPLASMA library. The HQR algorithm proposes to perform the tiled QR factorization of a $(p \times q)$-tile matrix, with $p \geq q$, by using a variety of trees that are optimized for both the target architecture and the matrix size. It relies on multi-level reduction trees. The highest level is a tree of size R, where R is the number of rows in the $R \times C$ two-dimensional grid distribution of the matrix, and it is configured by default to be a flat tree if $p \geq 2q$, and a Fibonacci tree otherwise. The second level, the domino level, is an optional intermediate level that enhances the pipeline of the lowest levels when they are connected together by the highest distributed tree. It is by default disabled when $p \geq 2q$, and enabled otherwise. Finally, the last two levels of trees are use to create parallelism within a node and work only on local tiles. They correspond to a composition of one or multiple FLATTS trees that are connected together with an arbitrary tree of TT kernels. The bottom FLATTS tree enables highly efficient kernels while the TT tree on top of it generates more parallelism to feed all the computing resources from the architecture. The default is to have FLATTS trees of size 4 that are connected by a GREEDY tree in all cases. This design is for QR trees, a similar design exists for LQ trees. Using these building blocks, we have crafted an implementation of BIDIAG and R-BIDIAG within the abridged representation used by PARSEC to represent algorithms. This implementation is independent of the type of trees selected for the computation, thereby allowing the user to test a large spectrum of configuration without the hassle of rewriting all the algorithm variants.

One important contribution is the introduction of two new tree structures dedicated to the BIDIAG algorithm. The first tree, GREEDY, is a binomial tree which reduces a panel in the minimum amount of steps. The second tree, AUTO, is an adaptive tree which automatically adapts to the size of the local panel and number of computing resources. We developed the auto-adaptive tree to take advantage of (i) the higher efficiency of the TS kernels with respect to the TT kernels, (ii) the highest degree of parallelism of the GREEDY tree with respect to any other tree, and (iii) the complete independence of each step of the BIDIAG algorithm, which precludes any possibility of pipelining. Thus, we propose to combine in this configuration a set of FLATTS trees connected by a GREEDY tree, and to automatically adapt the number of FLATTS trees, and by construction their sizes, a, to provide enough parallelism to the available computing resources. Given a matrix of size $p \times q$, at each step k, we need to apply a QR factorization on a matrix of size $(p-k-1) \times (q-k-1)$, the number of parallel tasks available at the step beginning of the step is given by $\lceil (p-k-1)/a \rceil * (q-k-1)$. Note that we consider the panel as being computed in parallel of the update, which is the case when a is greater than 1, with an offset of one time unit. Based on this formula, we compute a at each step of the factorization such that the degree of parallelism is greater than a quantity $\gamma \times nb_{cores}$, where γ is a parameter and nb_{cores} is the number of cores. For the experiments, we set $\gamma = 2$. Finally, we point out that AUTO is defined for a resourced-limited platform, hence computing its critical path would have no meaning, which explains a posteriori that it was not studied in Section IV.

VI. EXPERIMENTS

In this section, we evaluate the performance of the proposed algorithms for the GE2BND kernel against existing competitors.

A. Architecture

Experiments are carried out using the PLAFRIM experimental testbed[1]. We used up to 25 nodes of the `miriel` cluster, each equipped with 2 Dodeca-core Haswell Intel Xeon E5-2680 v3 and 128GB of memory. The nodes are interconnected with an Infiniband QDR TrueScale network with provides a bandwidth of 40Gb/s. All the software are compiled with gcc 4.9.2, and linked against the sequential BLAS implementation of the Intel MKL 11.2 library. For the distributed runs, the MPI library used is OpenMPI 2.0.0. The practical GEMM performance is of 37 GFlop/s on one core, and 642 GFlop/s when the 24 cores are used. For each experiment, we generated a matrix with prescribed singular values using LAPACK `LATMS` matrix generator and checked that the computed singular values were satisfactory up to machine precision.

B. Competitors

This paper presents new parallel distributed algorithms and implementations for GE2BND using DPLASMA. To compare against competitors on GE2VAL, we follow up our DPLASMA GE2BND implementation with the PLASMA multi-threaded BND2BD algorithm, and then use the Intel MKL multi-threaded BD2VAL implementation. We thus obtain GEVAL by doing GE2BND+BND2BD+BD2VAL.

It is important to note that we do not use parallel distributed implementations neither for BND2BD nor for BD2VAL. We only use shared memory implementations for these two last steps. Thus, for our distributed memory runs, after the GE2BND step in parallel distributed using DPLASMA, the band is gathered on a single node, and BND2BD+BD2VAL is performed by this node while all all other nodes are left idle. We will show that, despite this current limitation for parallel distributed, our implementation outperforms its competitors.

[1]Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS, see https://www.plafrim.fr/.

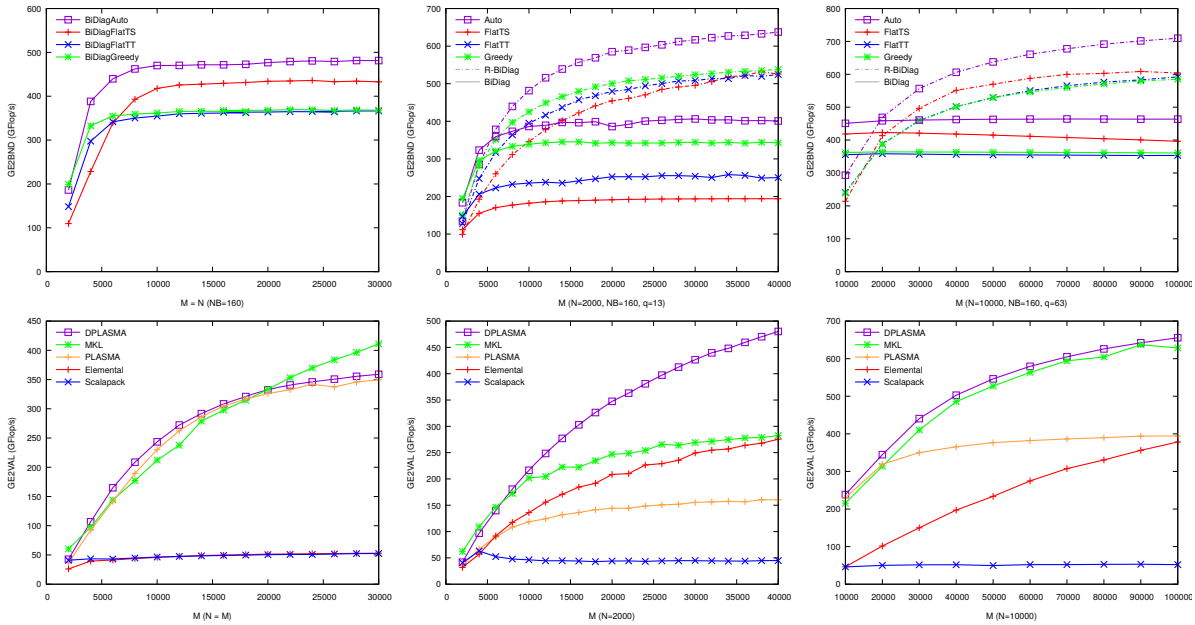


Figure 2: Shared memory performance of the multiple variants for the GE2BND algorithm on the first row, and for the GE2VAL algorithm on the second row, using a single 24 core node of the `miriel` cluster.

On the square test cases, only 23 cores of a 24-core node were used for computation, and the 24^{th} core was left free to handle MPI communications progress. The implementation of the algorithm is available in a public fork of the DPLASMA library at https://bitbucket.org/mfaverge/parsec.

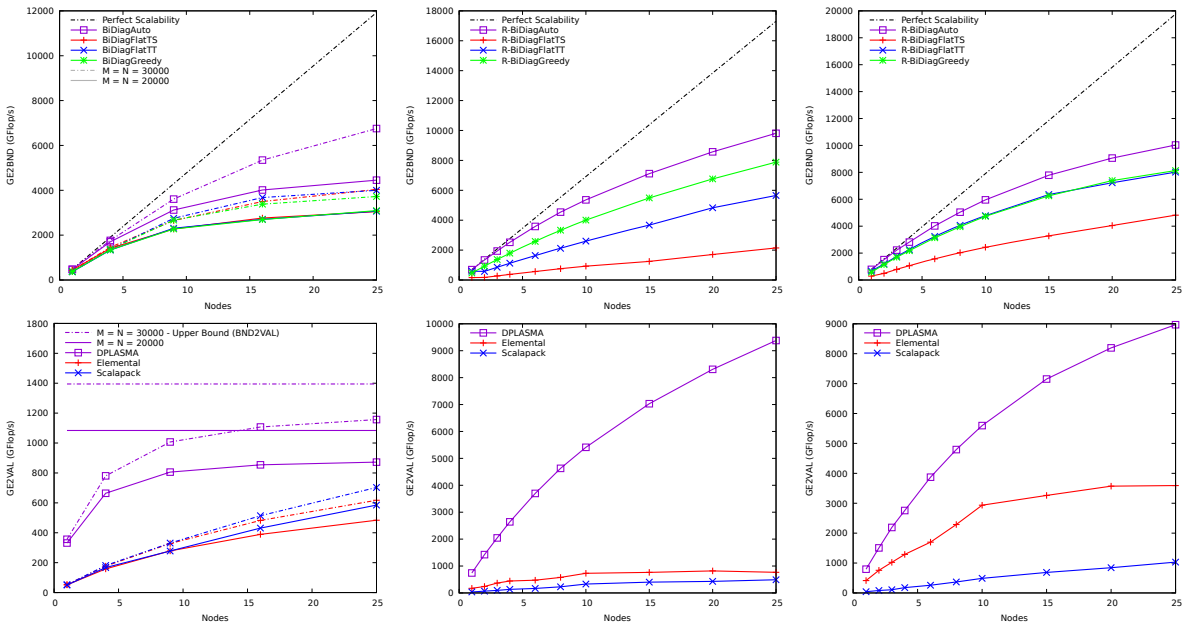
PLASMA is the closest alternative to our proposed solution but it is only using FLATTS as its reduction tree, and is limited to single-node platform, and is supported by a different runtime. For both our code, and PLASMA, the tile size parameter is critical to get good performance: a large tile size will get an higher kernel efficiency and a faster computation of the band, but it will increase the number of flops of the BND2BD step which is heavily memory bound. On the contrary, a small tile size will speed up the BND2BD step by fitting the band into cache memory, but decreases the efficiency of the kernels used in the GE2BND step. We tuned the n_b (tile size) and i_b (internal blocking in *TS* and *TT* kernels) parameters to get the better performance on the square case $m = n = 20000$, and $m = n = 30000$ on the PLASMA code. The selected values are $n_b = 160$, and $i_b = 32$. We used the same parameters in the DPLASMA implementation for both the shared memory runs and the distributed ones. The PLASMA 2.8.0 library was used.

Intel MKL proposes an multi-threaded implementation of the GE2VAL algorithm which gained an important speedup while switching from version 11.1 to 11.2 [30]. While it is unclear which algorithm is used beneath, the speedup reflects the move to a multi-stage algorithm. **Intel MKL** is limited to single-node platforms.

SCALAPACK implements the parallel distributed version of the LAPACK GEBRD algorithm which interleaves phases of memory bound BLAS2 calls with computational bound BLAS3 calls. It can be used either with one process per core and a sequential BLAS implementation, or with a process per node and a multi-threaded BLAS implementation. The latter being less efficient, we used the former for the experiments. The blocking size n_b is critical to get performances since it impacts the phase interleaving. We tuned the n_b parameter to get the better performance on a single node with the same test cases as for PLASMA, and $n_b = 48$ was selected.

Elemental implements an algorithm similar to SCALAPACK, but it automatically switches to Chan's algorithm [9] when $m \geq 1.2n$. As for SCALAPACK, it is possible to use it as a single MPI implementation, or an hybrid MPI-thread implementation. The first one being recommended, we used this solution. Tuning of the n_b parameter similarly to previous libraries gave us the value $nb = 96$. A better algorithm developed on top of the LibFLAME [21] is provided by Elemental, but this one is used only when singular vectors are sought.

In the following, we compare all these implementation on the `miriel` cluster with 3 main configurations: (i) square matrices; (ii) tall and skinny matrices with $n = 2,000$; this choice restricts the level of parallelism induced by the number of panels to half the cores; and (iii) tall and skinny matrices with $n = 10,000$: this choice enables for more parallelism. For all performance comparisons, we use the same operation count as in [3, p. 123] for the GE2BND and GE2VAL algorithms. The BD2VAL step has a negligible cost $O(n^2)$. For R-BIDIAG, we use the same number of flops as for BIDIAG; we



Square ($m = n$) Tall & skinny ($m = 2,000,000, n = 2,000$) Tall & skinny ($m = 1,000,000, n = 10,000$)

Figure 3: Distributed memory performance of the multiple variants for the GE2BND and the GE2VAL algorithms, respectively on the top and bottom row, on the `miriel` cluster. Grid data distributions are $\sqrt{nb_{nodes}} \times \sqrt{nb_{nodes}}$ for square matrices, and $nb_{nodes} \times 1$ for tall and skinny matrices. For the square case, solid lines are for $m = n = 20,000$ and dashed lines for $m = n = 30,000$.

do not assess the absolute performance of R-BIDIAG, instead we provide a direct comparison with BIDIAG.

C. Shared Memory

The top row of Figure 2 presents the performance of the three configurations selected for our study of GE2BND. On the top left, the square case perfectly illustrates the strengths and weaknesses of each configuration. On small matrices, FLATTT in blue and GREEDY in green illustrate the importance of creating algorithmically more parallelism to feed all resources. However, on large size problems, the performance is limited by the lower efficiency of the TT kernels. The FLATTS tree behaves at the opposite: it provides better asymptotic performance thanks to the TS kernels, but lacks parallelism when the problem is too small to feed all cores. AUTO is able to benefit from the advantages of both GREEDY and FLATTS trees to provide a significant improvement on small matrices, and a 10% speedup on the larger matrices.

For the tall and skinny matrices, we observe that the R-BIDIAG algorithm (dashed lines) quickly outperforms the BIDIAG algorithm, and is up to 1.8 faster. On the small case ($n = 2,000$), the crossover point is immediate, and both FLATTT and GREEDY, exposing more parallelism, are able to get better performances than FLATTS. On the larger case ($n = 10,000$), the parallelism from the larger matrix size allows FLATTS to perform better, and to postpone the crossover point due to the ratio in the number of flops. In both cases, AUTO provides the better performance with an extra 100 GFlop/s.

On the bottom row of Figure 2, we compare our best solutions, namely AUTO tree with BIDIAG for square cases and with R-BIDIAG on tall and skinny cases, to the competitors on the GE2VAL algorithm. The difference between our solution and PLASMA, which is using the FLATTS tree, is not as impressive due to the additional BND2BD and BD2VAL steps which have limited parallel efficiency. Furthermore, in our implementation, due to the change of runtime, we cannot pipeline the GE2BND and BND2BD steps to partially overlap the second step. However these two solutions still provide a good improvement over MKL which is slower on the small cases but overtakes at larger sizes. For such sizes, Elemental and SCALAPACK are not able to scale and reach up a maximum of 50 Gflop/s due to their highly memory bound algorithm.

On the tall and skinny cases, differences are more emphasized. We see the limitation of using only the BIDIAG algorithm on MKL, PLASMA and SCALAPACK, while our solution and elemental keep scaling up with matrix size. We also observe that MKL behaves correctly on the second test case, while it quickly saturates in the first one where the parallelism is less important. In that case, we are able to reach twice the MKL performance.

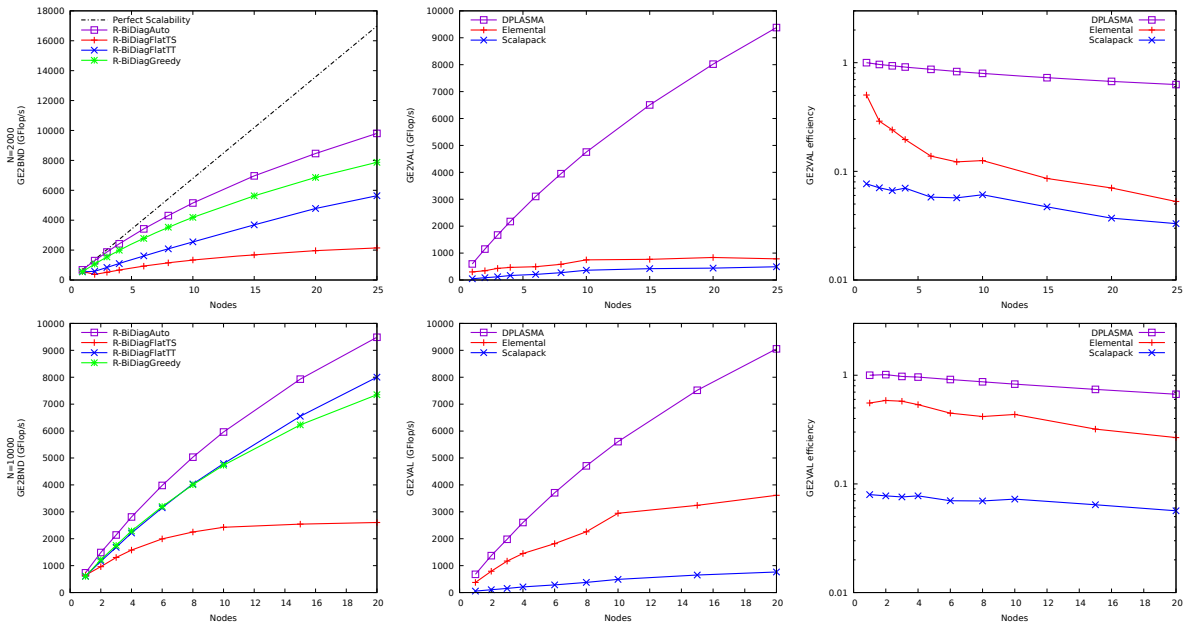


Figure 4: Study of the distributed weak scalability on tall and skinny matrices of size $(80,000\ nb_{nodes}) \times 2,000$ on the first row, and $(100,000\ nb_{nodes}) \times 10,000$ on the second row. First column presents the GE2BND performance, second column the GE2VAL performance, and third column the GE2VAL scaling efficiency.

D. Distributed Memory

a) Strong Scaling: Figure 3 presents a scalability study of the three variants on 4 cases: two square matrices with BIDIAG, and two tall and skinny matrices with R-BIDIAG. For all of them, we couple high-level distributed trees, and low-level shared memory trees. FLATTS and FLATTT configuration are coupled with a high level flat tree, while GREEDY and AUTO are coupled with a high level GREEDY tree. The configuration of the preQR step is setup similarly, except for AUTO which is using the automatic configuration described previously.

On all cases, performances are as expected. FLATTS, which is able to provide higher efficient kernels, hardly behaves better on the large square case; GREEDY, which provides better parallelism, is the best solution out of the three on the first tall and skinny case. We also observe the impact of the high level tree: GREEDY doubles the number of communications on square cases [12], which impacts its performance and gives an advantage to the flat tree which performs half the communication volume. Overall, AUTO keeps taking benefit from its flexibility, and scales well despite the fact that local matrices are less than 38×38 tiles, so less than 2 columns per core.

When considering the full GE2VAL algorithm on Figure 3, we observe a huge drop in the overall performance. This is due to the integration of the shared memory BND2BD and BD2VAL steps which do not scale when adding more nodes. For the the square case, we added the upper bound that we cannot beat due to those two steps. However, despite this limitation, our solution brings an important speedup to algorithms looking for the singular values, with respect to the competitors presented here. Elemental again benefits from the automatic switch to the R-BIDIAG algorithm, which allows a better scaling on tall and skinny matrices. However, it surprisingly reaches a plateau after 10 nodes where the performance stops increasing significantly. Our solution automatically adapts to create more or fewer parallelism, and reduces the amount of communications, which allows it to sustain a good speedup up to 25 nodes (600 cores).

b) Weak Scaling: Figure 4 presents a weak scalability study with tall and skinny matrices of width $n = 2,000$ on the first row, and $n = 10,000$ on the second row[2]. As previously, FLATTS quickly saturates due to its lack of parallelism. FLATTT is able to compete with, and even to outperform, GREEDY on the larger case due to its lower communication volume. AUTO offers a better scaling and is able to reach 10 TFlop/s which represents 400 to 475 GFlop/s per node. When comparing to Elemental and SCALAPACK on the GE2VAL algorithm, the proposed solution offers a much better scalability. Both Elemental and SCALAPACK suffer from their memory bound BIDIAG algorithm. With the switch to a R-BIDIAG algorithm, Elemental is able to provide better performance than SCALAPACK, but the lack of scalability of the Elemental QR factorization compared to the HQR implementation quickly limits the overall performance of the GE2VAL implementation.

[2]Experiments for the $n = 10,000$ case stop at 20 nodes due to the 32 bit integer default interface for all libraries

VII. Conclusion

In this paper, we have presented the use of many reduction trees for tiled bidiagonalization algorithms. We proved that, during the bidiagonalization process, the alternating QR and LQ reduction trees cannot overlap. Therefore, minimizing the time of each individual tree will minimize the overall time. Consequently, if one considers an unbounded number of cores and no communication, one will want to use a succession of greedy trees. We show that BiDiagGreedy is asymptotically much better than previously presented approaches with FlatTS. In practice, in order to have an effective solution, one has to take into account load balancing and communication, hence we propose trees that adapt to the parallel distributed topology (highest level tree) and enable more sequential but faster kernels on a node (Auto). We have also studied R-bidiagonalization in the context of tiled algorithms. While R-bidiagonalization is not new, it had never been used in the context of tiled algorithms. Previous work was comparing bidiagonalization and R-bidiagonalization in term of flops, while our comparison is conducted in term of critical path lengths. We show that bidiagonalization has a shorter critical path than R-bidiagonalization, that this is the opposite for tall and skinny matrices, and provide an asymptotic analysis. Along all this work, we give detailed critical path lengths for many of the algorithms under study. Our implementation is the first parallel distributed tiled algorithm implementation for bidiagonalization. We show the benefit of our approach (DPLASMA) against existing software on a multicore node (PLASMA, Intel MKL, Elemental and ScaLAPACK), and on a few multicore nodes (Elemental and ScaLAPACK) for various matrix sizes, for computing the singular values of a matrix. Future work will be devoted to gain access to a large distributed platform with a high count of multicore nodes, and to assess the efficiency and scalability of our parallel distributed BiDiag and R-BiDiag algorithms. Other research directions are the following: (i) investigate the trade-off of our approach when singular vectors are requested; a previous study [22] in shared memory was conclusive for FlatTS and no R-BiDiag (square matrices only); the question is to study the problem on parallel distributed platforms, with or without R-BiDiag, for various shapes of matrices and various trees; and (ii) develop a scalable parallel distributed BND2BD step; for now, for parallel distributed experiments on many nodes, we are limited in scalability by the BND2BD step, since it is performed using the shared memory library PLASMA on a single node.

Acknowledgements Work by J. Langou was partially supported by NSF award 1054864 and NSF award 1645514.

References

[1] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC '09*, pages 1–12. IEEE Computer Society Press, 2009.

[2] C. Bischof and C. V. Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13, 1987.

[3] S. Blackford and J. J. Dongarra. Installation guide for LAPACK. Technical Report 41, LAPACK Working Note, June 1999.

[4] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1–2):37–51, 2012. Extensions for Next-Generation Parallel Programming Models.

[5] H. Bouwmeester. *Tiled Algorithms for Matrix Computations on Multicore Architectures*. PhD thesis, Colorado University DEnver, 2013. Available at http://arxiv.org/abs/1303.3182.

[6] H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert. Tiled QR factorization algorithms. In *SC'2011*. ACM Press, 2011.

[7] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008.

[8] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[9] T. F. Chan. An improved algorithm for computing the singular value decomposition. *ACM Trans. Math. Softw.*, 8(1):72–83, Mar. 1982.

[10] J. Choi, J. J. Dongarra, and D. W. Walker. The design of a parallel dense linear algebra software library: Reduction to Hessenberg, tridiagonal, and bidiagonal form. *Numerical Algorithms*, 10(2):379–399, 1995.

[11] M. Cosnard, J.-M. Muller, and Y. Robert. Parallel QR decomposition of a rectangular matrix. *Numerische Mathematik*, 48:239–249, 1986.

[12] J. Dongarra, M. Faverge, T. Herault, M. Jacquelin, J. Langou, and Y. Robert. Hierarchical QR factorization algorithms for multi-core cluster systems. *Parallel Computing*, 39(4-5):212–232, 2013.

[13] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1):215 – 227, 1989.

[14] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1:211–218, 1936.

[15] G. B. et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *IPDPS Workshops*, pages 1432–1441, 2011.

[16] M. Faverge, J. Langou, Y. Robert, and J. Dongarra. Bidiagonalization with parallel tiled algorithms. Research Report 8969, INRIA, Oct. 2016.

[17] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *SIAM J. Numerical Analysis*, 2(2):205–224, 1965.

[18] G. H. Golub and C. F. V. Loan. *Matrix computations*. Johns Hopkins, 1989.

[19] B. Großer and B. Lang. Efficient parallel reduction to bidiagonal form. *Parallel Comput.*, 25(8):969–986, Aug. 1999.

[20] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the bidiagonal svd. *SIAM Journal on Matrix Analysis and Applications*, 16(1):79–92, 1995.

[21] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, Dec. 2001.

[22] A. Haidar, J. Kurzak, and P. Luszczek. An improved parallel singular value algorithm and its implementation for multicore hardware. In *SC '13*, pages 90:1–90:12. ACM, 2013.

[23] A. Haidar, H. Ltaief, P. Luszczek, and J. Dongarra. A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In *IPDPS 2012*, pages 25–35, 2012.

[24] B. Lang. Parallel reduction of banded matrices to bidiagonal form. *Parallel Computing*, 22(1):1 – 18, 1996.

[25] H. Ltaief, J. Kurzak, and J. Dongarra. Parallel two-sided matrix reduction to band bidiagonal form on multicore architectures. *IEEE TPDS*, 21(4):417–423, Apr. 2010.

[26] H. Ltaief, P. Luszczek, and J. Dongarra. Enhancing parallelism of tile bidiagonal transformation on multicore architectures using tree reduction. In *PPAM 2011*, pages 661–670. Springer, 2012.

[27] H. Ltaief, P. Luszczek, and J. Dongarra. High-performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures. *ACM Trans. Math. Softw.*, 39(3):16:1–16:22, May 2013.

[28] S. Rajamanickam. *Efficient algorithms for sparse singular value decomposition*. PhD thesis, University of Florida, 2009.

[29] R. Schreiber and C. V. Loan. A storage-efficient WY representation for products of householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(1):53–57, 1989.

[30] K. E. K. Vipin. Significant performance improvement of symmetric eigensolvers and SVD in Intel MKL 11.2, 2014.

[31] P. R. Willems, B. Lang, and C. Vömel. Computing the bidiagonal svd using multiple relatively robust representations. *SIAM Journal on Matrix Analysis and Applications*, 28(4):907–926, 2006.