This paper presents a high performance software framework for computing a dense SVD on distributed-memory manycore systems. Originally introduced by Nakatsukasa et al. (Nakatsukasa et al. 2010; Nakatsukasa and Higham 2013), the SVD solver relies on the polar decomposition using the QR Dynamically-Weighted Halley algorithm (QDWH). Although the QDWH-based SVD algorithm performs a significant amount of extra floating-point operations compared to the traditional SVD with the one-stage bidiagonal reduction, the inherent high level of concurrency associated with Level 3 BLAS compute-bound kernels ultimately compensates for the arithmetic complexity overhead. Using the ScaLAPACK two-dimensional block cyclic data distribution with a rectangular processor topology, the resulting QDWH-SVD further reduces excessive communications during the panel factorization, while increasing the degree of parallelism during the update of the trailing submatrix, as opposed to relying to the default square processor grid. After detailing the algorithmic complexity and the memory footprint of the algorithm, we conduct a thorough performance analysis and study the impact of the grid topology on the performance by looking at the communication and computation profiling trade-offs. We report performance results against state-of-the-art existing QDWH software implementations (e.g., Elemental) and their SVD extensions on large-scale distributed-memory manycore systems based on commodity Intel x86 Haswell processors and Knights Landing (KNL) architecture. On both platforms, the QDWH-SVD framework achieves up to 3/8-fold on the Haswell/KNL-based platforms, respectively, against ScaLAPACK PDGESVD and turns out to be a competitive alternative for well and ill-conditioned matrices. We finally come up herein with a performance model based on these empirical results. Our QDWH-based polar decomposition and its SVD extension are freely available at https://github.com/ecrc/qdwh.git and https://github.com/ecrc/ksvd.git, respectively, and have been integrated into the Cray Scientific numerical library LibSci v17.11.1.

CCS Concepts:
- Mathematics of computing → Solvers; Mathematical software performance;

Additional Key Words and Phrases: Dense SVD solver, Polar decomposition, QDWH, Performance analysis, Distributed-memory manycore systems

ACM Reference format:
DOI: 0000001.0000001

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 1559-1131/2017/3-ART4 $15.00
DOI: 0000001.0000001
1 INTRODUCTION
Reducing data movement is probably one of the most challenging tasks faced by application developers, due to the increasing hardware complexity and the lack of expressivity for data locality in the current programming models (Unat et al. 2017). In particular, the ScaLAPACK (Blackford et al. 1997) numerical library relies on the two-dimensional block cyclic data distribution to ensure load balancing as well as to reduce horizontal communication across the network interconnect. While most of dense linear algebra algorithms on distributed-memory systems, including the High Performance Linpack benchmark (Petitet et al. 2008), take advantage of this communication-reducing data layout, the processor grid topology remains an important factor to tune accordingly, depending on the considered workload type.

This paper revisits the polar decomposition based on the QR-based dynamically weighted Halley (QDWH) iteration, initially introduced by Nakatsukasa et al. (Nakatsukasa et al. 2010; Nakatsukasa and Higham 2013) and its extension to SVD solver. The polar decomposition is a key algorithm for various scientific applications, e.g., in continuum mechanics to decompose stress tensors and to simulate the deformation of an object, in aerospace computations (Meyer and Bar-Itzhack 1977) during strapdown inertial navigation and other aerospace systems to describe the rotation of one coordinate system relative to a reference coordinate system, and in chemistry (Goldstein and Levy 1991) to help the understanding of properties in terms of electron pair (chemical bond) transferability, etc. Further applications are also reported by Higham in (Higham 1986). The QDWH algorithm is backward stable and converges within six iterations in double precision, depending on the matrix condition number. The algorithm is composed of a set of calls, at each iteration, to highly parallel compute-bound matrix operations (QR, Cholesky, matrix-matrix multiplication, etc.). Once computed, the polar decomposition may be used toward the computation of the SVD, which is the core operation of many computational science and engineering applications, such as principal component analysis for machine learning (Wang and Zhu 2017).

In this paper, we extend our initial work (Sukkari et al. 2016b), which describes the first implementation of the QDWH-based polar decomposition and its application to SVD on distributed-memory systems using ScaLAPACK. One of the main objectives of the paper is to investigate the impact of the processor grid topology (square versus rectangular) on the overall performance by looking at computation and communication profiling phases. The square grid used in (Sukkari et al. 2016b) may not be adequate for dense linear algebra algorithms involved in the QDWH-SVD or even in the standard bidiagonal SVD algorithm (i.e., the PDGESVD routine from ScaLAPACK), due to excessive data movement in the panel factorization and limited parallelism in the update of the trailing submatrix. Therefore, the grid topology needs to be reconciled with the dense matrix computations, while keeping the same two-dimensional block cyclic data distribution inherited from ScaLAPACK. Performance reported using only rectangular grid topology on the standalone QDWH inner iterations shows up to 45% and 40% improvement compared to square grid topology on large-scale distributed-memory systems (Sukkari et al. 2016b), composed of commodity Intel x86 Haswell processors and Knights Landing (KNL) manycore architecture, respectively. To further assess the new QDWH performance with a rectangular grid topology, extensive performance comparisons have been also conducted against a high performance implementation of the QDWH-based polar decomposition from Elemental (Poulson et al. 2013). The new QDWH implementation achieves a fourfold speedup compared to Elemental. When looking at the SVD extension, the QDWH-based SVD is able to achieve up to 8-fold speedup against its equivalent PDGESVD routine from ScaLAPACK. A performance model is herein derived from these empirical results. The QDWH software library and its SVD extension are freely available under modified-BSD license at https://github.com/ecrc/qdwh.git and https://github.com/ecrc/ksvd.git, respectively, for
the scientific community and have been integrated into the Cray Scientific numerical library LibSci v17.11.1.

The remainder of the paper is organized as follows. Section 2 provides a literature review for the polar decomposition with its SVD extension and highlights the contributions of this paper compared to previous works. Section 3 briefly recalls the QDWH-based polar decomposition algorithm and its possible integration into an SVD solver, while performing a detailed algorithmic complexity and memory footprint analyses. Section 4 describes the QDWH-SVD software framework based on ScaLAPACK and outlines its application programming interface. Section 5 describes the thorough performance campaign conducted on two large-scale manycore systems and reports on the performance impact of the grid topology (square versus rectangular). Section 6 investigates communication / computation profiles and derives a model for performance scalability from these empirical results. We conclude in Section 7.

2 RELATED WORK

The polar decomposition has been extensively studied for more than two decades, thanks to its high numerical accuracy and backward stability properties, as well as its high degree of parallelism. However, it has the disadvantage of requiring lots of floating-point operations, which explains why its high performance implementation has become popular only in the last five years with the emergence of manycore architecture and hardware accelerators.

Higham and Papadimitriou (Higham and Papadimitriou 1994) proposed to use the polar decomposition in the context of the singular value decomposition on a shared-memory system. Their implementation relied on the yet expensive matrix inversion, while maintaining the numerical accuracy robustness. Later, Nakatsukasa et al. (Nakatsukasa et al. 2010) revisited the polar decomposition by replacing the overkill matrix inversion kernels with QR factorization kernels and theoretically proved the numerical robustness of the new inverse-free polar decomposition based on QDWH. The latter work was then extended by Nakatsukasa and Higham (Nakatsukasa and Higham 2013) toward developing a new spectral divide and conquer algorithms for the symmetric eigenvalue problem and the SVD that are backward stable. Moreover, experiments on numerical accuracy and orthogonality of the polar factor have been conducted with MATLAB to further assess the robustness of QDWH in presence of well and ill-conditioned matrices.

High performance QDWH implementations have been considered only recently: Sukkari et al. (Sukkari et al. 2016a) demonstrated an efficient implementation on a shared-memory system equipped with multiple GPUs and have extended it then to distributed-memory systems composed of homogeneous x86 nodes in (Sukkari et al. 2016b). It is noteworthy to mention another implementation of QDWH, part of the numerical library Elemental (Poulson et al. 2013). Elemental is a framework for high performance dense and sparse matrix computations, which operates on most of today’s supercomputers. To our knowledge, the aforementioned QDWH implementations are the only ones available on distributed-memory systems.

This paper is an extension of a previous work (Sukkari et al. 2016b). The main contributions here are fourfold. After describing the algorithmic complexity and memory footprint, we present performance analysis and study the grid topology and its impact on the overall QDWH performance with its SVD extension. We conduct a benchmarking campaign on two large-scale distributed-memory systems, including a system containing hundreds of the latest Intel Knights Landing (KNL) manycore architecture, from which we derive a performance model. We compare the performance against QDWH as implemented in Elemental (Poulson et al. 2013) and propose an extension to SVD solver using two variants. Finally, we release a ScaLAPACK-based software library at https://github.com/ecrc/qdwh.git, which packages the overall QDWH-SVD solver.
3 BACKGROUND

This section provides background information on the QDWH-based polar decomposition and its potential for integration into an SVD solver, as a pre-processing step. It also highlights a detailed algorithmic complexity and memory footprint analyses.

3.1 The QDWH-based Polar Decomposition

The polar decomposition 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 $H = \sqrt{A^T A}$ is a symmetric positive semidefinite matrix. We use the inverse-free QDWH-based iterative procedure to calculate the polar decomposition (Nakatsukasa and Higham 2013) as follows:

$$X_0 = A / \alpha,$$

$$\begin{bmatrix} \sqrt{c_k} X_k \\ I \end{bmatrix} R, \ X_{k+1} = b_k c_k X_k + \frac{1}{\sqrt{c_k}} \left( a_k - b_k \right) Q_1 Q_2^T, \ k \geq 0. \quad (1)$$

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

$$X_{k+1} = b_k c_k X_k + \left( a_k - b_k \right) (X_k W_k^{-1}) W_k^{-\tau},$$

$$W_k = \text{chol}(Z_k), \ Z_k = I + c_k X_k^T X_k. \quad (2)$$

This algorithmic switch at runtime permits to exploit the numerical properties of the matrix, which creates opportunities to reduce the overall algorithmic complexity. The QDWH-based polar decomposition converges then faster, while still maintaining numerical stability.

In the next two subsections, we assume the matrix is square of size $n \times n$, without loss of generality.

3.2 Algorithmic Complexity

The QDWH-Based Polar Decomposition. The number of floating-point operations of the polar decomposition depends on the total number of iterations required for QDWH to converge, which is dictated by the condition number $\lambda_0$ of the original matrix problem. However, the maximum number of iterations is six for double precision arithmetic in case the matrix is extremely ill-conditioned. We refer the reader to (Nakatsukasa and Higham 2013) for more details on the theoretical proof. QDWH typically performs an initial condition estimate of the matrix using LU factorization, and then enters the QDWH loop to perform QR or Cholesky-based iterations depending on $\lambda_0$. In case the matrix is ill-conditioned, QDWH may converge after additional QR or Cholesky-based iterations (but up to a total of six), besides executing in-between other expensive Level 3 BLAS matrix operations. As explained in (Sukkari et al. 2017), the overall cost of QDWH is $(\frac{4}{3}) n^3 + (8 + \frac{4}{3}) n^3 \times \#it_{QR} + (4 + \frac{1}{3}) n^3 \times \#it_{Chol} + 2n^2$, where $\#it_{QR}$ and $\#it_{Chol}$ correspond to the number of QR-based and Cholesky-based iterations, respectively. The total flop count of QDWH for dense matrices ranges then from $(11 + \frac{1}{3}) n^3$ (for $\lambda_0 \approx 1$ with $\#it_{Chol} = 2$) to $(41 + \frac{2}{3}) n^3$ (for $\lambda_0 \gg 1$, with typically $\#it_{QR} = 3$ and $\#it_{Chol} = 3$).

Although the resulting QDWH algorithmic complexity may sound daunting at first glance, most of its building blocks are conventional dense linear algebra kernels, highly parallel and tuned by most vendor numerical libraries on distributed-memory systems (e.g., Intel MKL ScaLAPACK, IBM PESSL, Cray LibSci, etc.). Furthermore, the latest commodity hardware improvements in terms of instruction and vector units (e.g., AVX2) on homogeneous x86 architectures may help rehabilitating expensive numerical linear algebra algorithms, such as QDWH.
The QDWH-Variant for SVD. As initially introduced in (Higham and Papadimitriou 1994), the polar decomposition can be used as a first computational stage toward calculating the SVD of a general dense matrix. The new SVD algorithm can be decomposed in three steps. The first phase performs the QDWH-based polar decomposition $A = U_p \times H$. The second step calculates the singular values and the right singular vectors $H = V \times \Sigma \times V^T$ using a symmetric dense eigensolver of choice, e.g., based on the multiple relatively robust representations (MRRR), the divide-and-conquer algorithm or the combination of two-stage tridiagonal reduction, with a divide-and-conquer eigensolver, as implemented in $PDSYEVR$ / $PDSYEVD$ from ScALAPACK or from ELPA (Marek et al. 2014), respectively. The last phase computes the left singular vectors $U \times V$. These three successive computational stages represent the crux of the QDWH-SVD on distributed-memory systems. The choices of the symmetric dense eigensolver in the second phase requires $3 + \frac{1}{3}n^3$ flops for $PDSYEVR$, or $4 + \frac{2}{3}n^3$ flops for $PDSYEVD$ $6 + \frac{2}{3}n^3$ for ELPA-EIG. Finally, computing the left singular vectors adds $2n^2$.

The standard bidiagonalization method for calculating the SVD of a dense matrix, as implemented in $PDGESVD$, relies on the QR solver and requires $8\frac{3}{2}n^3$ for reduction, $12n^3$ for solving the bidiagonal problem, $\frac{8}{3}n^3$ for back-transformation to compute the eigenvectors. Therefore, the total flop count for $PDGESVD$ is $(17 + \frac{1}{2})n^3$.

<table>
<thead>
<tr>
<th>SVD</th>
<th>well</th>
<th>ill</th>
</tr>
</thead>
<tbody>
<tr>
<td>QDWH + PDSYEVR</td>
<td>$(16 + \frac{2}{3})n^3 \leq \cdots \leq 47n^3$</td>
<td></td>
</tr>
<tr>
<td>QDWH + PDSYEVD</td>
<td>$18n^3 \leq \cdots \leq (48 + \frac{1}{3})n^3$</td>
<td></td>
</tr>
<tr>
<td>QDWH + ELPA-EIG</td>
<td>$20n^3 \leq \cdots \leq (50 + \frac{1}{3})n^3$</td>
<td></td>
</tr>
<tr>
<td>PDGESVD</td>
<td>$(17 + \frac{1}{3})n^3$</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 summarizes the algorithmic complexity of the four SVD variants. All variants are competitive with each other for well-conditioned matrices, while a factor of three exists between $PDGESVD$ and the others, when the matrix is ill-conditioned.

3.3 Memory Footprint
In this section, we present the memory allocations required by both solvers based on QDWH and the $PDGESVD$ routine from ScALAPACK to perform an SVD of a dense matrix of size $n \times n$. We recall the general QDWH algorithm in Algorithm 1 from (Sukkari et al. 2016b). We ignore the $O(n)$ allocations. The dominating factor in the overall memory allocation of QDWH-SVD is the memory allocation required by QDWH. Each QDWH iteration calculates the QR decomposition of a matrix size $2n \times n$ and updates the results in the polar factor data structure of size $n \times n$. A copy of the original matrix is also needed to compute the semi-definite Hermitian polar factor $H = U_p^* \times A$, each of size $n \times n$. Therefore, QDWH requires $6n \times n$ memory allocation. Once the polar decomposition is calculated, the symmetric eigensolver requires $2n \times n$ to store the eigenvectors.
and the necessary workspaces. It can reuse for that purpose the already allocated buffers from QDWH at no extra memory cost. Therefore, the overall memory allocation of QDWH-SVD is $6n \times n$, while as a comparison, the PDGESVD routine from ScaLAPACK needs $3n \times n$ to store the original matrix as well as the left and right singular vectors.

4 THE QDWH-SVD SOFTWARE FRAMEWORK

This section describes the QDWH software library (https://github.com/ecrc/qdwh.git) and its SVD extension (https://github.com/ecrc/ksvd.git), their programming model, software dependencies and application programming interfaces.

4.1 Programming Model and Software Dependencies

The QDWH-SVD software framework relies on ScaLAPACK (Blackford et al. 1997) to compute the various dense linear algebra operations, such as Level 3 BLAS and factorizations. ScaLAPACK relies on block algorithms, similar to LAPACK (Anderson et al. 1999), but operates on distributed-memory systems. The block algorithms can be expressed by two successive computational stages: the panel factorization and the update of the trailing submatrix. While the former is memory-bound, the latter is rich in compute-bound operations. And this is where most of ScaLAPACK (and LAPACK) dense linear algebra algorithms extract parallel performance by means of calls to Level 3 BLAS, as implemented in the Parallel BLAS (PBLAS) layer. PBLAS calls translate into on-node BLAS calls, usually supported by high performance implementations from vendors, followed by parallel reductions.

Moreover, ScaLAPACK uses two-dimensional (2D) block cyclic data distribution associated with a 2D processor grid topology to map the matrix data to the distributed-memory nodes, while ensuring load balancing. At the higher level of the library, ScaLAPACK employs the bulk synchronous programming model and uses the Message Passing Interface (MPI) (MPI Forum 1993) to move data between remote processing units. In particular, the Basic Linear Algebra Communication Subprograms (BLACS) layer is in charge of setting up the MPI send/receive communication functions during the matrix computations.

Furthermore, the processor grid topology needs to be appropriately dimensioned. The panel factorization is typically sequential and may not benefit from having many processors participating in this computational phase. However, one should properly calibrate the number of processors to carry on, in parallel, the update of the trailing submatrix.

The performance impact of the processor grid topology is, therefore, critical and is one of the main subjects addressed in the performance results section. This aspect is particularly challenging for QDWH, since it corresponds to a collection of successive calls to several ScaLAPACK routine, see Algorithm 1 in (Sukkari et al. 2016b), e.g., PDGETRF / PDGECON for the matrix condition estimate, PDGEQRF / PDORMQR for the QR-based QDWH iterations and PDPOSV / PDGEMM for the Cholesky-based QDWH iterations.

4.2 Application Programming Interfaces

This section describes the two following Application Programming Interfaces (APIs) of the QDWH-based polar decomposition (PDGEQDWH) and its corresponding SVD solver (PDGEQSVD).

Both APIs try to be similar to current ScaLAPACK APIs by respecting the structure of the function header as well as the data structure encapsulated within the descriptor. This facilitates the integration process of QDWH routines into existing ScaLAPACK user applications.
The PDGEQDWH API permits to compute the QDWH-based polar decomposition $A = U_p H$ of a general matrix $A$ of size $(m,n)$ and returns the orthogonal polar factor $U_p$ within $A$ and the symmetric positive semidefinite $H$, as follows:

```c
int pdgeqdwh ( char *jobh, int m, int n,
        double *A, int iA, int jA, int *descA,
        double *H, int iH, int jH, int *descH,
        double *Work1, int lWork1,
        double *Work2, int lWork2,
        int *info ).
```

The parameter `jobh` determines whether or not the symmetric positive semidefinite polar factor needs to be computed. The ScaLAPACK descriptors `descA` and `descH` contain information on how the final orthogonal and the symmetric positive semidefinite polar factors, respectively, are distributed across the two-dimensional block cyclic grid topology. If `jobh` is different to ‘H’, then $H$ is not referenced. $Work1$ and $Work2$ are the required local workspaces of dimensions $lWork1$ and $lWork2$, respectively, needed during the polar iteration on each processor. If $lWork1$ and/or $lWork2$ are set to $-1$, the size of the workspaces are returned to the user. If the i-th argument is an array and the j-entry had an illegal value, then the parameter `info` returns $-(i * 100 + j)$. If the i-th argument is a scalar and had an illegal value, then `info` is set to $-i$.

The second API calculates the SVD of a general matrix $A$ of size $(m,n)$ based on the QDWH-based polar decomposition $A = U \Sigma V^T$ and returns the orthogonal left/right singular vectors within $U / V$, respectively, and the singular values $\Sigma$ within the vector $S$, as follows:

```c
int pdgeqsvd ( char *jobu, char *jobvt, char *eigtype,
        int m, int n,
        double *A, int iA, int jA, int *descA,
        double *S,
        double *U, int iU, int jU, int *descU,
        double *VT, int iVT, int jVT, int *descVT,
        double *Work, int lWork,
        int *iWork, int liWork, int *info ).
```

The parameters `jobu` and `jobvt` are the parameters to compute the left and right singular vectors, respectively. The parameter `eigtype` defines the symmetric eigensolver to be used during the second phase of the QDWH-based SVD algorithm: `eigtype` can be equal to ‘R’, ‘D’ or ‘E’ for MRRR (PDSYEVR), divide-and-conquer (PDSYEVD) or two-stage ELPA symmetric eigensolver. The ScaLAPACK descriptors `descA`, `descU` and `descVT` contain information on how the original matrix, the left singular vector, and the right singular vector, respectively, are distributed across the two-dimensional block cyclic grid topology. If `jobu` and/or `jobvt` are set ‘N’, then $U$ and $VT$ are not referenced. The parameter $S$ contains the singular values. The double precision and integer workspaces $Work$ and $iWork$ of size $lWork$ and $liWork$, respectively, are used by the internal symmetric eigensolver. Similarly for the first API, if the i-th argument is an array and the j-entry had an illegal value, then the parameter `info` returns $-(i * 100 + j)$. If the i-th argument is a scalar and had an illegal value, then `info` is set to $-i$.

5 EXPERIMENTAL RESULTS
This section reports on the performance benchmarking campaign on two large-scale supercomputers and studies the impact of the grid topology.

5.1 Environment Settings

Our experiments have been conducted on two Cray systems. The first one is a 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 6174 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 second system is another Cray XC with Aries network interconnect but now featuring compute nodes with Intel Xeon Phi Knights Landing (KNL) processors. Every KNL has a base frequency of 1.4GHz and is equipped with 192GB DDR4-2400 RAM and 16GB MCDRAM. The KNL system is operated in quadrant mode while the memory is in cache mode, where all MCDRAM is configured as direct-mapped cache. We use 64 cores out of 68 on every KNL for computation, while 4 cores were dedicated to system services. Furthermore, "hugepages" are employed to improve memory accesses. Executables for the KNL target have been generated with the Intel compiler v17.0.1.132. The work load managers on Shaheen-2 and the KNL system are native SLRUM and Moab/TORQUE+ALPS, respectively. On both systems, we rely on the ScaLAPACK implementation from the high performance Cray LibSci numerical library. The Haswell nodes on Shaheen-2 have a theoretical peak performance of approximately 1.18 TFlops/s (vector) and 294 GFlops/s (scalar) while the corresponding values for the nodes on the KNL system are 2.6 TFlops/s (vector) and 334 GFlops/s (scalar). All runs for a given process configuration have been submitted in the same job submission script to reduce the impact from the system jitter. All the experiments are performed using IEEE double-precision arithmetic.

Finally, in the subsequent experiments, we switch from Equation 1 to Equation 2, if $c_k$ is smaller than 100, so that the code can exploit the matrix properties (e.g., the symmetry) and may increase the performance of the iterative procedure, while speeding up the convergence rate.

5.2 Impact of the Grid Topology on Performance

![Graphs showing performance and time to solution on Shaheen-2 and KNL system](image)

Fig. 1. Performance impact of square and rectangular grid topology ($P \times Q$) for ill-conditioned matrix on 1152 nodes.
Fig. 2. Performance impact of square and rectangular grid topology ($P \times Q$) for well-conditioned matrix on 1152 nodes.

The execution time and performance of the QDWH algorithm for a square ($P = 192, q = 192$) and rectangular ($P = 128, q = 288$) grid topology is shown in Figures 1 and 2 for ill-conditioned (cond = 1e16) and well-conditioned (cond = 1) synthetic matrices, respectively, of various sizes and on both platforms. The rectangular topology ($P < Q$) yields higher performance in all cases, up to 31% / 33% and up to 40% / 45% on Shaheen-2 / KNL system for ill and well-conditioned matrices, respectively. This performance gain for the rectangular processor grid configuration may be attributed to less data movement during the panel factorization (thanks to a lower $P$ number of processors) and improved parallelism in the update of the trailing submatrix (thanks to a higher $Q$ number of processors) of the various matrix operations involved in the QDWH algorithm. A deeper analysis, based on profiling results obtained from the ratios between computation and communication time, is provided in Section 6.1 with a focus on the largest matrix size.

5.3 QDWH Scalability for the Polar Decomposition

Figure 3 shows the parallel performance of QDWH in Tflop/s across various matrix sizes and number of MPI processes on both platforms. QDWH achieves around 200 Tflop/s on 1152 nodes of Shaheen-2 and 120 Tflop/s on 576 nodes of KNL system for ill and well-conditioned matrices, respectively. The obtained sustained peak performance numbers of the QDWH-based polar decomposition are almost identical for a given system, regardless of the matrix condition number. This can be explained by a similar arithmetic intensity achieved by the matrix computation kernels involved during the QDWH iterations. Figure 4 reports the elapsed time in seconds using the same configurations introduced in Figure 3. For rather small matrix sizes, the strong scalability of QDWH is limited due to a small workload on each processor. This indicates a low computation / communication ratio and affects the overall performance. However, the scalability decently improves as the matrix sizes increases and this trend can be observed on both platforms. By running on the same number of cores on both systems, we may be able to compare the sustained performance obtained across systems. Considering that both systems rely on the Aries network interconnect, the main difference
is the clock frequency of a single core. Shaheen-2’s single core runs at a 60% frequency higher than KNL system, which roughly corresponds to the QDWH performance discrepancy noticed between both systems.

5.4 Performance Comparisons of the Polar Decomposition

Figures 5 and 6 compare the implementation of our QDWH-based polar decomposition with two other state-of-the-art implementations of the polar decomposition, on ill and well-conditioned
matrices, respectively, using various numbers of computational nodes. The first one uses an SVD-based polar decomposition, as described in (Sukkari et al. 2016b), and relies on ScaLAPACK PDGESVD from the Cray LibSci numerical library. The second one uses also a QDWH-based polar decomposition, as implemented in the high performance library Elemental (Poulson et al. 2013). Unlike ScaLAPACK, which links the algorithmic and distribution block sizes, Elemental uses instead an elemental distribution, which fixes the distribution block sizes to one, and may provide more flexibility for computation/communication optimizations. The range of matrix sizes are adjusted for each subfigure, depending on the number of computational nodes. Our QDWH-based implementation for the polar decomposition outperforms the SVD-based implementation up to 5x and 12x on ill-conditioned matrices on Shaheen-2 and KNL system, respectively. These high speedup numbers can be explained by the slow convergence of the QR iteration for the SVD solver within PDGESVD. Compared to the SVD-based approach, our implementation seems also more tolerant to bandwidth starvation, as seen for manycore KNL architectures, having less bandwidth per core than Haswell architectures. On well-conditioned matrices, although PDGESVD performs better, our QDWH-based implementation for the polar decomposition achieves up to 3x and up to 4x on Shaheen-2 and KNL system, respectively. Furthermore, our QDWH-based implementation for the polar decomposition outperforms the QDWH-based Elemental implementation up to 4x on ill-conditioned matrices on both systems. On well-conditioned matrices, our QDWH-based implementation for the polar decomposition achieves up to 2x and up to 5x on Shaheen-2 and KNL system, respectively. Figures 5 and 6 permit also to investigate the scalability of the other implementations for the polar decomposition. Given the high number of processors, the matrix sizes being studied may engender small workload per processor, and therefore, highlight the impact of data motion on the overall performance, i.e., on-node communications through QuickPath/OmniPath network interconnects for Shaheen-2 and KNL system, respectively, or inter-node communications through the Aries network interconnect for both systems. Our QDWH-based implementation for the polar decomposition is able to better cope with challenging mode of operations, such as strong scaling.

5.5 Performance Comparisons of Various SVD Solvers

Figure 7 presents the elapsed time of various SVD solvers for well and ill-conditioned matrices, using various processor configurations on both systems. There are three SVD solvers to distinguish: (1) the standard PDGESVD routine from ScaLAPACK, (2) our QDWH-based SVD solver using the two-stage symmetric eigensolver from the ELPA library (Marek et al. 2014) v2015.11.001, or (3) the one-stage divide-and-conquer symmetric eigensolver PDSYEVD from ScaLAPACK. Although our two variants of QDWH-based SVD solvers perform more flops than PDGESVD (see Table 1), they outperform it by up to 3.6x / 1.7x speedups for ill and well-conditioned matrices, respectively, on Shaheen-2 and by up to 9x / 2x speedups for ill and well-conditioned matrices, respectively, on KNL system. Last but not least, although one can plug in any dense symmetric eigensolvers in the backend, the variant (3) is currently preferred since it relies on the standard and widely-used ScaLAPACK library. In fact, ScaLAPACK is being supported by many high performance vendor numerical libraries, including the Cray LibSci.

6 PROFILING AND PERFORMANCE MODELING

This section provides deeper information through application profiling to further support the performance results presented in Section 5. Computation time is first separated from communication time, followed by an analysis of peak performance. Finally, communication is investigated in detail and modeled as a function of problem size.
Fig. 5. Performance comparison of QDWH implementations for ill-conditioned matrix on different numbers of computational nodes.

6.1 Profiling the Impact of Grid Topology
Possible reasons for the superiority of the rectangular grid topology over the square topology have been mentioned in Section 5.2. This section provides a simplified yet comprehensive profiling view by splitting the execution time into computation and communication. Table 2 shows the ratios between communication and computation time for the QDWH experiments, depicted in Figures 1 and 2 for the largest matrix size $n = 122880$, on Shaheen-2 and KNL system. Note that these ratios are computed by intercepting the MPI calls and measuring the communication time using an instrumented executable, while the total time in column 4 has been measured with a fully optimized executable, as used for Figures 1 and 2. The impact of intercepting the MPI calls on the...
Fig. 6. Performance comparison of QDWH implementations for well-conditioned matrix on different numbers of computational nodes.

ratio between communication and computation is assumed to be negligible, although this overhead may increase the communication part in the ratio. For both rectangular and square processor layouts, the communication is predominant over computation. This is typically the case when the work per rank diminishes, such as in a strong scaling limit context. The rectangular layout shows a slightly larger computation component, which could be attributed to the reduction of excessive data movement in the panel factorization compared to the square case. Indeed, Figure 8 highlights the ratios of the communication times between square and rectangular grid topologies for ill-conditioned matrices of various sizes on Shaheen-2. The square grid topology performs up to 1.5x more communications than the rectangular grid topology. Another reason may be
(a) $P=16, Q=36$ on Shaheen-2.

(b) $P=16, Q=36$ on KNL system.

(c) $P=32, Q=72$ on Shaheen-2.

(d) $P=32, Q=72$ on KNL system.

(e) $P=64, Q=144$ on Shaheen-2.

(f) $P=64, Q=144$ on KNL system.

(g) $P=128, Q=288$ on Shaheen-2.

(h) $P=128, Q=288$ on KNL system.

Fig. 7. Performance comparison of SVD solvers on different process grid configurations.

Fig. 8. Ratios of the communication time between square and rectangular grid topologies for different matrix sizes.
the improved degree of parallelism during the update of the trailing submatrix of the underlying methods in the rectangular case. However, a deeper investigation of communication as given in Sec. 6.3 is required to better understand the impact of the grid topology.

### 6.2 Percentages of Theoretical Peak Performance

Although our QDWH-based polar decomposition starts to be communication-bound when reaching the limit of strong scaling, we here assess the compute-bound part of our implementation. As described in Section 5.1, the compute nodes on Shaheen-2 and KNL system have a theoretical peak performance of approximately 1.18 TFlops/s and 2.6 TFlops/s, respectively, when AVX2 and AVX512 vectorizations are taken into account. Using the expressions for the algorithmic complexity of QDWH from Section 3.2, i.e., $(13 + 1/3)n^3$ for well-conditioned and $(43 + 2/3)n^3$ for ill-conditioned matrices, as well as the timings reported in Table 2, one can compute the number of floating point operations per second and express these values relative to the peak performance of the various platforms, as shown case-by-case in Table 3.

For a matrix size of $n = 122880$, 576 compute nodes were used on KNL system and 1152 on Shaheen-2. The ratios from Table 2 also allow to exclude the communication time from the total time and thus to estimate the performance of the pure calculation, which is also reported in Table 3. Typical fractions of peak performance reached on a single node of Shaheen-2 and KNL system using a simple matrix multiplication (dgemm) benchmark are 82% and 59%, respectively, which shows that the part of the algorithm involved with the computation in QDWH is able to reach a satisfactory fraction of the peak performance. In some cases, this performance is even slightly larger than the sustained dgemm performance on single nodes, since the computation part may be underestimated due to the overhead of the MPI communication profiling tool necessary to calculate the ratios, as already mentioned in Section 6.1.

### 6.3 Towards Performance Modeling

The MPI communication plays a dominant role for the QDWH experiments carried out on 1152 compute nodes of Shaheen-2, as can be seen in Table 2. Detailed profiles have been generated with the Cray Perftools suite on Shaheen-2 only, for ill-conditioned matrices on square $(P = 192, Q = 192)$ and rectangular $(P = 128, Q = 288)$ grid topologies, to further characterize the complexity of communication as a function of matrix size for this specific node count.

Information, such as the number of calls of point-to-point and collective MPI communication routines as well as the corresponding amount of bytes transferred and message sizes, have been

<table>
<thead>
<tr>
<th>System</th>
<th>Condition</th>
<th>Grid</th>
<th>Time[s]</th>
<th>Comp.[%]</th>
<th>Comm.[%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shaheen-2</td>
<td>Well</td>
<td>Square</td>
<td>179.37</td>
<td>12</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>101.38</td>
<td>21</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>Ill</td>
<td>Square</td>
<td>541.99</td>
<td>14</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>377.87</td>
<td>19</td>
<td>81</td>
</tr>
<tr>
<td>KNL system</td>
<td>Well</td>
<td>Square</td>
<td>286.45</td>
<td>10</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>170.84</td>
<td>16</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>Ill</td>
<td>Square</td>
<td>899.39</td>
<td>11</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>615.10</td>
<td>15</td>
<td>85</td>
</tr>
</tbody>
</table>

Table 2. Ratios between communication and computation time for the QDWH experiments with square $(P = 192, Q = 192)$ and rectangular $(P = 128, Q = 288)$ grid topologies, extracted from Figures 1 and 2, for the largest matrix size $n = 122880$. Data is given for well and ill-conditioned matrices on both systems.
Table 3. Sustained performance in TFlops/s and percentage of theoretical peak performance including and excluding the time spent for communication for the cases shown in Table 2. The theoretical peak performance for the various platforms are reported in Section 5.1.

<table>
<thead>
<tr>
<th>System</th>
<th>Condition</th>
<th>Grid</th>
<th>TFlops/s</th>
<th>[%] of Peak Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shaheen-2</td>
<td>Well</td>
<td>Square</td>
<td>137.92</td>
<td>10.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>244.02</td>
<td>17.95</td>
</tr>
<tr>
<td></td>
<td>Ill</td>
<td>Square</td>
<td>149.49</td>
<td>11.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>214.91</td>
<td>15.81</td>
</tr>
<tr>
<td>KNL system</td>
<td>Well</td>
<td>Square</td>
<td>86.36</td>
<td>5.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>144.81</td>
<td>9.67</td>
</tr>
<tr>
<td></td>
<td>Ill</td>
<td>Square</td>
<td>90.08</td>
<td>6.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>131.72</td>
<td>8.80</td>
</tr>
</tbody>
</table>

extracted from these profiles. Without considering the details of the QDWH algorithm, simple polynomial fits are used to model the quantities as a function of matrix size.

In general, the execution time of an application instrumented for profiling may be larger than the time spent by a fully optimized executable to process the same problem because instrumentation typically affects optimizations such as inlining. Furthermore, data collection for profiling at runtime may engender overheads. However, it is assumed that the bytes transferred and number of messages are not noticeably affected by profiling.

The profiles show that blocking point-to-point communication predominates, compared to collective communication for all matrix sizes and both grid topologies, as shown in Figure 9 and Figure 10, respectively. In particular, Figure 9 shows the number of calls, total bytes, and message sizes sent over MPI_Send, where (a), (c), and (e) relate to the square grid topology and (b), (d), and (f) to the rectangular one, as a function of matrix size $n$. The data is always an average over the total number of MPI ranks. From the profiles, four ScaLAPACK computational routines emerge to be predominantly calling MPI_Send with QDWH, i.e. pdgeqr, pdposv, pdorgqr, and pdor2gqr (see also the QDWH pseudo-code in (Sukkari et al. 2016b)). This can be seen in (a)-(d) by summing up the individual data for these four routines and compare it to the total sum for all routines in the program. The polynomial fits show that the number of calls scales like $O(n)$ while the number of bytes sent scales like $O(n^2)$, where $n$ is the matrix size. This is valid for both square and rectangular grid topologies.

It is worth noting that the number of bytes transferred during point-to-point communication in the square topology is smaller than in the rectangular case, as reported in Figures 9(c) and 9(d), respectively. This seems to contradict the superior performance of the rectangular layout at first sight. Indeed, the update of the trailing submatrix is the richest phase in terms of point-to-point communications. Since more processors are involved in that phase when using a rectangular instead of a square topology, the number of transferred bytes is then higher when dealing with rectangular topology. The same holds for the total number of messages. However, the effective time spent in communication is larger for the square topology, as shown by the ratios in Figure 8, which means that more time is spent in blocking during point-to-point communication and collective synchronization compared to the rectangular case. The better load balance for the rectangular topology can be attributed to a higher degree of parallelism during the update of the trailing submatrix.
Figures 10(c) and 10(d) demonstrate an opposite trend, when it comes to assessing collective communications. The panel factorization is the richest phase in terms of collective communications, especially for the QR-related routines during/after the calculation of the Householder reflectors. Therefore, the number of transferred bytes is higher when using a square compared to a rectangular topology, since more processors participate in the panel factorization in the former than the latter configuration. Furthermore, for the square topology scenario, since the parallelism is limited in the panel factorization, mainly characterized by memory-bound operations, as well as during the update of trailing submatrix, with a fewer number of processors, the overall obtained performance is, therefore, lower than the rectangular topology scenario.

The messages are grouped according to their sizes in the ranges $[0B, 16B]$, $[16B, 256B]$, $[256B, 4KiB]$, $[4KiB, 64KiB]$, and $[64KiB, 1MiB]$ and plotted in Figures 9(e) and 9(f), as a function of $n$ for
square and rectangular grid topology, respectively. Most of the messages have sizes in the range $[64\text{KiB}, 1\text{MiB}]$ and this set grows like $O(n)$. This finding is consistent with the linear scaling of the total execution time shown in Figures 5(g) and (h), considering that the simulations are communication-bound. The $O(n^2)$ growth of the total number of bytes transferred is apparently not yet strong enough to affect the linear scaling of the messages in $[64\text{KiB}, 1\text{MiB}]$, and therefore, the total execution time.

Figures 10(e) and 10(f) present a similar analysis, as in Figures 9(e) and 9(f), but for collective communication. In particular, MPIReducers and MPI_Bcast are considered, where the latter is less dominant. Again the number of calls and bytes transferred scale like $O(n)$ and $O(n^2)$, respectively, and the majority of messages for MPIReducers are contained in $[64\text{KiB}, 1\text{MiB}]$.

All in all, in the strong scaling context, the message counts linearly grow like $O(n)$ over the problem sizes and this translates into the observed linear scaling of the total execution time in performance figures, as reported in this section.

7 CONCLUSION AND FUTURE WORK

This paper highlights our high performance QDWH-based implementation of the polar decomposition and its extension to SVD solver on two leading-edge supercomputers. Performance impact of the processor grid topology has been studied and a performance model has been derived to help end-users properly calibrate the needed resources for the problem size of interest. Our QDWH implementations outperform their current state-of-the-art counterparts for the polar decomposition and SVD solver. They also demonstrate their abilities to still extract performance in strong scaling mode of operation, where workload per processing units is very limited, while mitigating the engendered overhead.

We would like also to investigate fine-grained computations using task-based programming model to break the ScaLAPACK bulk synchronous programming model in the context of the Chameleon (CHAMELEON 2016) and DPLASMA (Bosilca et al. 2011) libraries. These libraries rely on a dynamic runtime system, to favor asynchronous computational task scheduling, while using non-blocking communications to not only overlap computations but also communications themselves. Moreover, this fine-grained computation paradigm will enable to exploit the matrix identity structure during the QR-based iteration of QDWH (Sukkari et al. 2017), resulting into lower algorithmic complexity and data movement. Our QDWH-based polar decomposition and its SVD extension are freely available at https://github.com/ecrc/qdwh.git and https://github.com/ecrc/ksvd.git, respectively, and have been integrated into the Cray Scientific numerical library LibSci v17.11.1.

ACKNOWLEDGMENT

The authors would like to thank Jack Poulson from Google Inc. for his help in tuning Elemental and the vendor Cray for systems’ remote accesses in the context of the Cray Center of Excellence awarded to the Extreme Computing Research Center at KAUST. The authors would like also to thank Mohammad Hadi and Nick Hill from Cray for their help into integrating QDWH-based SVD solver into Cray LibSci library. For computer time, this research used the resources from KAUST Supercomputing Laboratory for Shaheen-2 core hour allocations.

REFERENCES


Fig. 10. Number of \texttt{MPI\_Reduce} and \texttt{MPI\_Bcast} calls and corresponding amount of bytes transferred per MPI rank as a function of matrix size \( n \). Data shown for square \((P = 192, Q = 192)\) and rectangular \((P = 128, Q = 288)\) communication layouts. For (a)-(d), the data is given for the four most time consuming ScalAPACK routines pdgqr\(r\), pdposv, pdorgq, and pdorgqr, where the number of calls to \texttt{MPI\_Reduce} and \texttt{MPI\_Bcast} are summed up, as well as for the sum of these four routines. The total number of \texttt{MPI\_Reduce} and \texttt{MPI\_Bcast} calls per rank for all routines is also plotted as a reference as well as the sum of these two values (total). Subfigures (e) and (f) show again the total number of \texttt{MPI\_Reduce} calls as stacked bars, where the bars are partitioned according to the number of calls with message sizes in certain ranges and are appropriately colored.

http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6008655


Received January YYYY; revised January YYYY; accepted January YYYY