Massively Parallel Polar Decomposition on Distributed-Memory Systems

HATEM LTAIEF and DALAL SUKKARI, Extreme Computing Research Center, King Abdullah University of Science and Technology, KSA
ANIELLO ESPOSITO, Cray EMEA Research Lab, UK
YUJI NAKATSUKASA, Mathematical Institute, University of Oxford, UK
DAVID KEYES, Extreme Computing Research Center, King Abdullah University of Science and Technology, KSA

We present a high-performance implementation of the Polar Decomposition (PD) on distributed-memory systems. Building upon on the QR-based Dynamically Weighted Halley (QDWH) algorithm, the key idea lies in finding the best rational approximation for the scalar sign function, which also corresponds to the polar factor for symmetric matrices, to further accelerate the QDWH convergence. Based on the Zolotarev rational functions—introduced by Zolotarev (ZOLO) in 1877— this new PD algorithm ZOLO-PD converges within two iterations even for ill-conditioned matrices, instead of the original six iterations needed for QDWH. ZOLO-PD uses the property of Zolotarev functions that optimality is maintained when two functions are composed in an appropriate manner. The resulting ZOLO-PD has a convergence rate up to seventeen, in contrast to the cubic convergence rate for QDWH. This comes at the price of higher arithmetic costs and memory footprint. These extra floating-point operations can, however, be processed in an embarrassingly parallel fashion. We demonstrate performance using up to **102, 400** cores on two supercomputers. We demonstrate that, in the presence of a large number of processing units, ZOLO-PD is able to outperform QDWH by up to 2.3X speedup, especially in situations where QDWH runs out of work, for instance, in the strong scaling mode of operation.

CCS Concepts: • **Mathematics of computing → Mathematical software performance**; • **Computing methodologies → Parallel algorithms**; **Massively parallel algorithms**;

Additional Key Words and Phrases: Polar Decomposition, Zolotarev Functions, Parallel Algorithms, Strong Scaling, Distributed-Memory Systems

ACM Reference Format:
Hatem Ltaief, Dalal Sukkari, Aniello Esposito, Yuji Nakatsukasa, and David Keyes. 2018. Massively Parallel Polar Decomposition on Distributed-Memory Systems. *ACM Trans. Parallel Comput.* 1, 1, Article 1 (January 2018), 15 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Today's hardware landscape has persuaded computational scientists to rethink their underlying mathematical algorithms in order to take advantage of ever-increasing single node core counts

Authors' addresses: Hatem Ltaief; Dalal Sukkari, Extreme Computing Research Center, King Abdullah University of Science and Technology, 4700 King Abdullah Blvd, Thuwal, Jeddah, 23955, KSA, Hatem.Ltaief,Dalal.Sukkari@kaust.edu.sa; Aniello Esposito, Cray EMEA Research Lab, Bristol, UK, esposito@cray.com; Yuji Nakatsukasa, Mathematical Institute, University of Oxford, Oxford, UK, Yuji.Nakatsukasa@maths.ox.ac.uk; David Keyes, Extreme Computing Research Center, King Abdullah University of Science and Technology, 4700 King Abdullah Blvd, Thuwal, Jeddah, 23955, KSA, David.Keyes@kaust.edu.sa.

1539-9087/2018/1-ART1 $15.00
https://doi.org/0000001.0000001

for maximizing performance. For instance, Intel Knights Landing provides up to 72 cores with potentially 288 threads when hyperthreading is enabled, while NVIDIA has just announced a new GPU codenamed *Volta* with 84 streaming multiprocessors. While the scientific community has routinely enjoyed weak scaling across the entire system for the last two decades, the shrinking single chip memory per core ratio has exposed the vulnerability of the distributed-memory only execution model. Hybrid algorithmic adaptations that embrace strong scaling within a node have become critical to harvesting latent hardware capability, especially moving towards the exascale frontier [5].

Given this challenging apparatus, we redesign the distributed-memory implementation of the polar decomposition (PD) for dense matrices, a fundamental matrix decomposition revealing the nearest orthogonal matrix [10, Ch. 8], and more recently used as the building block for spectral divide-and-conquer algorithms to compute the singular value decomposition (SVD) of a general nonsymmetric matrix as well as the eigenvalue decomposition (EIG) for Hermitian matrices. These aforementioned matrix decompositions represent some of the main linear algebra algorithms required for a broad class of scientific applications [8, 9, 25]. Originally introduced by Nakatsukasa and Higham [18], the current PD employs the inverse-free, iterative QR-based Dynamically Weighted Halley (QDWH) algorithm. QDWH relies on conventional dense linear algebra operations such as QR/Cholesky factorizations and matrix-matrix operations, Level 3 BLAS compute-bound kernels, and are all widely available in vendors' optimized numerical libraries. Although QDWH is optimized for reducing data movement and uses computationally intensive numerical kernels, its SVD solver variant may perform, for instance, up to three times more floating-point operations (flops) than the standard SVD solver based on reduction to bidiagonal form, as implemented in the ScaLAPACK library [3]. This may sound prohibitive, but in practice, these extra flops are compensated by very efficient and high concurrency numerical kernel executions, as demonstrated in the first high performance distributed-memory QDWH implementation [21, 24]. The corresponding open-source software release is freely available at https://github.com/ecrc/qdwh.

Herein, we describe a novel high performance PD implementation, an algorithm introduced by Nakatsukasa and Freund [16], which is a higher-order variant of QDWH. It uses the best rational approximation to the sign function (Zolotarev functions) derived by Zolotarev (ZOLO-PD) [27], along with its remarkable property that composing two Zolotarev functions appropriately results in another Zolotarev function, of much higher degree. In fact, this new ZOLO-PD algorithm converges within two iterations in double precision, for matrices with condition number $\leq 10^{15}$, instead of the original six iterations, as seen in QDWH. The resulting ZOLO-PD convergence rate is up to seventeen, more than five times the cubic convergence rate observed for QDWH. However, there is no free lunch: this comes at the price of higher arithmetic costs and memory footprint than QDWH for the polar decomposition. These extra flops may however be processed in an embarrassingly parallel fashion. This *naturally* strong scaling algorithm is therefore well-suited for exploiting high granularity computing resources, as already exhibited by the current fastest supercomputer in the world [1] with more than 10 million cores.

Our new high performance ZOLO-PD implementation relies on ScaLAPACK [3] and its inherent two-dimensional block cyclic data distribution to scatter the matrix over the memories of remote nodes while reducing the impact of load imbalance. ZOLO-PD is able to excel in situations where QDWH runs out of work in the context of strong scaling experiments. Our experiments demonstrate that ZOLO-PD is able to outperform QDWH by up to 2.3X speedup using a Cray XC40 system on 3200 Intel two-socket 16-core Haswell nodes with a total of **102, 400** cores.

The remainder of the paper is organized as follows. Section 2 provides a literature review for the current PD algorithms and their corresponding implementations. Section 3 briefly recalls the main algorithmic phases of the QDWH algorithm. Section 4 describes the ZOLO-PD algorithm and

Section 5 highlights its high performance implementation. Section 6 details the ZOLO-PD algorithmic complexity and memory footprint and compares them against QDWH. Section 7 presents numerical accuracy, performance, profiling and scalability results. We conclude in Section 8.

2 RELATED WORK

Polar decomposition (PD) algorithms have been investigated intensively in terms of convergence theory, stability and accuracy [4, 6, 7, 11–14]. Classical algorithms include one via the SVD (which is clearly too expensive; the opposite direction of using PD for computing SVD is of current interest), and the scaled Newton method [4, 10], which involves explicit matrix inverses (so directly applicable only to square matrices) and is less accurate than QDWH for large matrices [16]. Other less efficient methods, such as Padé iterations, are detailed in [10, Ch. 9].

More recently, its iterative, inverse-free QR-based Dynamically Weighted Halley (QDWH) variant [15, 18] has enhanced the popularity of the PD algorithm in scientific computing with its inverse-free and communication-minimizing nature, together with a cubic convergence rate in addition to a favorable hardware landscape, thanks to the technology scaling (i.e., wider vector units).

Moreover, as described in [18], QDWH can be used as a building block for the dense symmetric eigensolver and singular value decomposition [9, 25], which represents a pathfinder toward future research directions. In fact, the first high performance QDWH implementation and its SVD variant were performed on hardware accelerators [23] and distributed-memory systems [21, 24], where the calculation of the polar factor is the most-time consuming phase. Performance results reported show a decent speedup against state-of-the-art SVD solvers.

Around the same time, QDWH was also developed in the high performance software library Elemental [19] on distributed-memory systems. Furthermore, a task-based QDWH has been implemented on various shared-memory hardware architectures [22] using fine-grained computational kernels associated with the StarPU dynamic runtime system [2]. The latter task-based QDWH implementation has shown performance enhancements against QDWH from Elemental and Intel MKL, while ensuring software portability across a wide range of x86, PowerPC and GPU-based systems. Last but not least, the authors in [16] have introduced the ZOLO-PD algorithm, which is projected to further improve the parallel performance, thanks to the high concurrency exposed by the ZOLO-PD algorithm. In [16], however, only MATLAB experiments are presented, with an actual parallel implementation left as future work.

In this paper, we describe, design and implement the ZOLO-PD algorithm on two large distributed-memory systems.

3 THE QDWH-BASED POLAR DECOMPOSITION ALGORITHM

The polar decomposition (PD) of the matrix $A \in \mathbb{C}^{m \times n}$ $(m \geq n)$ is written $A = U_p H$, where U_p has orthonormal columns and $H = \sqrt{A^* A}$ is a symmetric positive semidefinite matrix. To find the polar decomposition, the original dynamically weighted iteration can be derived as follows:

$$\begin{aligned} X_0 &= A/\alpha, \\ X_{k+1} &= X_k(a_k I + b_k X_k^* X_k)(I + c_k X_k^* X_k)^{-1}. \end{aligned} \tag{1}$$

In particular, choosing $(a_k = 3, b_k = 1, c_k = 3)$ reduces (1) to the Halley iteration. In the QDWH algorithm, the scalars (a_k, b_k, c_k) are dynamically chosen instead, to speed up the convergence [15]. After k iterations of (2), the singular values Σ_k of X_k are mapped to $\Sigma_k = R_k(\ldots R_2(R_1(\Sigma)))$, where $R_k(x) = x\frac{a_k + b_k x^2}{1 + c_k x^2}$ is a scaled Zolotarev function (best rational approximant to the sign function on $[-1, -\ell_k] \cup [\ell_{k-1}, 1]$) of type (3, 2), denoted by $Z_3(x; \ell_{k-1})$, where $\ell_0 = 1/\kappa_2(A)$ (or its estimate) and

$\ell_k = R_k(\ell_{k-1})$. Remarkably, the rational function $R_k(\ldots R_2(R_1(\Sigma)))$ is again a Zolotarev function, of much higher type $(3^k, 3^k - 1)$. Together with the exponential convergence of Zolotarev functions, QDWH converges in at most *six* iterations, in double precision for matrices with $\kappa_2(A) \leq 10^{15}$.

The iterates (1) can be computed using the mathematically equivalent but numerically more stable *QR*-based implementation [18]:

$$\begin{aligned} \begin{bmatrix} \sqrt{c_k} X_k \\ I \end{bmatrix} &= \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R, \\ X_{k+1} &= \frac{b_k}{c_k} X_k + \frac{1}{\sqrt{c_k}} \left(a_k - \frac{b_k}{c_k} \right) Q_1 Q_2^*. \end{aligned} \tag{2}$$

This uses the fact [10, p. 219] that $cX(I + c^2 X^* X)^{-1} = Q_1 Q_2^*$, where $\begin{bmatrix} cX \\ I \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R$ is the *QR* decomposition, where $X, Q \in \mathbb{R}^{m \times n}$ and $Q_2, R \in \mathbb{R}^{n \times n}$. This represents the QR-based Dynamically Weighted Halley (QDWH) algorithm. Further details can be found in [16].

Finally, after a few QDWH iterations, (2) can be replaced with a lower-cost Cholesky-based iteration, since X_k becomes well-conditioned:

$$\begin{aligned} X_{k+1} &= \frac{b_k}{c_k} X_k + \left(a_k - \frac{b_k}{c_k} \right) (X_k W_k^{-1}) W_k^{-*}, \\ W_k &= \text{chol}(Z_k),\ Z_k = I + c_k X_k^* X_k. \end{aligned} \tag{3}$$

All in all, the QDWH algorithm performs up to six successive QR/Cholesky-based iterations, depending on the original matrix condition number, see [18] for further details.

4 THE ZOLO-BASED POLAR DECOMPOSITION ALGORITHM

The main idea behind the ZOLO-PD algorithm is to generalize the rational approximant underlying the QDWH iterations. Surprisingly, this also results in an opportunity to parallelize the overall PD procedure *across* iterations.

Once we view the QDWH iterates as a composition of type $(3, 2)$ Zolotarev functions, a natural idea is to use Zolotarev functions of higher type $(2r + 1, 2r)$ for an integer $r \geq 1$. As derived by Zolotarev, the type $(2r + 1, 2r)$ Zolotarev function (the best rational approximant to the sign function) on $[-1, -\ell] \cup [l, \ell]$ is

$$Z_{2r+1}(x; \ell) = Mx \Pi_{j=1}^{r} \frac{x^2 + c_{2j}}{x^2 + c_{2j-1}}, \tag{4}$$

where $M > 0$ and satisfies the following:

$$1 - Z_{2r+1}(1; \ell) = -(1 - Z_{2r+1}(l; \ell)).$$

The scalars $c_1, c_2, \ldots, c_{2r}$ can be computed using the Jacobi elliptic functions $sn(u; \ell'), cn(u; \ell')$. Evaluating $Z_{2r+1}(x; \ell)$ at a matrix argument X to obtain $U Z_{2r+1}(\Sigma; \ell) V^*$ where $X = U \Sigma V^*$ is the SVD can be done by

$$Z_{2r+1}(X; \ell) = MX \Pi_{j=1}^{r} (X^* X + c_{2j} I)(X^* X + c_{2j-1} I)^{-1}.$$

We rewrite this in partial fraction form, to improve the degree of parallelism: we can find $a_j \in \mathbb{R}$ such that

$$Z_{2r+1}(X; \ell) = M(X + \Sigma_{j=1}^{r} a_j X (X^* X + c_{2j-1} I)^{-1}). \tag{5}$$

Again, $Z_{2r+1}(X;\ell)$ in (5) can be computed stably as

$$\begin{aligned} \begin{bmatrix} X \\ \sqrt{c_{2j-1}}I \end{bmatrix} &= \begin{bmatrix} Q_{j1} \\ Q_{j2} \end{bmatrix} R_j, \\ Z_{2r+1}(X;\ell) &= X + \Sigma_{j=1}^{r} \frac{a_j}{\sqrt{c_{2j-1}}} Q_{j1} Q_{j2}^*. \end{aligned} \tag{6}$$

This represents a number of r embarrassingly parallel QR factorizations and matrix-matrix multiplications $Q_{j1}Q_{j2}^*$. The upshot of ZOLO-PD is that by taking $r = 8$, we obtain $\|Z_{2r+1}(Z_{2r+1}(A;\ell_0);\ell_1) - U_p\| \leq 10^{-15}$ for any A with $\kappa_2(A) \leq 10^{12}$, implying that convergence is attained in just two steps.

Similar to the QDWH algorithm, the QR iterations in (6) can be reformulated as Cholesky-based iterations with a lower arithmetic cost, as in (3), once X_k is well-conditioned. This condition is already satisfied at the second iteration even for ill-conditioned matrices.

All in all, the ZOLO-PD algorithm can be seen as a generalization of QDWH into a series of independent QDWH subproblems with only two iterations per subproblem, instead of the original six iterations, as mentioned in Section 3.

5 IMPLEMENTATION DETAILS

Our high performance ZOLO-PD implementation relies on the ScaLAPACK library [3]. Algorithm 1 presents the ZOLO-PD pseudo-code. ScaLAPACK uses the two-dimensional block cyclic data distribution (2D_BCDD) to scatter the matrices across remote main memory's nodes. The internal block size has been set to 64, which is a typical value for tuning most of the dense linear algebra operations occurring in the ZOLO-PD algorithm. ScaLAPACK relies on the Basic Linear Algebra Communication Subprograms (BLACS) library, which is in charge of abstracting data movements through the traditional MPI functions.

The pseudo-code can be split into four computational phases:

(1) **line 1 - line 19:** this phase initializes the two-dimensional grid of processors and instantiates two contexts from the BLACS library. These contexts help in defining processor group, similar to the group concept in MPI with its corresponding communicator. The *ictxt_all* context is used when all processors have to collaboratively participate in the same operation. The *ictxt_local* context is used only when subgroups of processors have to participate in a given operation. These subgroups are defined by a mapping array from line 10 to line 16. This latter context ensures embarrassingly parallel PD iterations.

(2) **line 20 - line 36:** this phase estimates the condition number *cond* of the input matrix. This determines how many iterations are necessary per subproblems. It also sets the *group_id* for each MPI process. It is then is used to identify to which group each MPI process belongs to.

(3) **line 37 - line 110:** this phase executes the bulk of the ZOLO-PD computation. After solving for the Zolotarev rational functions, the processor subgroups simultaneously perform either a QR or a Cholesky-based iteration depending on the condition number *cond* (**line 50 - line 60**). The matrix condition number *cond* of the iterate is then updated for the next iteration. There are then two communication steps: a gather and accumulate step (**line 68 - line 94**), which is similar to MPI_Reduce using the MPI_SUM operation mode, is first performed across all subgroups to a root subgroup, followed by a broadcast step (**line 95 - line 110**) from the root subgroup to all other subgroups. These two communication steps are both handled and encapsulated within the BLACS *pdgemr2d* routine. After these two steps, all subgroups are ready to launch the second and last iteration, in case the matrix is ill-conditioned, or to stop the iteration procedure otherwise.

Algorithm 1 Distributed-memory ZOLO-PD Pseudo-Code based on ScaLAPACK.

```
▷ Set the block size
nb = 64
Cblacs_get( -1, 0, ictxt_all );
Cblacs_gridinit( ictxt_all, "R", nprow_all, npcol_all );
Cblacs_gridinfo( ictxt_all, nprow_all, npcol_all, myrow, mycol );
    ▷ Map processes to different context to solve independent nbprob problems in parallel
int *imap = (int *)malloc(nprow*npcol*sizeof(int));
int *ictxt_id = (int *)malloc(nbprob*sizeof(int));
memset(ictxt_id, -1, nbprob* sizeof(int));
k = 0;
for i = 0; i < nprow; i++ do
    for j = 0; j < npcol; j++ do
        *(imap + i + j * nprow) = nprow*npcol*(int)(myrank_mpi / (nprow*npcol)) + k;
        k = k + 1;
    end for
end for
Cblacs_get(0, 0, ictxt_local);
Cblacs_gridmap(ictxt_local, imap, nprow, nprow, npcol);
Cblacs_gridinfo(ictxt_local, nprow, npcol, myrow, mycol);
*(ictxt_id + (int)(myrank_mpi / (nprow*npcol))) = ictxt_local;
    ▷ Initialize data structures using the 2D-BCDD descinit()
    ▷ Estimate the condition number
pdlacpy(A, descA, B, descB, ictxt_all);
pdgetrf(B, descB, ictxt_all);
pdgecon(B, descB, alpha, ictxt_all);
pdlacpy(A, descA, X, descX, ictxt_local);
pdlascl(X, descX, α, ictxt_local), α ≈ ||A||_2;
k = 1; Li = β * α;
cond = 1/Li;
    ▷ Determine the number of independent problems m_zol to be solved in parallel
choosem(cond, m_zol);
    ▷ Determine the number of required iterations based on the cond
if cond < 2 then
    num_while_itr = 1;
else
    num_while_itr = 2;
end if
group_id = (int)(myrank_mpi / (nprow * npcol));
if group_id == 0 then
    descinit(nb, nb, U_ac, descU_ac); Fill_in(U_ac, descU_ac);
end if
    ▷ Compute the polar decomposition A = U_p H using ZOLO-PD
c = (double *)malloc(2 * m_zol * sizeof(double));
it = 0;
while it < num_while_itr do
    pdlacpy( X, descX, B, descB, ictxt_local);
    it = it + 1;
    kp = 1/cond;
    alpha = acos(kp);
    ▷ Compute the scalars for each subproblem using ELLIPKE Complete elliptic integral and mELLIPJ Jacobi elliptic functions
    mellipke( ictxt_local, K);
    for ii = 1; ii ≤ (2 * m_zol); ii++) do
        mellipj(ii * K / (2 * m_zol+1),ictxt_local, sn, cn, tn);
        c[ii-1] = (sn * sn)/(cn * cn);
    end for
    compute (c, maxc);
    if it ≤ 1 & & maxc > 1e2) then
    ▷ QR-based iterations
        C = [ sqrt(c[2 * ii − 2])I ; B ];
        pdgeqrf (C, descC, tau, ictxt_local);
        pdorgqr (C, descC, tau, ictxt_local);
    ▷ Compute X_k from X_{k−1}
        pdgemm(C(1:m, :), descC, C(m:m+n, :), descC, X, descX, ictxt_local);
    else
    ▷ Cholesky-based iterations
        pdlaset(C, descC, 0.0, 1.0, ictxt_local);
        pdgemm(B, descB, B, descB, C, descC, ictxt_local);
        pdposv(C, descC, B, descB, ictxt_local);
    ▷ Compute X_k from X_{k−1}
        pdgeam(B, descB, X, descX, ictxt_local);
    end if
    k = k + 1;
    ff_1 = 1.; ff_cond = cond;
    for i=1; i ≤ m; i++ do
        ff_1 = ff_1 * (1.+c[2 * i-1])/(1.+c[2 * i-2]);
        ff_cond = ff_cond * (cond * cond+c[2 * i-1])/(cond * cond+c[2 * i-2]);
    end for
    cond = max(ff_cond/ff_1,1);
    ▷ Gather and accumulate the computed U to U_ac on the group_id = 0
    if group_id == 0 then
        descX[1] = -1;
        for j=1; j <nbprob; j++ do
            if group_id ≠ 0 & & group_id != j then
                descX[1] = -1;
            end if
            pdgemr2d(M, N, X, i1, i1, descX, U_ac, i1, i1, descU_ac, ctxt_all);
            if group_id != 0 & & group_id != j then
                descX[1] = ictxt;
            end if
            if group_id == 0 then
                descX[1] = ictxt;
                pdgeadd(alpha, U_ac, descU_ac, alpha, X, descX, ictxt_local);
                descX[1] = -1;
            end if
        end for
    end if
    if group_id == 0 then
        descX[1] = ictxt;
        alpha = 1.0;
        pdlascl( ff_1, alpha, X, descX, ictxt_local);
        if (symm) then
            pdlacpy( X, descX, B, descB, ictxt_local);
            alpha = 0.5;
            pdgeadd ( alpha, B, descB, alpha, X, descX, ictxt_local);
        end if
    end if
    ▷ Broadcast the new U_ac on group_id = 0 to the different subgroup contexts
    if group_id == 0 then
        pdlacpy(X, descX, U_ac, descU_ac, ictxt_local);
        descX[1] = -1;
    end if
    descX[1] = ictxt;
    for j=1; j < nbprob; j++ do
        if group_id ≠ j then
            descX[1] = -1;
        end if
        pdgemr2d(M, N, U_ac, i1, i1, descU_ac, X, i1, i1, descX, ctxt_all);
        if group_id ≠ j then
            descX[1] = ictxt;
        end if
    end for
    descX[1] = ictxt;
end while
    ▷ Compute the Hermitian factor of A
pdgemm(X, descX, A, descA, H, descH, ictxt_all);
```

(4) **line 111:** once the PD iteration procedure has converged to the polar factor, the Hermitian factor is then calculated.

All processes jointly participate in the first, second and fourth computational phases. The third phase involves all processes, however, they have to split into processor subgroups to operate independently from each other in the PD iterations, resulting in less processing units per PD iterations than QDWH.

6 ALGORITHMIC COMPLEXITY AND MEMORY FOOTPRINT

In this section, we compare the algorithmic complexity of the QDWH algorithm against the ZOLO-PD algorithmic variants with successive or independent PD iterations. We consider square matrices $A \in \mathbb{C}^{n\times n}$ for simplicity; the algorithms are directly applicable to rectangular matrices.

The condition number estimate L_0 can be calculated using the LU factorization, which requires $\frac{2}{3}n^3$, followed by a few triangular solvers, which cost $O(n^2)$ flops. As shown in (2) and (6) for QDWH and ZOLO-PD, respectively, the QR-based PD iteration requires the QR factorization of $2n \times n$ matrix for a cost of $(3+\frac{1}{3})n^3$ flops. Then, forming $\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}$ explicitly, needs $(3+\frac{1}{3})n^3$ flops. The product $Q_1Q_2^*$ additionally needs $2n^3$ flops. Therefore, the arithmetic cost of each QR-based iteration is $(8+\frac{2}{3})n^3$ flops [1]. For the Cholesky-based PD iteration in (3), matrix-matrix multiplication involves $2n^3$, the Cholesky factorization needs $\frac{1}{3}n^3$, and solving two linear systems requires $2n^3$. Therefore, the arithmetic cost of a Cholesky-based iteration (3) is $(3+\frac{1}{3})n^3$ per iteration. Computing the positive semidefinite matrix $H = U_p^*A$ requires $2n^3$.

Hence, the overall cost of QDWH is

$$\begin{aligned} \#flops =& \frac{2}{3}n^3 + (8+\frac{2}{3})n^3 \times \#it_{QR} \\ &+(3+\frac{1}{3})n^3 \times \#it_{Chol} \\ &+2n^3, \end{aligned} \tag{7}$$

where $\#it_{QR}$ and $\#it_{Chol}$ correspond to the number of QR-based and Cholesky-based iterations, respectively.

The cost of ZOLO-PD is

$$\frac{2}{3}n^3 + (8+\frac{2}{3})n^3 \times r + (3+\frac{1}{3})n^3 \times r + 2n^3, \tag{8}$$

where the total number of iterations is 2 ($\#it_{QR}$=1 and $\#it_{Chol}$=1), and $r = 1, \ldots, 7, 8$ is the number of independent problems to be solved in an embarrassingly parallel fashion at each iteration, as we mentioned above. As shown in (6), ZOLO-PD solves r embarrassing parallel factorizations, and along the critical path, the arithmetic cost of ZOLO-PD is

$$\frac{2}{3}n^3 + (8+\frac{2}{3})n^3 + (3+\frac{1}{3})n^3 + 2n^3. \tag{9}$$

For ill-conditioned matrices with condition number $\kappa = 10^{12}$, QDWH requires 2 QR-based iterations followed by 4 Cholesky-based iterations, and ZOLO-PD needs two successive iterations with $r = 8$. As far as memory footprint is concerned, there is a trade-off between degree of parallelism and memory allocation. Executing ZOLO-PD with independent problems obviously requires as many distinct data structures to operate on as the number of problems. The following table summarizes and compares the flop count and memory footprint of QDWH and ZOLO-PD

[1]The flop counts here are different from [16] since the counting here does not exploit the symmetry and the identity structure in the bottom block.

for matrices with $\kappa = 10^{12}$. QDWH performs around 2.2 times more flops than the parallel ZOLO-

Table 1. Algorithmic complexity and memory footprint for various PD algorithms with $\kappa_2(A) = 10^{12}$.

	QDWH	Successive ZOLO-PD	Independent ZOLO-PD
# QR-based iterations	2	8	1
# Cholesky-based iterations	4	8	1
Algorithmic complexity	$33n^3$	$100n^3$	$15\ n^3$
Memory footprint	$6n^2$	$6n^2$	$48n^2$

PD version, but this assumes again that there are enough compute and memory resources to simultaneously execute the independent problems on the targeted system.

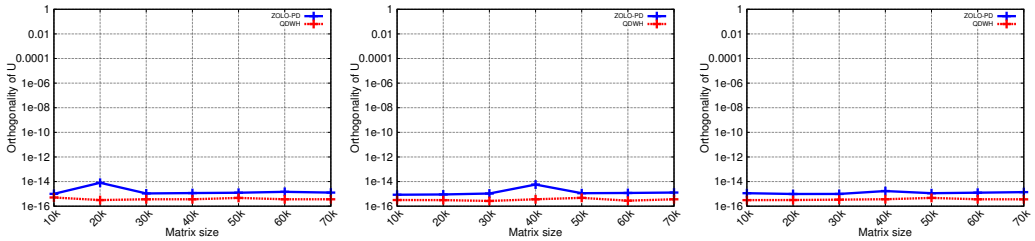
7 PERFORMANCE RESULTS

This section highlights the comprehensive experimental results and reports numerical accuracy, performance, speedup, profiling and scalability results.

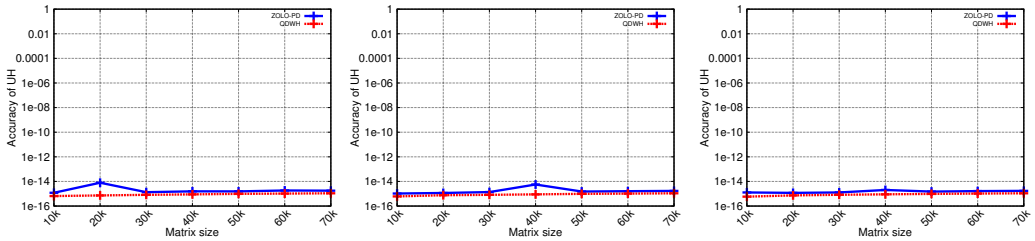
7.1 Apparatus

Our experiments have been conducted on 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-socket Intel Haswell 16-cores running at 2.3GHz and 128GB of DDR3 main memory. We use the vendor ScaLAPACK library from the optimized Cray LibSci numerical library with an internal block size of 64 and the Cray MPICH library. The second test system codenamed *Crystal* is still a Cray XC but now featuring compute nodes with two-socket Intel Broadwell processors each. The core counts of the Broadwell processors range from 18 to 22, i.e., from 36 to 44 per node, with the majority of processors having 18 cores. The base frequency is 2.1GHz and both processors on a node share 128GB of DDR4 memory. We use only 32 cores per compute node, which are evenly distributed among the two sockets in order to properly compare the experiments between both systems. Similarly, the Cray LibSci and MPICH libraries are used. The work load managers on *Shaheen-2* and *Crystal* are native SLRUM and Moab/TORQUE+ALPS, respectively. Hugepages are employed on both systems to improve memory accesses and communication. While *Shaheen-2* is a shared resource with many users during the experiments, *Crystal* is used exclusively for this purpose.

Our code is written in C programming language, is purely MPI, and is linked against sequential Intel Math Kernel Library for single core high performance. This MPI-only programming model turns out to be the best performing configuration for our ScaLAPACK-based code, as also seen in previous works [21, 24]. We compile our code with the Intel compiler suites v15.0.2.164 and v17.0.4.196 on *Shaheen-2* and *Crystal*, respectively. We have generated only ill-conditioned matrices, which are the most challenging numerically, using the *pdmatgen* ScaLAPACK routine with a condition number $\kappa = 10^{12}$. All computations are performed in double precision arithmetic. All experiments have been run five times and only the minimum time to solution is reported for each test case. The studied matrix sizes are selected within the range from 20K to 70K to create opportunities for strong scaling mode of operation, for which ZOLO-PD may demonstrate its potential.

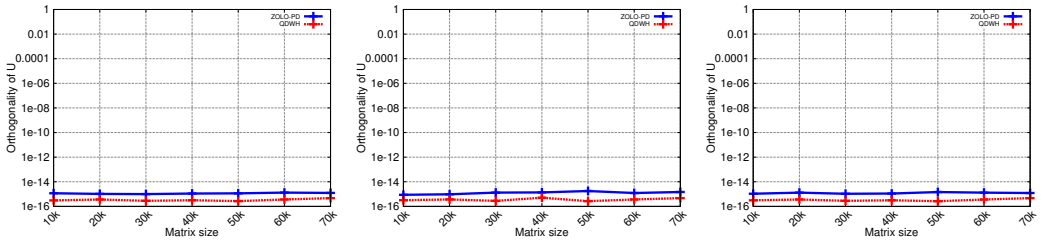


(a) Orthogonality of U_p on 200 nodes. (b) Orthogonality of U_p on 400 nodes. (c) Orthogonality of U_p on 800 nodes.

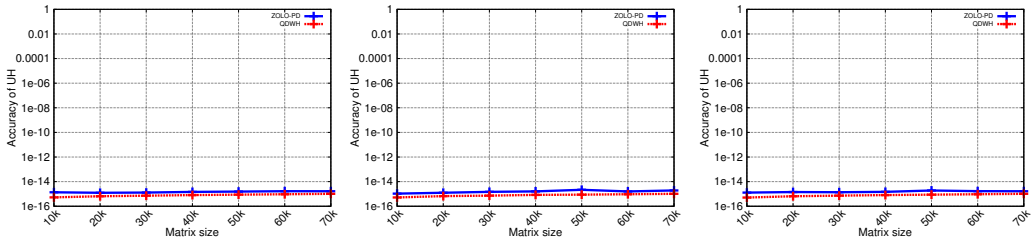


(d) Accuracy of U_pH on 200 nodes. (e) Accuracy of U_pH on 400 nodes. (f) Accuracy of U_pH on 800 nodes.

Fig. 1. Assessing the numerical accuracy/robustness of QDWH and ZOLO-PD on *Shaheen-2*.



(a) Orthogonality of U_p on 200 nodes. (b) Orthogonality of U_p on 400 nodes. (c) Orthogonality of U_p on 800 nodes.



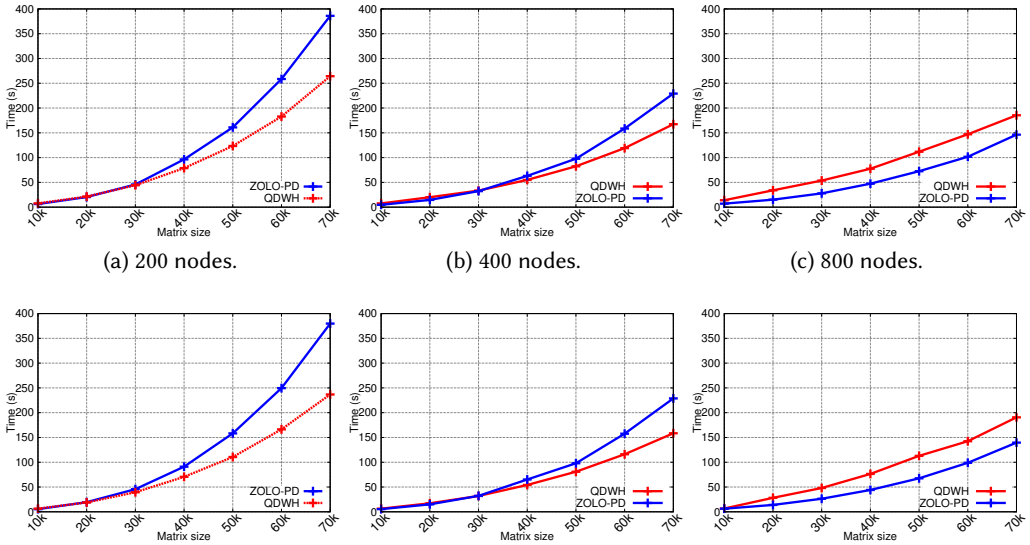
(d) Accuracy of U_pH on 200 nodes. (e) Accuracy of U_pH on 400 nodes. (f) Accuracy of U_pH on 800 nodes.

Fig. 2. Assessing the numerical accuracy/robustness of QDWH and ZOLO-PD on *Crystal*.

7.2 Numerical Accuracy

Before focusing on the performance results, it is crucial to check on the numerical accuracy and stability of the ZOLO-PD algorithm compared to the QDWH implementation. Figures 1 and 2 show the numerical assessment by checking on the orthogonality of the polar factor as well as the backward error using 200, 400 and 800 nodes, on *Shaheen-2* and *Crystal*, respectively.

The orders of the orthogonality and backward errors are similar to QDWH and stand around machine precision, i.e., $1e-15$, which demonstrates the numerical robustness of the algorithm. QDWH is indeed proven to be stable [17], while a proof for ZOLO-PD is currently unavailable; it is conjectured to be stable.



(a) 200 nodes. (b) 400 nodes. (c) 800 nodes.

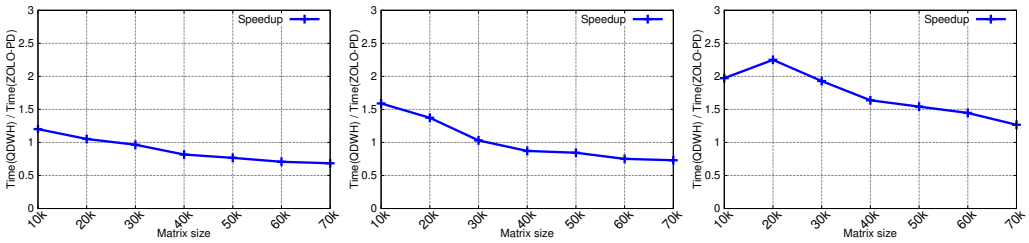
(d) 200 nodes. (e) 400 nodes. (f) 800 nodes.

Fig. 3. Performance comparisons on *Shaheen-2* (a-b-c) and *Crystal* (d-e-f).

7.3 Performance Comparisons

Figure 3 presents the performance comparisons of ZOLO-PD against QDWH using 200, 400 and 800 nodes on *Shaheen-2* and *Crystal*. On 200 nodes for both systems, QDWH outperforms ZOLO-PD across all matrix sizes, especially for large matrix sizes. Indeed, ZOLO-PD performs much more flops than QDWH and is not capable of compensating them by executing the independent problems in parallel, due to the lack of resources. To better understand this phenomenon, one should recall how the PD iteration works for QDWH as opposed to ZOLO-PD (see Section 5). In QDWH, although the PD iterations are done successively, and therefore, all processes work together in computing the QR and Cholesky-based iterations (up to six). In ZOLO-PD, although the PD iterations are performed in parallel, the overall number of processes is split in process subgroups to work independently on each iteration. As a consequence, there are less processing units per subproblem, which is of similar size than the single QDWH problem. Therefore, for the same matrix size and number of processes, we can highlight the fundamental performance trade-off between QDWH and ZOLO-PD: successive versus independent PD iterations and all processes versus process subgroup per PD

iteration. This trade-off stands like a tuning recipe, provides a great flexibility and makes ZOLO-PD amenable to various hardware configurations.



(a) 200 nodes. (b) 400 nodes. (c) 800 nodes.

Fig. 4. Performance speedup ZOLO-PD versus QDWH on *Shaheen-2*.

On 400 nodes, we notice a quite similar performance pattern between QDWH and ZOLO-PD for the small matrix sizes. For the large matrix sizes, the performance gap between both implementations shrinks: as the matrix size increases, the highly parallel ZOLO-PD code starts getting closer to the performance of QDWH, thanks to a better exploitation of the resources available at hand, but still suffers from the lower number of computing resources per independent PD iterations.

On 800 nodes, this configuration actually provides the necessary computational power for ZOLO-PD to outperform QDWH. The performance curves are now inverted. The crossover point occurs directly at the very first small matrix size. While ZOLO-PD exposes plenty of concurrent workloads, QDWH runs out of work and hits the limits of strong scaling by being mostly communication-bound. ZOLO-PD takes better advantage of the underlying hardware than QDWH and demonstrates a clear performance advantage across all matrix sizes. Furthermore, the performance reported on *Shaheen-2* and *Crystal* are very similar, although one would have expected that *Shaheen-2* would have been faster due to a higher clock frequency, as described in Section 7.1. But since *Shaheen-2* resources are shared (e.g., the network interconnect) and not dedicated like *Crystal*, performance on the former may be close to *Crystal* or slightly slower, as shown in Figure 3.

For the subsequent graphs, we decide to only focus on *Shaheen* results, since similar benchmarking numbers have been obtained for *Crystal*.

To further highlight the performance gain, Figure 4 reports the speedup between QDWH and ZOLO-PD. The trend is even clearer: the speedup improves significantly, as the number of node increases. All in all, ZOLO-PD outperforms QDWH by achieving up to 2.3X speedup on up to 102,400 cores (i.e., 800 nodes), especially for small matrix sizes, when running in challenging situations, such as strong scaling mode of operation.

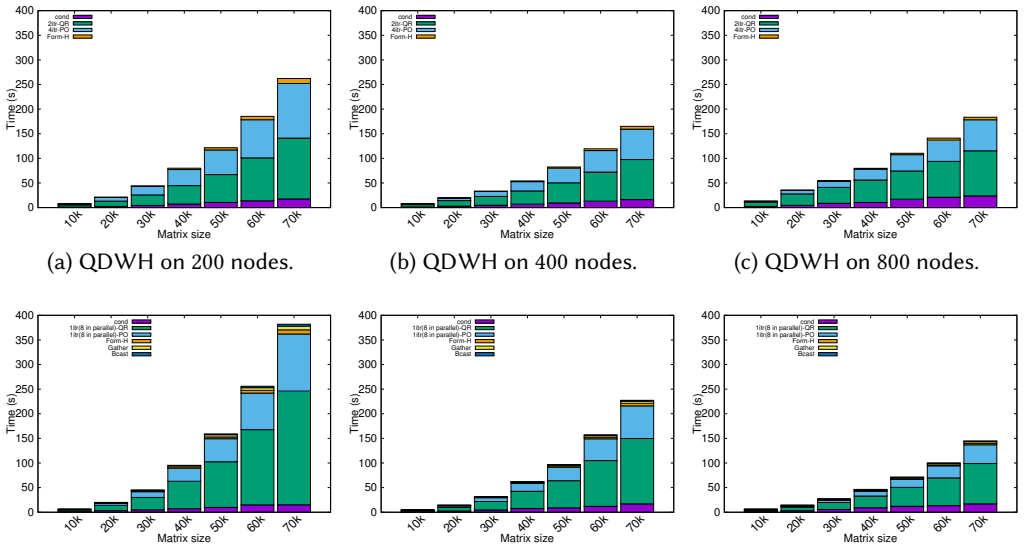
One can also notice that the performance speedup decreases as the matrix size increases, for a given node configuration. This shows QDWH regaining its compute-bound regime of operations, while ZOLO-PD performance starting to cripple due to the algorithmic complexity overhead.

Recalling previous performance comparisons on ill-conditioned matrices obtained in [21], QDWH outperforms its two counterparts, from Elemental [20] and from the SVD-based ScaLAPACK implementation, by up to 4X and 5X, respectively, on *Shaheen-2*. This makes our ZOLO-PD implementation, which is the crux of this paper, outperforming by an order of magnitude the current state-of-the-art software libraries for the polar decomposition.

7.4 Performance Profiling

To better put the performance results from Section 7.3 in perspective, we present in this section some profiling results where we break down the time to solution into the computational phases, introduced in Algorithm 1 of Section 5. Figure 5 shows the time breakdown for QDWH and ZOLO-PD on 200, 400 and 800 nodes. For QDWH, we can see that the PD iterations (i.e., including PO/QR iterations) take up to 85% of the overall execution time. In particular, the QR-based iterations are more time consuming than Cholesky-based iterations. We can see a nice stair-step shape for QDWH, as the matrix size increases for the configuration of 200 nodes. This shape gets attenuated for larger node counts, due to the predominance of communication overheads in performing small dense linear algebra workloads on rather large number of processing units. In particular, for the 800 node case, QDWH experiences a slowdown, since the implementation is mostly communication-bound driven and performs only low arithmetic intensity kernel computation at that scale.

For ZOLO-PD, we observe a nice stair-step shape for all node configurations, as we increase the matrix sizes. The time taken by the independent subproblems to calculate the two PD iterations (one QR-based and one Cholesky-based iterations) is also the predominant computational part of the overall execution time. When the number of nodes increases, the PD iterations keep scaling nicely, since they can be launched in an embarrassingly parallel fashion. The remaining necessary housekeeping operations for ZOLO-PD (i.e., Gather and Scatter operations) are not critical and do not impede the overall parallel performance. All in all, ZOLO-PD benefits from the concurrency exposure and extracts performance from the available processing units.



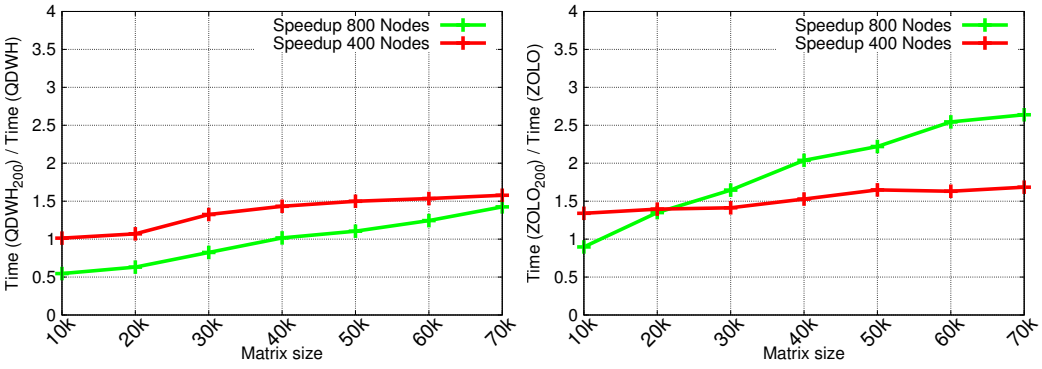
(a) QDWH on 200 nodes. (b) QDWH on 400 nodes. (c) QDWH on 800 nodes.

(d) ZOLO-PD on 200 nodes. (e) ZOLO-PD on 400 nodes. (f) ZOLO-PD on 800 nodes.

Fig. 5. Profiling QDWH and ZOLO-PD on *Shaheen-2*.

7.5 Performance Scalability

Figure 6 shows the strong scalability for each PD implementations. This figure does not compare QDWH against ZOLO-PD but rather looks separately at their own strong scalability on 400 and 800 nodes, using the corresponding elapsed time of the 200 nodes configuration as the reference.



(a) QDWH. (b) ZOLO-PD.

Fig. 6. Performance scalability study of QDWH and ZOLO-PD on *Shaheen-2*.

QDWH has major issues in scaling for all matrix sizes studied in this paper, which are representative of strong scaling scenarios. The 800 node case slows down the overall application and does not leverage performance compared to 400 nodes, let alone the 200 nodes case. On the opposite, ZOLO-PD decently scales up to 800 nodes. There are however some room for further performance improvements. For instance, process placements have not been studied in this paper and this is an important tuning parameter in order to mitigate the data movement overheads in favor of locally cached data within a single node and/or closer physical inter-node communication operations, especially when the hardware occupancy is low [26].

8 CONCLUSION AND FUTURE WORK

We have presented a high performance implementation of the massively parallel polar decomposition using Zolotarev rational functions (ZOLO-PD) on large-scale distributed-memory systems. Compared to the QR-based Dynamically Weighted Halley (QDWH) algorithm for the polar decomposition, ZOLO-PD further exposes concurrency, and therefore, is able to better extract performance from the underlying hardware architecture in a strong scaling mode of operation. Although ZOLO-PD requires more floating-point operations than QDWH, ZOLO-PD still outperforms by up to 2.3X speedup QDWH on up to 102,400 cores for ill-conditioned matrices. This certainly comes at the price of higher memory footprint. This may be mitigated when operating on well-conditioned matrices using Cholesky-based PD iterations. The open-source software will be released and made freely available at https://github.com/ecrc and is currently under consideration by Cray for integration into their numerical software library LibSci, as a follow-up to their QDWH software release [21].

Furthermore, ZOLO-PD opens new research opportunities in investigating the direct performance impact it may have on computing the symmetric eigendecomposition and the SVD, since both traditional linear algebra algorithms suffer from data movement overheads during the panel factorization and the resulting poor scalability. Moreover, we would like to implement a task-based ZOLO-PD, similar to [22], using a dynamic runtime system for task scheduling. This will enable an asynchronous flow of fine-grained computational tasks, which may result in an out-of-order task execution, where communication can potentially be overlapped by computation.

ACKNOWLEDGMENTS

The authors would like to thank Cray Inc. and Intel Corp. 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 supercomputer hosted at the Supercomputing Laboratory at King Abdullah University of Science and Technology (KAUST).

REFERENCES

[1] 2017. TOP500 Supercomputing Sites. http://www.top500.org/. (November 2017).
[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198. https://hal.inria.fr/inria-00550877
[3] L. Suzan Blackford, J. Choi, Andy Cleary, Eduardo F. D'Azevedo, James W. Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clint Whaley. 1997. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia.
[4] Ralph Byers and Hongguo Xu. 2008. A New Scaling for Newton's Iteration for the Polar Decomposition and its Backward Stability. *SIAM J. Matrix Analysis Applications* 30, 2 (2008), 822–843. http://dx.doi.org/10.1137/070699895
[5] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. 2011. The International Exascale Software Project Roadmap. *Int. J. High Perform. Comput. Appl.* 25, 1 (Feb. 2011), 3–60. https://doi.org/10.1177/1094342010391989
[6] Walter Gander. 1985. On Halley's iteration method. *Amer. Math. Monthly* 92, 2 (1985), 131–134.
[7] Walter Gander. 1990. Algorithms for the Polar Decomposition. *SIAM J. Scientific Computing* 11, 6 (1990), 1102–1115. http://dx.doi.org/10.1137/0911062
[8] Gene H. Golub and C. Reinsch. 1970. Singular Value Decomposition and Least Squares Solutions. *Numerische Mathematik* 14 (1970), 403–420.
[9] Gene H. Golub and Charles F. Van Loan. 1996. *Matrix Computations* (third ed.). Johns Hopkins University Press, Baltimore, Maryland.
[10] Nicholas J. Higham. 2008. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. xx+425 pages.
[11] Nicholas J. Higham and Pythagoras Papadimitriou. 1994. A Parallel Algorithm for Computing the Polar Decomposition. *Parallel Comput.* 20, 8 (Aug. 1994), 1161–1173.
[12] C. S. Kenney and A. J. Laub. 1992. On Scaling Newton's Method for Polar Decomposition and the Matrix Sign Function. *SIAM J. Matr. Anal. Appl.* 13 (1992), 688–706. Cited in a personal communication by Alan Laub.
[13] Andrzej Kielbasinski and Krystyna Zietak. 2003. Numerical Behaviour of Higham's Scaled Method for Polar Decomposition. *Numerical Algorithms* 32, 2-4 (2003), 105–140. http://dx.doi.org/10.1023/A:1024098014869
[14] B. Laszkiewicz and K. Zietak. 2006. Approximation of Matrices and a Family of Gander Methods for Polar Decomposition. *BIT Numerical Mathematics* 46, 2 (2006), 345–366. http://dx.doi.org/10.1007/s10543-006-0053-4
[15] Yuji Nakatsukasa, Zhaojun Bai, and François Gygi. 2010. Optimizing Halley's Iteration for Computing the Matrix Polar Decomposition. *SIAM J. Matrix Anal. Appl.* 31, 5 (2010), 2700–2720.
[16] Yuji Nakatsukasa and Roland W. Freund. 2016. Computing Fundamental Matrix Decompositions Accurately via the Matrix Sign Function in Two Iterations: The Power of Zolotarev's Functions. *SIAM Rev.* 58, 3 (2016), 461–493. http://dx.doi.org/10.1137/140990334
[17] Y. Nakatsukasa and N. J. Higham. 2012. Backward Stability of Iterations for Computing the Polar Decomposition. *SIAM J. Matr. Anal. Appl.* 33, 2 (2012), 460–479.
[18] Yuji Nakatsukasa and Nicholas J. Higham. 2013. Stable and Efficient Spectral Divide and Conquer Algorithms for the Symmetric Eigenvalue Decomposition and the SVD. *SIAM Journal on Scientific Computing* 35, 3 (2013), A1325–A1349. https://doi.org/10.1137/120876605 arXiv:http://epubs.siam.org/doi/pdf/10.1137/120876605
[19] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. 2013. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Softw* 39, 2 (2013), 13.

http://doi.acm.org/10.1145/2427023.2427030

[20] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. 2013. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Softw* 39, 2 (2013), 13. http://doi.acm.org/10.1145/2427023.2427030

[21] Dalal Sukkari, Hatem Ltaief, Aniello Esposito, and David Keyes. 2017. A QDWH-Based SVD Software Framework on Distributed-Memory Manycore Systems. *Submitted to ACM Transactions on Mathematical Software, KAUST Technical Report Available at http://hdl.handle.net/10754/626212* (2017).

[22] Dalal Sukkari, Hatem Ltaief, Mathieu Faverge, and David E. Keyes. 2017. Asynchronous Task-Based Polar Decomposition on Manycore Architectures. *Submitted to IEEE Transactions on Parallel and Distributed Systems (minor revision)* (2017).

[23] Dalal Sukkari, Hatem Ltaief, and David E. Keyes. 2016. A High Performance QDWH-SVD Solver Using Hardware Accelerators. *ACM Trans. Math. Softw* 43, 1 (2016), 6:1–6:25. http://doi.acm.org/10.1145/2894747

[24] Dalal Sukkari, Hatem Ltaief, and David E. Keyes. 2016. 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 (Lecture Notes in Computer Science)*, Pierre-François Dutot and Denis Trystram (Eds.), Vol. 9833. Springer, 605–616. http://dx.doi.org/10.1007/978-3-319-43659-3

[25] Lloyd N. Trefethen and David Bau. 1997. *Numerical Linear Algebra.* SIAM, Philadelphia, PA. http://www.siam.org/books/OT50/Index.htm

[26] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (Oct 2017), 3007–3020. https://doi.org/10.1109/TPDS.2017.2703149

[27] E. I. Zolotarev. 1877. Reprinted in his Collected Works, Vol. II, Izdat. Akad. Nauk SSSR, Moscow, 1932, pp. 1–59. In Russian. Application of Elliptic Functions to Questions of Functions Deviating Least and Most from Zero. *Zap. Imp. Akad. Nauk. St. Petersburg* 30, 5 (1877. Reprinted in his Collected Works, Vol. II, Izdat. Akad. Nauk SSSR, Moscow, 1932, pp. 1–59. In Russian).

Received January 2018