Leveraging Task-Based Polar Decomposition Using PARSEC on Massively Parallel Systems

Dalal Sukkari, Hatem Ltaief and David Keyes
Extreme Computing Research Center
King Abdullah University of Science and Technology
Thuwal, Jeddah 23955 Saudi Arabia
Email: Firstname.Lastname@kaust.edu.sa

Mathieu Faverge
Bordeaux INP - Inria - CNRS - Univ. of Bordeaux
Talence, 33400 France
Email: Mathieu.Faverge@inria.fr

***Abstract*—This paper describes how to leverage a task-based implementation of the polar decomposition on massively parallel systems using the PARSEC dynamic runtime system. Based on a formulation of the iterative QR Dynamically-Weighted Halley (QDWH) algorithm, our novel implementation reduces data traffic while exploiting high concurrency from the underlying hardware architecture. First, we replace the most time-consuming classical QR factorization phase with a new hierarchical variant, customized for the specific structure of the matrix during the QDWH iterations. The newly developed hierarchical QR for QDWH exploits not only the matrix structure, but also shortens the length of the critical path to maximize hardware occupancy. We then deploy PARSEC to seamlessly orchestrate, pipeline, and track the data dependencies of the various linear algebra building blocks involved during the iterative QDWH algorithm. PARSEC enables to overlap communications with computations thanks to its asynchronous scheduling of fine-grained computational tasks. It employs look-ahead techniques to further expose parallelism, while actively pursuing the critical path. In addition, we identify synergistic opportunities between the task-based QDWH algorithm and the PARSEC framework. We exploit them during the hierarchical QR factorization to enforce a locality-aware task execution. The latter feature permits to minimize the expensive inter-node communication, which represents one of the main bottlenecks for scaling up applications on challenging distributed-memory systems. We report numerical accuracy and performance results using well and ill-conditioned matrices. The benchmarking campaign reveals up to 2X performance speedup against the existing state-of-the-art implementation for the polar decomposition on 36,864 cores.**

I. Introduction

As the scientific community confronts the challenges and embraces the opportunities at the dawn of the exascale era, leveraging existing numerical algorithms becomes critical for scaling up applications on supercomputers. This often means that algorithmic redesign is inevitable in order to effectively exploit the unprecedented level of thread concurrency promised by these massively parallel systems.

This paper focuses on optimizing the polar decomposition (PD) [1] of dense matrices on distributed-memory systems. The PD is an important component in the solution of symmetric eigenvalue problems and the computation of singular value decompositions [2]. It is also an essential component in continuum mechanics, in aerospace computations [3], and in chemistry [4]. In particular, this paper revisits the PD variant based on the iterative QR-based dynamically weighted Halley (QDWH) iteration, initially introduced by Nakatsukasa et al. [5], [6], and highlights optimization steps towards democratizing it on current and upcoming large-scale machines. The process to deploy QDWH-based PD algorithm on systems with million of cores requires a profound algorithmic reformulation and a disruptive shift from the traditional bulk synchronous to asynchronous task-based programming model. The challenges involved behind both major game-changing optimizations are further exacerbated by the fact that QDWH-based PD is not a single matrix operation, but instead to a collection of advanced dense linear algebra, i.e., QR/Cholesky factorizations/solvers, matrix-matrix multiplication, matrix norm computations, and condition number estimation. In other words, these optimizations have to be applied in a holistic manner across all computational phases of the QDWH-based PD algorithm to ultimately maximize performance.

One of the traditional key ingredients to scale up a numerical algorithm to high node counts is to mitigate expensive inter-node data movement in favor of computations on intra-node, locally cached data. Moreover, the bulk synchronous programming model, which has driven the research agenda of the dense linear algebra community for the last two decades [7], needs to be replaced with a more asynchronous task-based programming model for synchronization-reducing purposes. We use the PARSEC dynamic runtime system [8] to seamlessly orchestrate, pipeline, and track the data dependencies of the various tasks generated from the dense linear algebra building blocks involved during the iterative QDWH-based PD algorithm. Thanks to fine-grained task scheduling, PARSEC is capable of overlapping communications with computations. Based on tile algorithms [9]–[11], the resulting QDWH-based PD algorithm translates then into a directed acyclic graph (DAG), where vertices and edges represent tasks and data dependencies, respectively. PARSEC also detects and takes advantage of look-ahead opportunities to further expose parallelism, while actively pursuing the critical path of the DAG.

Given this scalability objective, we replace the classical QR factorization of the QDWH iterations with a novel Hierar-

978-1-7281-4734-5/19/$31.00

chical QR (HQR) formulation [12]. Customized for QDWH, HQR significantly reduces data traffic, while exploiting the matrix sparsity structure during the QDWH iterations. We also develop the first implementation of the matrix two-norm computations on distributed-memory using PARSEC. Although this matrix operation is not the most time-consuming in QDWH iterations, it still has to be ported to PARSEC due to the methodology employed to maximize performance. We additionally integrate existing communication-reducing algorithms into QDWH-based PD, such as the SUMMA algorithm [13] for the matrix-matrix multiplication. Last but not least, we compose the different matrix operation phases to further mitigate data traffic overheads, while increasing data locality. This latter process is rather manual and may turn out to be tedious. Automatic DAG composition remains still an open research problem.

The algorithmic adaptations and the paradigm shift needed in the bulk synchronous programming model create synergism situations, which may help PARSEC promoting a locality-aware task execution. Although the QDWH-based PD herein represents the targeted algorithm, some of the optimization techniques are not specific to QDWH-PD and may be used toward improving a broader class of dense linear algebra algorithms and applications on exascale systems [11], [14]–[18].

We conduct a detailed performance campaign using well and ill-conditioned matrices on distributed-memory systems. We assess the scalability of QDWH-based PD using PARSEC. The benchmarking campaign reveals up to 2X performance speedup against the state-of-the-art SCALAPACK implementation for QDWH-based PD using up to 36,864 cores. The performance gap between SCALAPACK and PARSEC implementations widens as the number of nodes increases, which shows the suitability of our implementation moving forward with extreme-scale systems.

The remainder of the paper is organized as follows. Section II provides a literature review on the latest PD developments. Section III highlights the contributions of this paper and compares them to previous work. Section IV and Section V briefly recall the QDWH-based PD algorithm and the PARSEC dynamic runtime system, respectively. Section VI introduces the new hierarchical QR variant customized for QDWH, while Section VII presents the additional mandatory algorithmic optimizations. Section VIII describes holistic high performance implementations of the resulting task-based QDWH-PD algorithm using PARSEC. Section IX demonstrates the numerical robustness and reports on the thorough performance benchmarking and profiling campaigns. We conclude in Section X.

II. Related Work

The polar decomposition (PD) exhibits high numerical accuracy with backward stability properties, which renders it an algorithm of choice in several scientific applications [1]. The downside of the PD algorithm is its high algorithmic complexity, whether the PD variant is based on the singular value decomposition (SVD) [2], [19], on an iterative method such as Newton which requires the explicit matrix inversion at each iteration, or on the inverse-free iterative QR dynamically-weighted Halley algorithm (QDWH) [5], [6]. The latter QDWH-based PD variant has attracted lots of interest due to its high degree of parallelism, as opposed to the other aforementioned approaches, which may compensate for the the excessive number of floating-point operations (flops).

In fact, the first high performance implementation of the QDWH-based PD runs on single shared-memory node and relies on multiple GPU hardware accelerators [20], which are capable of running the compute-bound QDWH kernels at a rate of execution by an order of magnitude higher than standard x86 CPU. This first attempt used the MAGMA library [21], the LAPACK [22] for GPUs software package. In fact, MAGMA extends the fork-join paradigm of LAPACK, which is at the core of the bulk synchronous programming model, in order to tackle heterogeneous shared-memory systems.

This initial work was followed by new QDWH-based PD implementations for distributed-memory, homogeneous x86 machines available in the Elemental [23] and POLAR [24] libraries. The latest developments of QDWH-based PD within POLAR permit to further improve its performance thanks to a topology-aware grid of processors [25]. POLAR is now fully integrated into the Cray Scientific numerical Library v17.11.1 and v19.02.1. (LibSci) [26]. To our knowledge, this is the fastest QDWH-based PD implementation freely available.

However, POLAR exploits parallelism only within each computational phase of the QDWH-based PD algorithm. The look-ahead techniques are not feasible due to the rigidness of the two-dimensional block cyclic data descriptor (2DBCDD), inherited from the SCALAPACK library [7]. This may result into a limited hardware occupancy, especially in strong scaling mode of operation. To overcome the performance bottlenecks raised by the bulk synchronous programming model, the first asynchronous task-based QDWH-based PD implementation demonstrates a performance gain across a myriad of shared-memory systems possibly equipped with GPUs [27]. Based on tile algorithms [9]–[11], the task-based QDWH-based PD implementation outperforms its predecessors on the same platforms, thanks to pipelining fine-grained tasks across the computational phases of QDWH-based PD. Moreover, the fine task granularity permits to exploit the matrix sparsity structure during the QR factorization required for the QDWH iterations, which reduces the overall algorithmic complexity. Integrated into CHAMELEON [11], it relies on STARPU [28], a dynamic runtime system, which is in charge of scheduling the various fine-grained tasks on the homogeneous as well as heterogeneous hardware architectures. Although STARPU runs on distributed-memory environments, its sequential task flow (STF) programming model limits its capability to provide support for efficient collective communication, e.g., broadcast messages. Indeed, as the task discovery occurs at runtime in the order the tasks are queued, it is difficult to identify communication patterns. All tasks involved may have been reordered (e.g., thanks to task priority), or perhaps not all

1: **/* Estimate the condition number */**
2: dlacpy(A, U) ▷ $U = A$
3: dlacpy(A, B) ▷ $B = A$
4: Anorm = dlange(A) ▷ $\|A\|_1$
5: dgenm2(A, α) ▷ $\alpha \approx \|A\|_2$
6: **/* Compute U_0 and l_0 */**
7: dlascl(U, $1./\alpha$) ▷ $U_0 = A/\alpha$
8: dgeqrf(B) ▷ $A = QR$
9: dtrtri(B) ▷ Compute R^{-1}
10: Ainvnorm = dlantr(B) ▷ $\approx \|A^{-1}\|_1$
11: $l_0 = 1./(Ainvnorm * Anorm)$
12: $l_0 = (\alpha/1.1) * l_0$
13:
14: **/* Compute the polar decomposition $A = U_p H$ using QDWH */**
15: $k = 1,\ Li = l_0,\ conv = 100$
16: **while** $(conv \geq \sqrt[3]{5eps}\ ||\ |Li - 1| \geq 5eps)$ **do**
17: $L2 = Li^2,\ dd = \sqrt[3]{(4(1 - L2)/L2^2)}$
18: $sqd = \sqrt{1 + dd}$
19: $a1 = sqd + \sqrt{8 - 4 \times dd + 8(2 - L2)/(L2 \times sqd)})/2$
20: $a = real(a1);\ b = (a - 1)^2/4;\ c = a + b - 1$
21: $Li = Li(a + b \times L2)/(1 + cL2)$
22: dlacpy(U, U1) ▷ Backup U_{k-1}
23:
24: **/* Compute U_k from U_{k-1} */**
25: **if** $c > 100$ **then**
26: $C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} \sqrt{c}U_{k-1} \\ I \end{bmatrix}$
27: dgeqrf(C) ▷ $C = QR = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R$
28: dorgqr(C) ▷ $C = Q = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}$
29: dgemm(Q_1, $Q_2^\top$, U) ▷ $U_k = \frac{1}{\sqrt{c}}\left(a - \frac{b}{c}\right) Q_1 Q_2^\top + \frac{b}{c} U_{k-1}$
30: **else**
31: dlaset(Z, 0., 1.) ▷ Z = I
32: dgemm($U^\top$, U, Z) ▷ $Z_k = I - cU_{k-1}^\top U_{k-1}$
33: dgeadd(U, B) ▷ $B = U_{k-1}^\top$
34: dposv(Z, B) ▷ Solve $Z_k x = U_{k-1}^\top$
35: dgeadd(B, U) ▷ $U_k = \frac{b}{c} U_{k-1} + \left(a - \frac{b}{c}\right)(U_{k-1} W_{k-1}^{-1}) W_{k-1}$
36: **end if**
37: dgeadd(U, U1) ▷ $U_k - U_{k-1}$
38: dlange(U1, conv) ▷ $conv = \|U_k - U_{k-1}\|_F$
39: $k = k + 1$
40: **end while**
41:
42: **/* Compute H */**
43: dgemm(U_k, A, H) ▷ $H = U_p^\top A$
44: dlacpy(H, B) ▷ $B = H$
45: dgeadd(B, H) ▷ $H = \frac{1}{2}(H + H^\top)$

Fig. 1. Pseudo-code for the QDWH-based PD algorithm using LAPACK.

submitted yet, when the communication is triggered (e.g, a dependency is released). While this should not be a showstopper on shared-memory systems, data movement may engender a significant overhead when moving forward with distributed-memory machines.

In this paper, we extend [27] and introduce the first asynchronous task-based implementation of the QDWH-based PD algorithm on distributed-memory architectures using instead the PARSEC dynamic runtime system [8]. PARSEC uses a Domain-Specific Language (DSL) called Job Data Flow (JDF) to express the algorithm as a Parametrized Task Graph (PTG), which enables to express collective communications directly from the language. In addition to choosing the PARSEC framework, which triggers the paradigm shift compared to [25] as far as the programming model is concerned, the novel algorithmic adaptations are paramount in deploying the asynchronous task-based implementation of the QDWH-based PD on large-scale systems.

III. CONTRIBUTIONS

We list below our main contributions, which constitute the crux of the paper:

- We deploy the first asynchronous QDWH-based PD algorithm using the PARSEC [8] dynamic tasking framework on distributed-memory systems.
- We reformulate critical dense matrix operations involved in the QDWH-based PD algorithms. The traditional QR factorization is replaced with a novel, customized version of hierarchical QR [12] for QDWH-based PD to reduce communication volume. The first matrix two-norm computation implemented using the domain specific language from PARSEC. The SUMMA algorithm is incorporated in the QDWH-based PD to improve the matrix-matrix multiplication kernel. DAG composition within QDWH iterations by fusing the DAGs of successive matrix operations is proposed in order to enforce locality-aware task execution.
- We report on the performance impact of the four algorithmic optimizations, conduct performance comparisons against the state-of-the-art SCALAPACK implementation, and demonstrate the performance scalability on up to 36,864 cores.

IV. BACKGROUND ON THE POLAR DECOMPOSITION

The Polar Decomposition (PD) of the matrix $A \in \mathbb{R}^{m \times n}$ $(m \geq n)$ is written $A = U_p H$, where U_p is an orthogonal matrix and $H = \sqrt{A^\top A}$ is a symmetric positive semidefinite matrix. Fig. 1 recalls the pseudo-code for the QDWH-based PD algorithm based on LAPACK notation. There are four main computational phases in the QDWH-based PD algorithm: (1) the computation of the condition number estimate, (2) The QR-based QDWH iteration (lines 26-29), (3) the Cholesky-based QDWH iteration (lines 31-35), and (4) once converged, the calculation of the symmetric positive semi-definite polar factor H. (lines 43-45).

We use the inverse-free QDWH-based iterative procedure [6] to calculate the PD as follows:

$$\begin{aligned} X_0 &= A/\alpha, \\ \begin{bmatrix} \sqrt{c_k} X_k \\ I \end{bmatrix} &= \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R, \\ X_{k+1} &= \frac{b_k}{c_k} X_k + \frac{1}{\sqrt{c_k}} \left(a_k - \frac{b_k}{c_k}\right) Q_1 Q_2^\top,\ k \geq 0. \end{aligned} \tag{1}$$

When, X_k becomes well-conditioned, it is possible to replace Equation 1 with a Cholesky-based iteration variant as follows:

$$\begin{aligned} X_{k+1} &= \frac{b_k}{c_k} X_k + \left(a_k - \frac{b_k}{c_k}\right)(X_k W_k^{-1}) W_k^{-\top}, \\ W_k &= \text{chol}(Z_k),\ Z_k = I + c_k X_k^\top X_k. \end{aligned} \tag{2}$$

As shown in line 25 of Fig. 1, this algorithmic switch at runtime allows to further speed up the overall computation, thanks to a lower algorithmic complexity, while still maintaining numerical stability. Therefore, it is clear that the overall algorithmic complexity, in terms of flops, depends on

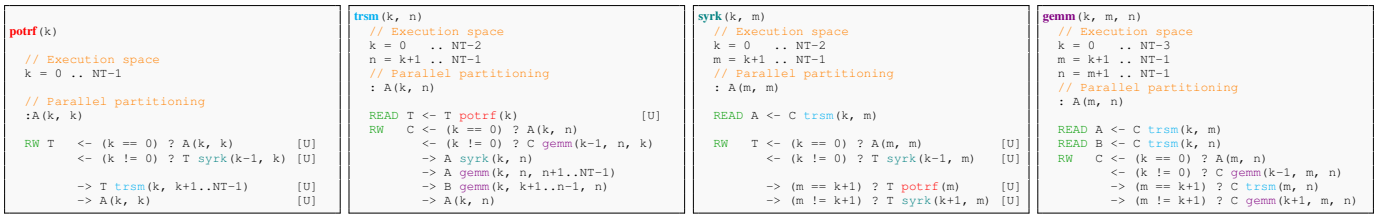


Fig. 2. PTG pseudo-code of the Cholesky factorization of an upper triangular matrix for the PARSEC runtime.

the number of iterations required to converge. The number of iterations is solely determined by the matrix condition number. The theoretical analysis in [6] shows that the upper-bound for the number of iterations is six, assuming double precision arithmetic. All in all, the pseudo-code of the QDWH-based PD algorithm is presented in Fig. 1 using LAPACK naming convention. If the initial matrix is ill-conditioned (this is our worst-case scenario), QDWH-based PD typically performs an initial condition estimate of the matrix, enters the inner iteration loop to perform QR factorization for the first three iterations, reduces the condition number c to less than 100 and switches to Cholesky-based QDWH iterations, performs three more Cholesky-based iterations and reaches numerical convergence for the U_p polar factor, and finally, compute the H polar factor after exiting the main computational loop. However, if the initial matrix is well-conditioned (i.e., best case scenario), Fig. 1 shows the performance when only a single QR followed by a single Cholesky-based QDWH iterations is required. To make the paper self-contained, we recall the algorithmic complexity of Fig. 1. As explained in [25], the overall number of flops is $(\frac{2}{3})n^3+(8+\frac{2}{3})n^3\times\#it_{QR}+(4+\frac{1}{3})n^3\times\#it_{Chol}+2n^3$. The total flop count of QDWH for dense matrices ranges then from $(11+\frac{1}{3})n^3$ (for $l_0 \approx 1$ with $\#it_{Chol} = 2$) to $(41+\frac{2}{3})n^3$ (for $l_0 \gg 1$, i.e., for the worst case scenario of ill-conditioned matrix, with typically $\#it_{QR} = 3$ and $\#it_{Chol} = 3$). Besides dealing with the challenge of a numerical algorithm defined as a collection of successive matrix operations, the overall algorithmic complexity of QDWH-based PD may represent a serious showstopper in efficiently porting the algorithm onto massively parallel systems. To ease this process, we rely on the dynamic runtime system PARSEC [8] to expose fine-grained parallelism from the QDWH-based PD algorithm, while overlapping communication with computation.

V. The PARSEC Dynamic Runtime System

PARSEC [29] is a generic data-flow engine for heterogeneous distributed-memory systems which executes task-based algorithms. It aims at supporting multiple domain specific languages which helps the programmer to describe an algorithm in its own high-level concepts that are later translated to the direct acyclic graph (DAG) representation by a dedicated compiler. It actually supports two programming models: the Sequential Task Flow, similar to STARPU and OPENMP (a.k.a. Dynamic Task Discovery), and the Parameterized Task Graph (PTG) [8]. Figures 2 and 3 show the pseudo-codes of

```
for (k = 0; k < NT; k+)
  potrf( RW, A[k][k] );
  for (n = k+1; n < NT; n++)
    trsm( READ, A[k][k], RW, A[k][n] );
  for (m = k+1; m < NT; m++)
    syrk( READ, A[k][m], RW, A[m][m] );
    for (n = m+1; n < NT; n++) {
      gemm( READ, A[k][m], READ, A[k][n],
            RW, A[m][n] );
```

Fig. 3. STF pseudo-code of the Cholesky factorization of an upper triangular matrix for the STARPU runtime.

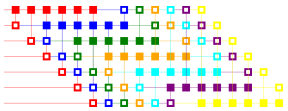
the Cholesky factorization the PTG used by PARSEC model and for the STF programming used by STARPU, respectively. The STF directly extracts the parallelism and the task dependencies from the nested loops thanks to hints provided by the programmer (READ, RW, WRITE) to describe the data accesses. The dependencies are then inferred from the submission order. The PTG exploits an abridge representation of the graph described in [30] from which all the knowledge of the algorithm is extracted. This compact representation enables the engine to easily traverse the DAG from any node without a complete knowledge. This programming model, although more advanced, allows the developer to provide more information to the underlying runtime system. The local knowledge of the dependencies helps the many scheduling strategies available to pursue local dependencies and to greatly improve the cache locality at runtime. As discussed in Section II, collective communications are explicitly described, which enable the runtime to optimize them with communication trees, for example. This can be seen in the output edge of the potrf toward the trsm tasks in the same row. Additionally, the edges can be annotated to adapt the communication to the type of data (e.g. [U] for upper triangle in Fig. 2).

Note that all these dependencies can be either computed through external functions, or described with a user structure. This makes the PTG programming model a good candidate to implement challenging task graphs such as the complex reduction trees targeted in this work, as opposed to the STF model which would require to submit all tasks following a reverse depth first search algorithm to preserve the data dependencies.

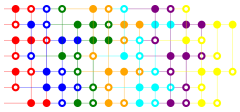
VI. The New Hierarchical QR Factorization

The most complex and expensive functions of the QDWH-based PD algorithm are the QR-based iteration (lines 26-29 of Fig. 1), and the QR factorization included in the computation

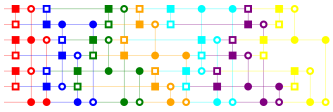
(a) $Flat(0)$. A flat tree is applied and the main row is always the first one ($cp = 42$).



(b) $Flat(k)$. A flat tree is applied and the main row is selected by round-robin ($cp = 18$).



(c) $Greedy(x)$. A Greedy tree is applied to reduce as much as possible the critical path. The main row may be any row ($cp = 15$).



(d) $Mix(x)$. A combination of efficient flat trees, and a greedy tree is applied. The main row may be any row ($cp = 21$).

Fig. 4. Impact of the different tree strategies on the critical path of the QR factorization of the lower matrix in the `xTPQRT` kernel. The example shows the reduction trees for a 7×7 tiles matrix. Each color represents one step of the factorization, and `xTSQRT` (resp. `xTTQRT`) kernels are represented by square (resp. by circle).

of the condition number estimate (line 8). QR factorization has incrementally improved in the last decade with the advent of task-based runtime systems. Tiled-QR algorithm were first introduced in [31], [32] for the PLASMA library, and in [33] for the Elemental Library, both targeting shared-memory systems. In these works, the factorization of the panel was split into tiles and the diagonal tile was used as a single *phagocyte* to sequentially eliminate all other tiles in the panel. Later, [34] introduced reduction trees of any sort to factorize each panel in a general manner.

Based upon this work, multiple types of reduction trees: binary, greedy, and Fibonacci have been introduced in multicore shared-memory implementations [35], [36], and eventually with two levels of reduction trees to handle distributed-memory nodes of multicore [35], [37]. These later works were especially targeting tall and skinny matrices ($m \gg n$). More recently, [12], [38] proposed more advanced trees matching the hierarchy of the targeted architecture and adapted to all shapes of matrices. We herein propose to extend this work for QDWH-based PD algorithm.

We recall that these reduction trees are built on top of two families of kernels. The `TS` family reduces square matrices. While these are highly efficient kernels (`xTSQRT`, `xTSMQR`), however, they force the reduction tree to be flat, preventing from generating additional parallelism. The `TT` family (`xTTQRT`, `xTTMQR`) reduces upper triangular matrices, which occur when multiple domains have been independently reduced to create parallelism or to avoid communication. The drawback is the lower arithmetic intensity of these kernels. Although both have been merged to handle the general pentagonal case under the names `xTPQRT` and `xTPMQRT` in LAPACK, we may still refer to the former naming convention to distinguish them. Based on these kernel specifications, three levels of trees are exploited in [12], [38]. A first level uses `xTSQRT` family and a flat tree to provide data locality and kernel efficiency. On top of it, a second level uses a reduction tree (flat, binary, greedy, or Fibonacci) of `xTTQRT` kernels to provide more parallelism within each shared-memory node. A third tree level allows for reducing the tiles between the different nodes while greatly decreasing the amount of communications. One last option, called domino, improves the coupling of the middle and high level trees to increase the potential pipeline of the operations, which turns out to be particularly efficient for the tall and skinny cases.

While all these solutions have improved the scalability of QR algorithms, QR still focuses on factorizing a single matrix to generate an upper triangular R matrix. The reduction trees are thus optimized to reduce all the rows $r \in [\![k, MT]\!]$ at each iteration k. In the context of the QDWH-based PD algorithm, the QR iteration performs: $C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = QR = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R$, which can be seen as the factorization of two independent tiled matrices C_1 and C_2, that are later combined. This is even more true when the factorization of C_1 in the condition number estimate is reused to save flops. This construction naturally provides two domains that can be factorized independently in parallel and then coupled together as in the communication-avoiding QR algorithm. On the one hand, the factorization of the C_1 matrix is a classic algorithm, and can thus be performed with trees introduced in [12]. On the other hand, the reduction trees of the C_2 matrix must be optimized for the rows $r \in [\![1, MT]\!]$, and to go further for $r \in [\![1, k]\!]$ if we exploit the identity structure of C_2 in the QDWH-based PD to avoid extra flops [27].

Fig. 4 illustrates the pipeline of the QR factorization steps of the general shape C_2 matrix of size 7×7 tiles and with different configuration of reduction trees. Each color represents an iteration of the QR factorization. Each reduction with a `xTSQRT` (resp. `xTTQRT`) is represented by a couple of squares (resp. circles). The filled one is used to reduce the empty one. Thus, the number of couples per columns represents the amount of row parallelism available at each step. The top one, $Flat(0)$ (4a), presents the first implementation using a single level flat tree. One observes, that this solution, as opposed to the classic QR factorization, does not pipeline at all due to the fact that the row used to reduce all others is the same at each iteration. Based on the *domino* idea from [12], the second one, $Flat(k)$ (4b), illustrates that we can have a regular pipeline of the operations by shifting the *main* row at each iteration. The critical path is thus reduced from 42 to 18 steps. The third one, $Greedy(x)$ (4c), exploits the later idea that any row can be picked at each iteration to be the main one to apply a greedy tree. The critical path is further

1: **function** GEQRF(C_1, C_2)
2: **for** $k = 0$ to $NT - 1$ **do**
3: $R1_{k,k:*}$ = FACTORIZEONEPANEL($C1_{k:*,k:*}$)
4: $R2_{i,k:*}$ = FACTORIZEONEPANEL($C2_{0:*,k:*}$)
5: $R_{k,k:*}$ = REDUCEROWS($R1_{k,k:*}, R2_{i,k:*}$)
6: **end for**
7: **end function**

Fig. 5. Pseudo-code of the factorization of a matrix composed of two-submatrices on top of each other $C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R$.

reduced to 15 steps. However, as it has been shown in [12], [38], using only triangular kernels can lead to poor efficiency, and a mix of both square kernels for efficiency and optimized trees of triangle kernels for parallelism must be combined to provide the highest performance. The bottom one, $Mix(x)$ (4d), illustrates the combination of a flat tree of size 2 with a greedy reduction tree. The critical path is slightly increased to 21 but this is balanced by the higher arithmetic intensity of the kernels at runtime. With an additional tree level to handle distributed-memory systems, we apply the last one in this work as a generalization of the HQR algorithm to efficiently factorize any matrix shapes.

Once we are able to provide efficient reduction trees for any shape: lower triangular (e.g. C_1), general (e.g. C_2), or upper triangular (e.g. C_2 is identity), the QR factorization of the matrix C can be simply described as in Fig. 5. At each iteration, the factorization of the panels of C_1 and C_2 provides two upper triangles that are later reduced together to finalize the factorization. Note that, as explained above, the R_2 matrix can result from any row i of C_2.

VII. Additional Algorithmic Optimizations

In order to implement an efficient QDWH-based PD algorithm for distributed-memory machines on top of the PaRSEC runtime systems, we discuss in this section the additional algorithmic optimizations that needs to be considered toward the final optimized implementation.

A. Two-Norm Matrix Estimator

One of the first requirements in the QDWH-based PD algorithm is the availability of a two-norm estimator to scale the matrix and improve the convergence rate. (line 5 of Fig 1). Indeed, although it has low complexity, the lack of its PaRSEC variant represents a major showstopper for leveraging the overall task-based QDWH-based PD algorithm.

B. SUMMA-Based Matrix-Matrix Multiplication

The second major cost of computation in the QDWH-based PD algorithm iterative loop is the multiplication of two $N \times N$ matrices. An efficient distributed-memory matrix-matrix implementation is required once per iteration (lines 29 and 32 of Fig. 1), and one time at the end of the algorithm (line 43). The literature is rich in describing efficient matrix-matrix multiplications. The best known implementation is the SUMMA algorithm [13] used by ScaLAPACK. More recently, Solomonik and Demmel [39] proposed a 2.5D algorithm exploiting additional buffers to perform partial sums that are later reduced together similarly to what we proposed for the two-norm estimator. The communication reduction is enabled at the cost of a large memory overhead since the C matrix ($C = \alpha AB + \beta C$) is replicated multiple times. Then, Demmel et al. [40] proposed CARMA, a recursive algorithm that reduces the volume of communications but performs only with power of two processes.

C. DAG Composition for Cholesky-Based Solve

The PaRSEC runtime system provides a composition of high level algorithms to ease the job submission to the user, but it is unfortunately not yet capable of fine-grained composition. Indeed, while it is available through the sequential task flow (STF) programming model, the runtime is not yet able to compose elementary tasks in algorithms described with the PTG programming model. Fusing these DAGs may reduce the number of communications thanks to data locality. We propose here to discuss the impact on the `POSV` operation which performs a Cholesky factorization `POTRF`, followed by two triangular solves `TRSM`. If we consider the problem as being three independent algorithms submitted to the runtime, one observes that the factorized matrix may be communicated three times: 1) during the factorization to update the trailing submatrix, 2) during the forward substitution (first `TRSM`), and 3) during the backward substitution (second `TRSM`). Furthermore, it has been shown with modern task-based implementations that the factorization step and the first triangular solve can efficiently pipeline. Indeed, this pipeline removes one of the two main synchronization points in that sequence. In a distributed-memory environment, it can also avoid to communicate A multiple times.

VIII. Implementation Details

We now describe the implementation details of the previous algorithmic optimizations in the context of the DPLASMA library [14], a high-performance dense linear algebra library powered by the PaRSEC engine on distributed-memory machines.

A. Hierarchical QR Factorization for QDWH

The first algorithm optimization corresponds to a major extension of the Hierarchical QR algorithm from [12] to more general problems, including the QDWH-based PD algorithm. This previous work relies on pre-defined formulas that enable the runtime to compute any edges in the tree from any nodes quickly without extra-memory. The complexity induced to support the reduction of any matrix shapes makes it difficult to describe through formula all these relations. Therefore, we propose to rely on a pre-computed tree structure that will be used at runtime to span the tree. This is an additional memory cost but negligible with respect to the storage of the matrix. The tree structure is built with a dynamic greedy algorithm that tracks the latest update performed on a row. As in [12], we structure our algorithm on three levels: the top

```
1: function GeneratePartialTree(tree, L)
2:     /* Flat tree */
3:     if tree == Flat then
4:         i = PopRowWithSmallestDate(L)
5:         while L is not empty do
6:             j = PopRowWithSmallestDate(L)
7:             Set row i as pivot of row j for step k, and update their dates
8:         end while
9:     else
10:        /* Greedy binary tree */
11:        (i, j) = PopCoupleWithSmallestDate(L)
12:        while j is not NULL do
13:            Set row i as pivot of row j for step k, and update their dates
14:            Push(i, L)
15:            (i, j) = PopCoupleWithSmallestDate(L)
16:        end while
17:    end if
18:    return i
19: end function
```

Fig. 6. The generic partial tree function to generate flat or greedy reduction tree with all the elements of a list.

```
1: function GenerateTree(MT, NT, P, a)
2:     nblist_2 = MT/(P * a)
3:     for k = 0 to NT − 1 do
4:         for r = 0 .. MT − 1 do
5:             Push(r, L_{2,r%P,r/(P*a)})          ▷ Initialize the queues
6:         end for
7:         for p = 0 .. P − 1 do
8:             /* Generate the lower level trees (TS kernels) */
9:             for l = 0 .. nblist_2 − 1 do
10:                i = GeneratePartialTree(Flat, L_{2,p,l})
11:                Push(i, L_{1,p})          ▷ Move the pivot to the upper level
12:            end for
13:            /* Generate the local reduction tree */
14:            i = GeneratePartialTree(Greedy, L_{1,p})
15:            Push(i, L_0)                  ▷ Move the pivot to the upper level
16:        end for
17:        /* Generate the distributed reduction tree */
18:        i = GeneratePartialTree(Flat, L_0)
19:        Register i as the main pivot for step k
20:    end for
21: end function
```

Fig. 7. Pseudo-code of the algorithm used to generate the reduction trees in the `xTPQRT` kernel.

level to handle distributed-memory environment and reduce communications, the middle one on each node to increase the amount of parallelism, and finally, the bottom level with flat trees to exploit kernel efficiency and cache locality.

The general algorithm is described in Fig. 7. It relies on three levels of list which contains the rows that must be reduced at each of these levels. For each of these lists, we compute a tree, flat or greedy, out of which a single tile, the pivot, emerge as not being reduced. Fig. 6 describes this algorithm. In the case of a flat tree, the pivot is chosen as the first available row and all others are reduced as soon as possible in any order. This prevents from having the issue presented in Fig. 4a. For the greedy tree, we recursively take the first available couple and push back the temporary pivot. This way, we ensure the shortest critical path. The emerging pivot is then pushed to the higher level. The three levels of list used in this algorithm are the following:

- the bottom is composed of $nblist_2$ lists L_2 per node. At each iteration, all tiles that must be reduced are dispatched in these lists following the 2D block-cyclic data distribution to collect tiles belonging to a same node together. The number of these lists per node is tuned with the parameter a that defines the maximum size of these lists from kernel efficiency ($a = MT$) to parallelism ($a = 1$).
- the middle is composed of one L_1 list per node. These lists gather all the local pivots that have emerged from the local L_2 lists.
- the top is composed from the single L_0 list. This list gathers the single pivots emerging from each node to compute the tree that will impact the amount of communication.

At each level, it is possible to choose between flat and greedy trees. As shown in Fig. 7, the bottom tree is flat to ensure computations with the `TS` family kernels. The middle one uses greedy to reduce the critical path. Finally, both types of trees can be used for the top level tree. Since C_2 is a square (or upper triangular) matrix, we chose in this work to use a flat tree as in [12] for the top level. This can be explained by the fact that flat tree generates a ring of communications, while a greedy tree generates a binary tree where data are sent back and forth along each edge to keep the initial data partitioning. Thus, the former has a longer critical path (P versus $log(P)$), but a smaller volume of communications (P tiles versus $2P$), which may have a larger impact on large square matrices, as explained in [12].

The PTG implementation of the DPLASMA HQR has fully parameterized dependencies thanks to functions describing the tree connections. Thus, we extended this algorithm to consider our new tree structure as well as the specific structure of the C matrix composed of two tiled-matrices on top of each other. As proposed for the `POSV` algorithm, we merged together the QR factorization and the generation of Q.

B. Two-Norm Matrix Estimator

We propose an implementation on top of PaRSEC of the 2-norm estimator algorithm presented in [41]. This implementation relies on an efficient power method that minimizes the communications. Fig. 8 illustrates the major steps of this algorithm. First, we recall that we use a 2D block-cyclic data distribution as in ScaLAPACK, represented by the set of six colors for a grid $P \times Q$ of 2×3 in this example. In order to efficiently perform the matrix-vector operations (`GEMV`), we implemented a so-called 2.5D algorithm [39]. The vector x (resp. y) is given on input as replicated P times (resp. Q) on each line (resp. column) of processors of the grid distribution of A. This makes it possible to compute Q (resp P) local partial sums of the output vector. The final solution is then computed by a reduction/broadcast of the partial sums. We choose to implement it with a ring of communication, but this can be optimized for larger number of nodes with Bruck's algorithm [42]. The matrix A is then never communicated, and only small portions of the vectors are communicated. As shown in Fig. 8, the norms of the computed vector x (resp. y) can be overlapped with the computation of the next

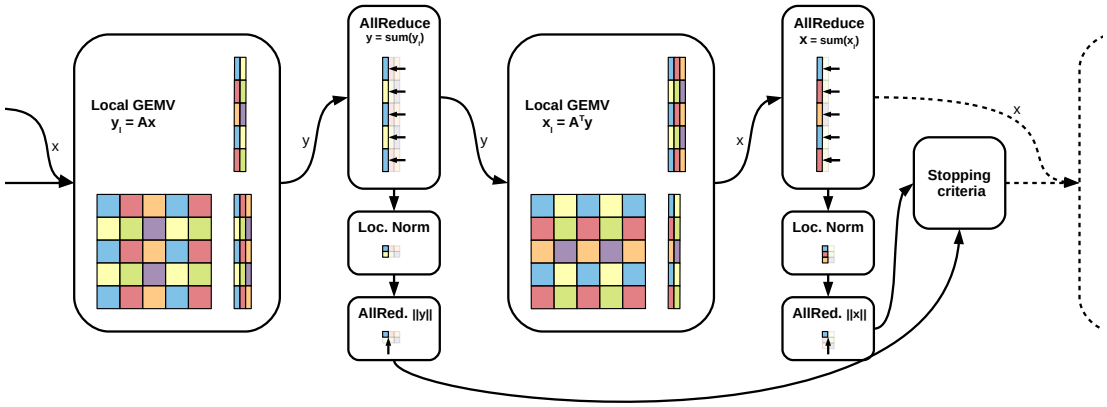


Fig. 8. High level steps of the condition number estimate iteration loop: the two matrix-vector products and the two norm computations. In this example, a 5×5 tile matrix is distributed across a 2×3 grid of processor. The vectors x (resp. y) are replicated 2 (resp. 3) times to reduce communications.

matrix-vector product. This is performed in a similar manner to reduce local information to obtain the final result among the processors. To reduce the number of processors involved in the synchronizations, we replicate the norm computation per column of the vector as the input data is replicated. This costs a little more communication, but may allow for more flexibility in the scheduling. We extend this implementation for all norm computations of the DPLASMA library and reduce as much as possible the volume of communications.

C. SUMMA-Based Matrix-Matrix Multiplication

Due to the large amount of memory already required for the QDWH algorithm, it is difficult to dedicate more memory workspace to the replication of the C matrix in the `GEMM` function. Thus, we chose to modify the `GEMM` algorithm of the DPLASMA library with a SUMMA algorithm. PARSEC provides control flow edges in PTG to implement additional dependencies. These control flows have been used to limit the number of columns of A (resp. rows of B) that can be communicated by the runtime to avoid saturation of the communication network. Next, we modify the algorithm to replace the communication broadcast by pipelined rings of communications, as done in SCALAPACK. The number of columns (rows) sent in the pipeline is automatically defined by a look-ahead parameter. This parameter is computed based on the number of tiles in the matrix to always have a minimum and sufficient number of `GEMM` tasks per available core. The look-ahead parameter is set to three in our experiments after some tuning. This work, similar to the solution proposed in TileDB [43], has improved by 30% the performance of the DPLASMA matrix-matrix multiplication. However, as opposed TileDB and to SCALAPACK, the communication we herein propose are tile based, and each row of tiles (column of tiles) has its own set of control edges, providing more flexibility to the scheduling of the tasks and communications.

D. DAG Composition for Cholesky-Based Solve

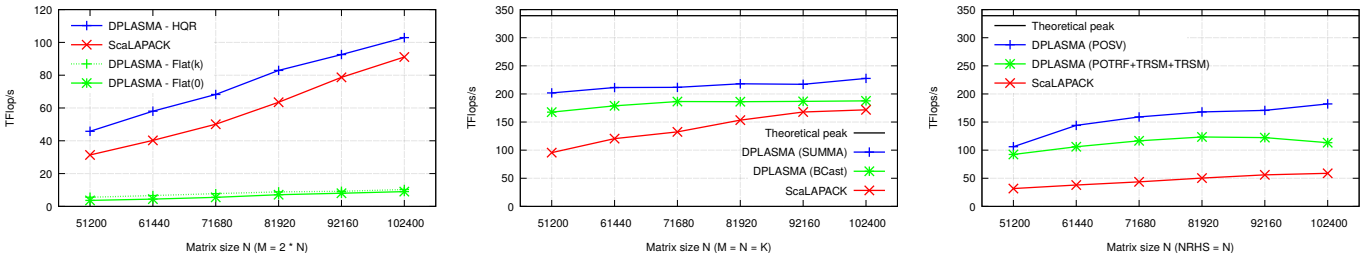
We study the impact of implementing a single PTG for the first two operations `POTRF+TRSM`, versus having two separated DAGs. There are two reasons for not fusing the three algorithms in one. First, there is an unavoidable synchronization point between the two `TRSM`. Second, this would require us to store a local copy of A matrix on each node. The first data communicated during the first stage are the last ones used by the third stage. As the QDWH algorithm requires to apply the Cholesky solve on an $N \times N$ right hand side, the communications in the third stage are overlapped with the increasing amount of computations.

This solution of combining together the PTG of the different stages of the algorithms has also been used in the implementation of the QR iteration of the QDWH algorithm. In that iteration, four PTGs are combined together: the factorization of C_1 with the HQR algorithm presented in [12], the computation of the associated Q_1 following the same tree, the factorization of C_2 with the new tree proposed in this work, and the computation of the associated Q_2.

IX. Numerical Results and Analysis

A. Environment Settings

Our experiments have been conducted on the Cray XC40 system codenamed *Shaheen-2*, installed at the KAUST Supercomputing Laboratory (KSL), with the Cray Aries network interconnect, which implements a Dragonfly network topology. It has 6,174 compute nodes, each with two-sockets Intel Haswell 16 cores running at 2.3GHz and 128GB of DDR3 main memory and we use the Intel compiler v15.0.2.164. The work load managers on *Shaheen-2* is native SLURM. The Haswell nodes on *Shaheen-2* have a theoretical peak performance of approximately 1.18 TFlops/s. All the experiments are performed using IEEE double-precision arithmetic. The reported performance is the average of three runs, since all the elapsed times have minimal variation. The dense synthetic matrices are generated as $A = UDV^\top$, where U, V are orthogonal matrices obtained through the QR factorization of random matrices, D is a diagonal matrix generated based on the condition number $COND$ of the matrix A, where, $D(i) = 1 - (i-1)/(N-1) * (1 - 1/COND)$.



(a) Impact of HQR. (b) Impact of SUMMA. (c) Impact of DAG composition.

Fig. 9. Performance comparison in TFlop/s on 288 nodes.

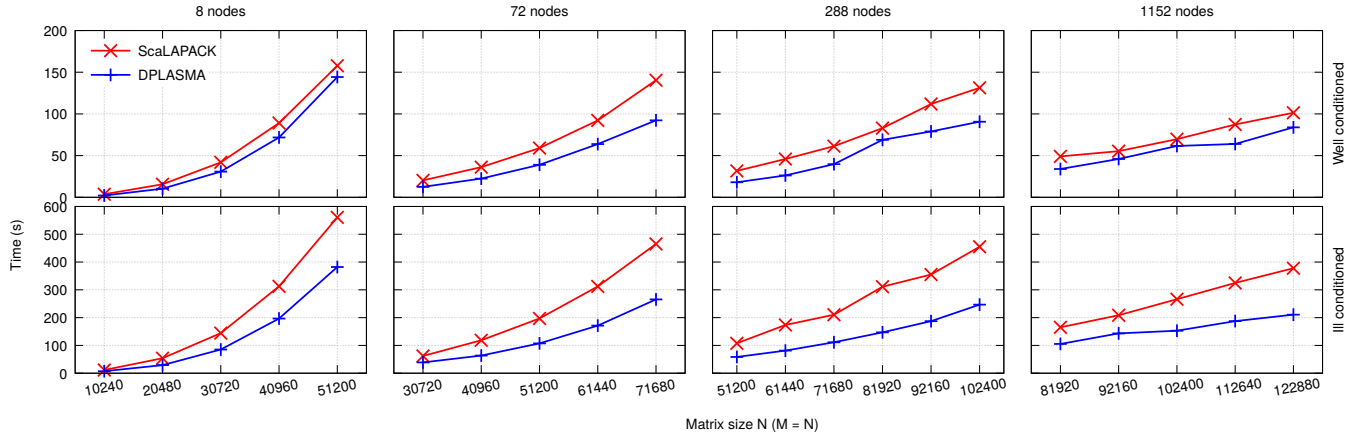


Fig. 10. Assessing DPLASMA task-based QDWH performance in blue against the ScaLAPACK block-algorithm QDWH implementation in red on well-conditioned matrices on top, and ill-conditioned matrices on bottom from 18 nodes on the left to 1,152 nodes on the right.

Regarding the QDWH-based PD decomposition, our implementations switch from Equation 1 to Equation 2 when c_k (line 25 of Fig. 1) becomes smaller than 100.

In the subsequent experiments, the ScaLAPACK reference will be shown in red, and the optimized version proposed by this work in blue. Additional intermediate solutions may be presented in green.

B. Numerical Accuracy

For a given general matrix $A \in \mathbb{R}^{n \times n}$, the polar decomposition $A = U_p H$. The norm $\| \cdot \|_F$ denotes the Frobenius norm. The various accuracy tests are then expressed as follows: $\frac{\|I - U_p U_p\|_F}{\|A\|_F}$, for the orthogonality of the U_p, $\frac{\|A - U_p H\|_F}{\|A\|_F}$,for the accuracy of the overall computed polar decomposition. The orders of the orthogonality and backward errors stand around machine precision, i.e., $1e-15$, for well and ill-conditioned matrices. This demonstrates the numerical robustness of the new DPLASMA formulation of QDWH.

C. Performance impact of the algorithmic optimizations

a) Hierarchical QR factorization for QDWH: We here analyze the impact of the optimized reduction trees on one QR-iteration of the QDWH algorithm. We consider the full operation to compute Q, which means that we consider the factorization of both parts of C and the generation of both parts of Q. We report in Fig. 9a the number of TFlop/s with a fixed number of flops corresponding to the QR factorization of a $2N \times N$ matrix and the generation of the associated Q. As reported in [44], this gives: $20/3N^3 + o(N^3)$. The optimized tree dedicated to the factorization of the specific matrix appearing in the QDWH algorithm outperforms the reference ScaLAPACK implementation by more than 10% continuously on all sizes. The performances of both $Flat()$ implementations highlights the needs for optimized trees reducing the communications to get high performance. $Flat(k)$ performs a little better than $Flat(0)$ thanks to the round-robin optimization, but both are overwhelmed by the large amount of communications and the lack of local parallelism.

b) SUMMA-based matrix-matrix multiplication: Fig. 9b shows the performance of the GEMM operation on 288 nodes for the three implementations we considered: ScaLAPACK, the original DPLASMA implementation performing broadcast of all data as soon as possible and ordered by priority (rank of column for A, rank of the row for B respectively), and the SUMMA implementation performing rings of communications with backward dependencies that prevent communications to be released too early. We observe that both DPLASMA implementations outperforms ScaLAPACK. When the size increases as well as communication volume, the original version slows down and is not able to

achieve the same level of efficiency as the SUMMA implementation. Note that the later reaches 82% of the theoretical peak of the architecture, is 32% faster than SCALAPACK, and 21% faster than the existing DPLASMA implementation.

c) Impact of DAG Composition: We now study the impact of composing the DAG by hand whenever possible to reduce the communication amount and potentially, improve the scalability of the application. Fig. 9c compares the performance achieved on the `POSV` operation with matrices A and B of same sizes with SCALAPACK, the original DPLASMA implementation based on three independent DAGs corresponding to each stage of `POSV`, and the optimized DPLASMA version where the first two stages are merged by hand, as explained in VII-C. The two DPLASMA `POSV`, decomposed and merged, achieves gains up to 1.9x/3.1x against SCALAPACK `POSV`, respectively. The 2-DAGs formulation of the DPLASMA `POSV`, engenders up to 38% performance improvements compared to the reference 3-DAG approach. This basically highlights the communication-reducing optimization achieved by interleaving the factorization step and the first triangular solve in a single JDF file.

D. DPLASMA *QDWH Performance*

a) Overall Performance: The performance analysis of SCALAPACK QDWH has been presented in [25], where it achieved up to 2x/4x against the the SCALAPACK SVD-based polar decomposition and the Elemental [23] implementation of QDWH, respectively. Fig. 10 compares the performance of the new DPLASMA task-based implementation of QDWH on top of the PARSEC runtime system against the former SCALAPACK QDWH, on both well and ill-conditioned matrices, using various numbers of computational nodes and matrix sizes. The DPLASMA QDWH outperforms the SCALAPACK QDWH by up to 1.56x/2.1x speedups for well and ill-conditioned matrices, respectively.

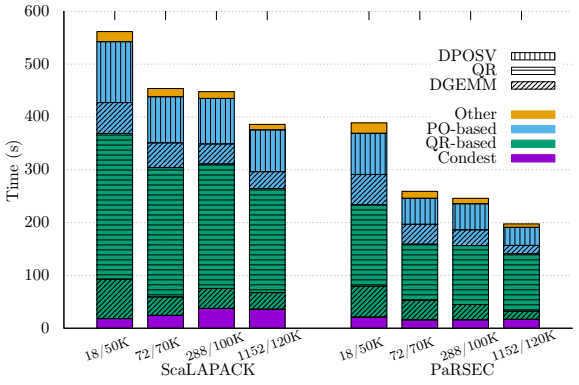


Fig. 11. Profiling the computational stages of the QDWH implementations, SCALAPACK (left) and DPLASMA (right). The profiling results are reported on the various number of nodes for the largest matrix size (#nodes/N).

b) Profiling and Time Breakdown: Fig. 11 shows the time breakdown for SCALAPACK and DPLASMA QDWH across various numbers of MPI processes on the largest matrix size on each grid configuration. The profile is for ill-conditioned matrix, where the number of iterations required to converge is a maximum of six and consists of three QR and three Cholesky-based successive iterations. In particular, the QR-based iterations are more time consuming than Cholesky-based iterations for the two implementations. The DPLASMA implementation of QDWH improves all the computational stages of QDWH compared with SCALAPACK implementation. For instance, the DPLASMA QR/Cholesky-based iterations outperform its corresponding computations in SCALAPACK implementation by up to 2x/2.4x, respectively. The DPLASMA QDWH obtains a better scalability than SCALAPACK QDWH up to 1,152 nodes. There are however some room for further performance improvements, e.g., the process placements, which is an important tuning parameter to mitigate the data movement overheads [25].

X. CONCLUSION AND FUTURE WORK

We have demonstrated how to leverage the QDWH-based Polar Decomposition (PD) using the dynamic runtime system PARSEC on massively parallel systems. Major algorithmic adaptations along with a paradigm shift from the traditional bulk synchronous to asynchronous task-based programming model are the two game-changing ingredients to deploy QDWH-based PD at an unprecedented scale. In particular, the design and implementation of a new hierarchical QR variant customized for QDWH is necessary to reduce data traffic across remote computational nodes. Additional algorithmic optimizations are also shown to be critical, i.e., the new implementation of the two-norm matrix computation, the DAG composition to increase data locality, and the integration of the SUMMA algorithm. Based on the task-based programming model associated with the asynchronous task execution style, the resulting task-based high performance implementation of the QDWH-based PD using PARSEC outperforms by up to 2X speedup the state-of-the-art SCALAPACK-based POLAR implementation using 36,000 cores. For future work, we would like to further extend this distributed-memory implementation using GPU hardware accelerators, possibly with mixed-precision techniques [20]. Another extension of this work is to plugin this newly developed task-based implementation of QDWH-based PD into the ZOLO-PD algorithm [45], which will bring a performance advantage in strong scaling mode of operation. Last but not least, we would also like to integrate QDWH-based PD toward solving the symmetric eigenvalue problem and the SVD [27].

ACKNOWLEDGMENT

The authors would like also to thank Cray Inc. and Intel in the context of the Cray Center of Excellence and Intel Parallel Computing Center awarded to the Extreme Computing Research Center at KAUST. For computer time, this research used *Shaheen-2* supercomputer hosted at the Supercomputing Laboratory at KAUST, a remote system hosted by our Cray partners, and by the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Universit de Bordeaux, Bordeaux INP and Conseil Rgional d'Aquitaine.

REFERENCES

[1] N. J. Higham, "Computing the Polar Decomposition with Applications," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 4, pp. 1160–1174, 1986. [Online]. Available: http://dx.doi.org/10.1137/0907079

[2] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., ser. John Hopkins Studies in the Mathematical Sciences. Baltimore, Maryland: Johns Hopkins University Press, 1996.

[3] I. Bar-Itzhack, "Iterative optimal orthogonalization of the strapdown matrix," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. AES-11, no. 1, pp. 30–37, Jan 1975.

[4] J. A. Goldstein and M. Levy, "Linear algebra and quantum chemistry," *Am. Math. Monthly*, vol. 98, no. 10, pp. 710–718, Oct. 1991. [Online]. Available: http://dx.doi.org/10.2307/2324422

[5] Y. Nakatsukasa, Z. Bai, and F. Gygi, "Optimizing Halley's Iteration for Computing the Matrix Polar Decomposition," *SIAM Journal on Matrix Analysis and Applications*, pp. 2700–2720, 2010.

[6] Y. Nakatsukasa and N. J. Higham, "Stable and Efficient Spectral Divide and Conquer Algorithms for the Symmetric Eigenvalue Decomposition and the SVD," *SIAM Journal on Scientific Computing*, vol. 35, no. 3, pp. A1325–A1349, 2013. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/120876605

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

[8] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An abstraction for unhindered parallelism," *Proceedings of WOLFHPC 2014: 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Stor*, pp. 21–30, 2014.

[9] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA projects," in *Journal of Physics: Conference Series*, vol. 180, 2009.

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

[11] "The Chameleon Project," http://project.inria.fr/, INRIA Bordeaux, 2018.

[12] J. Dongarra, M. Faverge, T. Hrault, M. Jacquelin, J. Langou, and Y. Robert, "Hierarchical qr factorization algorithms for multicore clusters," *Parallel Computing*, vol. 39, no. 4, pp. 212 – 232, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819113000100

[13] R. A. Van De Geijn and J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291096-9128%28199704%299%3A4%3C255%3A%3AAID-CPE250%3E3.0.CO%3B2-2

[14] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Hérault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA," in *IPDPS Workshops*. IEEE, 2011, pp. 1432–1441. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6008655

[15] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Investigating Applications Portability with the Uintah DAG-based Runtime System on PetaScale Supercomputers," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013.

[16] "The HiCMA Library," http://github.com/ecrc/hicma, Extreme Computing Research Center, King Abdullah University of Science and Technology, 2018.

[17] K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes, "Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures," in *High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings*. Cham: Springer International Publishing, 2017, pp. 22–40.

[18] K. Akbudak, H. Ltaief, A. Mikhalev, A. Charara, A. Esposito, and D. Keyes, "Exploiting data sparsity for large-scale matrix computations," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 721–734.

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

[20] D. Sukkari, H. Ltaief, and D. E. Keyes, "A High Performance QDWH-SVD Solver Using Hardware Accelerators," *ACM Trans. Math. Softw*, vol. 43, no. 1, pp. 6:1–6:25, 2016. [Online]. Available: http://doi.acm.org/10.1145/2894747

[21] MAGMA, "Matrix Algebra on GPU and Multicore Architectures. Innovative Computing Laboratory, University of Tennessee. Available at http://icl.cs.utk.edu/magma/," 2009.

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

[23] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A New Framework for Distributed Memory Dense Matrix Computations," *ACM Trans. Math. Softw*, vol. 39, no. 2, p. 13, 2013. [Online]. Available: http://doi.acm.org/10.1145/2427023.2427030

[24] D. Sukkari, H. Ltaief, and D. Keyes, "High Performance Polar Decomposition on Distributed Memory Systems," in *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, ser. Lecture Notes in Computer Science, P.-F. Dutot and D. Trystram, Eds., vol. 9833. Springer, 2016, pp. 605–616. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-43659-3

[25] D. Sukkari, H. Ltaief, A. Esposito, and D. Keyes, "A QDWH-based SVD software framework on distributed-memory manycore systems," *ACM Trans. Math. Softw.*, vol. 45, no. 2, pp. 18:1–18:21, Apr. 2019. [Online]. Available: http://doi.acm.org/10.1145/3309548

[26] Cray, "Libsci," http://docs.cray.com.

[27] D. Sukkari, H. Ltaief, M. Faverge, and D. Keyes, "Asynchronous task-based polar decomposition on single node manycore architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 312–323, Feb 2018.

[28] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[29] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemariner, and J. Dongarra, "DAGuE: A generic distributed DAG Engine for High Performance Computing." *Parallel Computing*, vol. 38, no. 1-2, pp. 27–51, 00-2012 2012.

[30] M. Cosnard, E. Jeannot, and T. Yang, "Compact dag representation and its symbolic scheduling," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, pp. 921–935, 2004.

[31] A. Buttari, J. Langou, and J. K. ad Jack J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. Volume 35, pp. pp. 38–53.

[32] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurrency: Practice and Experience*, vol. 20, no. 13, pp. 1573–1590, 2008.

[33] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. V. Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Transactions on Mathematical Software*, vol. 36, no. 3, 2009.

[34] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-avoiding parallel and sequential QR and LU factorizations: theory and practice," LAPACK Working Note, Tech. Rep. 204, 2008. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn204.pdf

[35] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra, "Tile QR factorization with parallel panel processing for multicore architectures," in *IPDPS'10, the 24st IEEE Int. Parallel and Distributed Processing Symposium*, 2010.

[36] H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert, "Tiled QR factorization algorithms," in *SC'2011, the IEEE/ACM Conference on High Performance Computing Networking, Storage and Analysis*. ACM Press, 2011.

[37] F. Song, H. Ltaief, B. Hadri, and J. Dongarra, "Scalable tile communication-avoiding QR factorization on multicore cluster systems," in *SC'10, the 2010 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2010.

[38] M. Faverge, J. Langou, Y. Robert, and J. Dongarra, "Bidiagonalization and R-Bidiagonalization: Parallel Tiled Algorithms, Critical Paths and Distributed-Memory Implementation," in *IPDPS'17 - 31st IEEE International Parallel and Distributed Processing Symposium* , Orlando, United States, May 2017. [Online]. Available: https://hal.inria.fr/hal-01484113

[39] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms," in *Euro-Par 2011 Parallel Processing*, E. Jeannot, R. Namyst, and J. Roman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 90–109.

[40] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 261–272.

[41] N. J. Higham, "Estimating the matrix p-norm," *Numerische Mathematik*, vol. 62, no. 1, pp. 539–555, Dec. 1992. [Online]. Available: https://doi.org/10.1007/BF01396242

[42] J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1143–1156, Nov 1997.

[43] J. A. Calvin, C. A. Lewis, and E. F. Valeev, "Scalable task-based algorithm for multiplication of block-rank-sparse matrices," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:8. [Online]. Available: http://doi.acm.org/10.1145/2833179.2833186

[44] S. Blackford and J. J. Dongarra, "Installation guide for LAPACK," LAPACK Working Note, Tech. Rep. 41, Jun. 1999, originally released March 1992. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn41.pdf

[45] H. Ltaief, D. Sukkari, A. Esposito, Y. Nakatsukasa, and D. Keyes, "Massively Parallel Polar Decomposition on Distributed-Memory Systems," *Accepted at ACM Transactions on Parallel Computing, available at http://hdl.handle.net/10754/626359*, 2019.