Elemental: A New Framework for Distributed Memory Dense Matrix Computations

JACK POULSON, BRYAN MARKER, and ROBERT A. VAN DE GEIJN,
The University of Texas at Austin
JEFF R. HAMMOND and NICHOLS A. ROMERO,
Argonne Leadership Computing Facility

Parallelizing dense matrix computations to distributed memory architectures is a well-studied subject and generally considered to be among the best understood domains of parallel computing. Two packages, developed in the mid 1990s, still enjoy regular use: ScaLAPACK and PLAPACK. With the advent of many-core architectures, which may very well take the shape of distributed memory architectures within a single processor, these packages must be revisited since the traditional MPI-based approaches will likely need to be extended. Thus, this is a good time to review lessons learned since the introduction of these two packages and to propose a simple yet effective alternative. Preliminary performance results show the new solution achieves competitive, if not superior, performance on large clusters.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: *Efficiency*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: linear algebra, libraries, high-performance, parallel computing

ACM Reference Format:
Poulson, J., Marker, B., Van de Geijn, R. A., Hammond, J. R., and Romero, N. A. 2011. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. ACM Trans. Math. Softw. 0, 0, Article 0 (0000), 24 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

With the advent of widely used commercial distributed memory architectures in the late 1980s and early 1990s came the need to provide libraries for commonly encountered computations. In response two packages, ScaLAPACK [Blackford et al. 1997; Anderson et al. 1992; Dongarra and van de Geijn 1992; Anderson et al. 1992; Dongarra et al. 1994] and PLAPACK [Wu et al. 1996; Alpatov et al. 1997; van de Geijn 1997], were created in the mid-1990s, both of which provide a substantial part of the functionality offered by the widely used LAPACK library [Anderson et al. 1999]. Both of these packages still enjoy loyal followings.

One of the authors of the present paper contributed to the early design of ScaLAPACK and was the primary architect of PLAPACK. This second package resulted from a desire to introduce abstraction in order to overcome the hardware and soft-

Authors' addresses: Jack Poulson, Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712, poulson@ices.utexas.edu. Robert A. van de Geijn and Bryan Marker, Department of Computer Science, The University of Texas at Austin, Austin, TX 78712, rvdg@cs.utexas.edu, bamarker@gmail.com. Nichols A. Romero and Jeff R. Hammond, Argonne National Laboratory, 9700 South Cass Avenue, LCF/Building 240, Argonne, IL 60439, {naromero, jhammond}@anl.gov.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

```
Algorithm: $A := \textsc{Chol\_blk}(A)$   Variant 3: down-looking
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$ where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do
    $b = \min(m(A_{BR}), b_{\rm alg})$
    Repartition
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$ where $A_{11}$ is $b \times b$
    ========================================
    $A_{11} := \textsc{Chol}(A_{11})$
    $A_{12} := A_{11}^{-H} A_{12}$          $(\textsc{Trsm})$
    $A_{22} := A_{22} - A_{12}^{H} A_{12}$  $(\textsc{Herk})$
    ========================================
    Continue with $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$
endwhile
```

Fig. 1. Blocked algorithms for computing the Cholesky factorization.

ware complexity that faced computational scientists in the early days of massively parallel computing much like what the community now faces as multicore architectures evolve into many-core architectures and distributed memory architectures become heterogeneous architectures. After major development on the PLAPACK project ceased around 2000, many of the insights were brought back into the world of sequential and multi-threaded architectures (including SMP and multicore), yielding the FLAME project [Gunnels et al. 2001], `libflame` library [Van Zee 2009], and SuperMatrix runtime system for scheduling dense linear algebra algorithms to multicore architectures [Chan et al. 2007b; Quintana-Ortí et al. 2009]. With the advent of many-core architectures that may soon resemble "distributed memory clusters on a chip", like the Intel 80-core network-on-a-chip terascale research processor [Mattson et al. 2008] and the recently announced Intel Single-chip Cloud Computer (SCC) research processor with 48 cores in one processor [Howard et al. 2010], the research comes full circle: distributed memory libraries may need to be (and have been) mapped to single-chip environments [Marker et al. 2011a].

This seems an appropriate time to ask what we would do differently if we had to start all over again building a distributed memory dense linear algebra library. In this paper, we attempt to answer this question by describing a new effort, the Elemental library. This time the solution must truly solve the programmability problem for this domain. It cannot compromise (much) on performance. It must be easy to retarget from a conventional cluster to a cluster with hardware accelerators to a distributed memory cluster on a chip.

Both the ScaLAPACK and PLAPACK projects generated dozens of papers. Thus, this paper is merely the first in what we expect to be a series of papers that together provide the new design. As such it is heavy on vision and somewhat light on details.

2. MATRIX DISTRIBUTIONS AND COLLECTIVE COMMUNICATION

A key insight that underlies scalable dense linear algebra libraries for distributed memory architectures is that the matrix must be distributed to MPI processes (processes hereafter) using a two-dimensional data distribution [Schreiber 1992; Stewart 1990; Hendrickson and Womble 1994]. The p processes in a distributed memory architecture are logically viewed as a two-dimensional $r \times c$ process grid with $p = r \times c$. Subsequently, communication when implementing dense matrix computations can be cast (almost) entirely in terms of collective communication within rows and columns of processes, with an occasional collective communication that involves all processes.

In this section, we discuss redistributions and the collective communications that support them. Readers that are unfamiliar with collective communication algorithms and their costs may wish to consult [Chan et al. 2007a] to gain a basic understanding.

2.1. Motivating example

In much of this paper, we will use the Cholesky factorization as our motivating example. An algorithm for this operation, known as the down-looking variant, that lends itself well to parallelization is expressed using FLAME notation [Gunnels et al. 2001] in Figure 1.

2.2. Two-dimensional (block) cyclic distribution

In order to understand how dense linear algebra algorithms are parallelized on distributed memory architectures, it is crucial to understand how data is typically distributed among processes. To this end, we briefly introduce two-dimensional (block) cyclic matrix distributions.

The first step in defining a two-dimensional cyclic distribution is to take a matrix $A \in T^{m \times n}$, where T is an arbitrary datatype, and to partition A into blocks,

$$A = \begin{pmatrix} A_{0,0} & \cdots & A_{0,N-1} \\ \vdots & & \vdots \\ A_{M-1,0} & \cdots & A_{M-1,N-1} \end{pmatrix},$$

where the size of each $A_{i,j}$ is determined by the user-chosen "distribution block size", say $m_b \times n_b$, with the possible exception of the blocks on the boundaries of A being smaller. A two-dimensional (Cartesian) block-cyclic matrix distribution over an $r \times c$ logical process grid assigns each $A_{s,t}$ to process $((s+\sigma_i) \bmod r, (t+\sigma_j) \bmod c)$, where σ_i and σ_j are arbitrary alignment parameters. In particular, the usual scheme is for each process to store all of the blocks assigned to it in a contiguous matrix; for example, the local matrix of process (s,t) would be

$$A = \begin{pmatrix} A_{\gamma,\delta} & A_{\gamma,\delta+c} & \cdots \\ A_{\gamma+r,\delta} & A_{\gamma+r,\delta+c} & \cdots \\ \vdots & \vdots & \end{pmatrix},$$

where $\gamma \equiv (s-\sigma_i) \bmod r$ is the first block row assigned to the s'th row of the $r \times c$ grid, and likewise $\delta \equiv (t-\sigma_j) \bmod c$ is the first block column assigned to the t'th column of the grid. For the particular case where $m_b = n_b = 1$, the discussed scheme is referred to as an "elemental" or "torus-wrap" matrix distribution [Johnsson 1987; Hendrickson and Womble 1994].

2.3. ScaLAPACK

ScaLAPACK [Blackford et al. 1997; Anderson et al. 1992; Dongarra and van de Geijn 1992; Anderson et al. 1992; Dongarra et al. 1994] was designed to extend the most commonly used LAPACK routines onto distributed-memory computers using two-dimensional block cyclic decompositions. In particular, the approach taken by the vast majority of ScaLAPACK routines is to mirror the LAPACK approach of layering on top of the Basic Linear Algebra Subprograms (BLAS) by instead building upon a parallel versions of the BLAS[1] namely, the PBLAS.

[1]The optimized routines for reduction to tridiagonal form, e.g., `pdsyttrd`, are notable exceptions.

While in theory ScaLAPACK allows blocks $A_{i,j}$ to be rectangular and arbitrarily aligned, routines that exploit symmetry, e.g., Cholesky factorization via `pdpotrf`, require both $m_b = n_b$ and $\sigma_i = \sigma_j$. In addition, the design decisions that underly ScaLAPACK link the distribution block sizes to the algorithmic block size (e.g., the size of block A_{11} in Figure 1). Predicting the best distribution block size is often difficult because there is a tension between having block sizes that are large enough for local Basic Linear Algebra Subprogram (BLAS) [Dongarra et al. 1990] efficiency, yet small enough to avoid inefficiency due to load-imbalance.

One benefit of linking algorithmic and distribution block sizes is that factorizations of small submatrices can usually be performed locally on a single process without any communication. For example, the A_{11} block in Figure 1 is owned by a single process that can locally compute its Cholesky factor. After factorization, the block only needs to be broadcast within the row of processes that owns A_{12}, and then those processes can independently perform their part of $A_{12} := A_{11}^{-H} A_{12}$ with local calls to a triangular solve with multiple right-hand sides routine (`trsm`) (thus only c processes participate in this operation). Finally, A_{12} is redistributed in two different manners such that each process may locally perform its portion of the $A_{22} := A_{22} - A_{12}^H A_{12}$ update.

In the case of LU factorization, linking the distribution and algorithmic block sizes has the added benefit of allowing for the formation of a pipeline within rows of the process grid that can drastically lower the effective communication latency of the factorization [Dongarra and Ostrouchov 1990; Anderson et al. 1992]. This communication pipeline is a fundamental part of the High Performance Linpack (HPL) benchmark [Petitet et al.], but a much shallower pipeline can be created within ScaLAPACK's LU factorization routine [Choi et al. 1994; van de Geijn 1992].

On the other hand, [Sears et al. 1998] extended the work of [Hendrickson et al. 1999] by creating a blocked Householder tridiagonalization routine that exploits an elemental distribution to construct a pipeline within rows of the process grid. This approach provides a large enough speedup over alternative parallelizations that ScaLAPACK has incorporated it via the routines `p[s,d]syntrd` and `p[c,z]hentrd`, which handle redistribution into an elemental distribution, performing the fast tridiagonalization, and then redistributing back to a blocked distribution. It is telling that ScaLAPACK has incorporated an elemental tridiagonalization routine: it shows that elemental distributions are not just a simplification of blocked distributions; for some operations they can provide substantial performance improvements.

2.4. PLAPACK

Contrary to the approach of ScaLAPACK, in PLAPACK the distribution block size is not tied to the algorithmic block size, and it is thus possible to keep the distribution block size small for good load balancing and to separately tune the algorithmic block size to improve the efficiency of local computation.

PLAPACK defines matrix distributions in a manner that is meant to ensure fast redistributions to and from vector distributions [Edwards et al. 1995; van de Geijn 1997], which are defined by subdividing either the rows or columns of a matrix into subvectors of length m_b and assigning them in a cyclic fashion to all processes. The related "induced" matrix distribution in PLAPACK is, for an $r \times c$ grid, a two-dimensional block-cyclic decomposition where $n_b = r\, m_b$. One advantage of this distribution strategy is that the diagonal of any sufficiently large matrix will be evenly distributed amongst all processes. This observation allows for a large conceptual simplification of redistributions that require transposition: each row of the matrix simply needs to be gathered within a row of the process grid to the process owning the diagonal entry, and then scattered within the diagonal process's column of the process grid.

Due to PLAPACK's self-imposed constraint that $n_b = r\,m_b$, it is mildly nonscalable in the sense that, as the number of processes p increases, the widths of distributed matrices must grow proportionally to $r \approx \sqrt{p}$ in order to keep their rows evenly distributed. This mild nonscalability was the result of a conscious choice made to simplify the implementation at a time when the number of processes was relatively small. While lifting the restriction on the aspect ratio of the $m_b \times n_b$ distribution blocks is a matter of "inducing" the matrix distribution from *two* vector distributions (that need only be different by a simple permutation) instead of just one, this generalization was instead implemented within the new package described in this paper, Elemental.

2.5. Elemental

Elemental is a framework for distributed memory dense linear algebra that is designed to be a modern extension of the communication insights of PLAPACK to elemental distributions. While ScaLAPACK linked algorithmic and distribution block sizes, and PLAPACK introduced a mild inscalability through the use of only a single vector distribution, Elemental's simplification is to fix the distribution block sizes at one. This approach is not new and is best justified in [Hendrickson et al. 1999]:

> "In principle, the concepts of storage blocking and algorithmic blocking are completely independent. But as a practical matter, a code that completely decoupled them would be painfully complex. Our code and ScaLAPACK avoid this complexity in different ways. Both codes allow any algorithmic blocking, but ScaLAPACK requires that the storage blocking factor be equal to [the] algorithmic blocking factor. We instead restrict storage blocking to be equal to 1."

There are a number of reasons to argue the restriction to $m_b = n_b = 1$, such as the obvious benefit to load balancing, but perhaps the most convincing is that it eliminates alignment problems that frequently appear when operating on submatrices. For example, if one desires to compute the Schur complement of the bottom right quadrant of a symmetric matrix, the symmetric update is greatly complicated when m_b is not an integer multiple of the height of the top-left matrix and/or n_b is not a multiple of the width. In particular, unaligned symmetric factorizations are not supported in ScaLAPACK. Since every integer is clearly an integer multiple of one, elemental distributions eliminate these burdensome alignment issues and allow for straightforward manipulation of arbitrary contiguous submatrices.

On early distributed memory architectures, before the advent of cache-based processors that favor blocked algorithms, elemental distributions were the norm [Johnsson 1987; Hendrickson and Womble 1994]. We again quote [Hendrickson et al. 1999], where it is noted that

> "Block storage is not necessary for block algorithms and level 3 [BLAS] performance. Indeed, the use of block storage leads to a significant load imbalance when the block size is large. This is not a concern on the Paragon, but may be problematic for machines requiring larger block sizes for optimal BLAS performance."

Similarly, in [Strazdins 1998], the performance benefits from choosing separate algorithmic and (small) distribution block sizes were shown. These insights are more relevant now than before: The algorithmic block size used to be related to the square root of the size of the L1 cache [Whaley and Dongarra 1998], which was relatively small. Kazushige Goto [Goto and van de Geijn 2008] showed that higher performing implementations should use the L2 cache for blocking, which means that the algorithmic block size is now typically related to the square root of the size of the much larger L2 cache. Linking the distribution and algorithmic block sizes will thus create a tension between load balancing and filling the L2 cache.

2.6. Discussion

To some the choice of $m_b = n_b = 1$ may seem to contradict the conventional wisdom that the more process boundaries are encountered in the data partitioning for distribution, the more often communication must occur. To explain why this is not necessarily true for dense matrix computations, consider the following observations regarding the parallelization of a blocked down-looking Cholesky factorization (with $n \geq p$ in order to simplify discussion): In the ScaLAPACK implementation, A_{11} is factored by a single process after which it must be broadcast within the column of processes that owns it. If the matrix is distributed using $m_b = n_b = 1$, then A_{11} can be (all)gathered to all processes and factored redundantly (see Appendix A.3). We note that, if communication cost is ignored, this is as efficient as having a single process compute the factorization while the other cores idle. If implemented optimally, and $b \gtrsim \sqrt{p}$ (where A_{11} is a $b \times b$ matrix), an allgather to all processes is comparable in cost [Chan et al. 2007a] to the broadcast of A_{11} performed by ScaLAPACK: If the process grid is $p = r \times c$, under reasonable assumptions, the former requires $\log_2(p)$ relatively short messages while the latter requires $\log_2(r) \approx \log_2(p)/2$ such messages.

Next, consider the update of A_{12}: In the ScaLAPACK implementation, A_{12} is updated by the c processes in the process row that owns it, requiring the broadcast of A_{11} within that row of processes. Upon completion, the updated A_{12} is then broadcast within rows and columns of processes. If the matrix is distributed using $m_b = n_b = 1$, then columns of A_{12} must be brought together so that they can be updated as part of $A_{12} := A_{11}^{-H} A_{12}$. This can be implemented as an all-to-all collective communication within columns (details of which are illustrated in Appendix A.4), whose cost is ignorable since its communication volume is roughly $1/\sqrt{p}$ of that of later broadcasts/allgathers. Since A_{11} was redundantly factored by each process, the update $A_{12} := A_{11}^{-H} A_{12}$ is shared among all p processes (again, details are illustrated in Appendix A.4). Notice that the ScaLAPACK algorithm would only involve $c \approx \sqrt{p}$ processes in the triangle solves, whereas the Elemental approach involves all p processes. On the other hand, triangle solves only constitute $O(n^2)$ of the $O(n^3)$ flops required for the factorization, so the penalty for the ScaLAPACK approach is mild for large n.

Finally, consider the update of A_{22}: An allgather within rows and columns then duplicates the elements of A_{12} (also illustrated in Appendix A.5) so that A_{22} can be updated in parallel; the ScaLAPACK approach is similar but uses a broadcast rather than an allgather.

In summary, for Cholesky factorization, an elemental distribution requires different communications that are comparable in cost to those incurred when using block-cyclic distributions, but with the benefit of enhancing load balance for operations like $A_{12} := A_{11}^{-H} A_{12}$ and $A_{22} := A_{22} - A_{12}^H A_{12}$.

Perhaps more interestingly, the case for using elemental distributions in the context of Householder tridiagonalization has already been made for us: as previously discussed, ScaLAPACK has already incorporated some of the contributions from [Sears et al. 1998], of which the Householder tridiagonalization routines exploit an elemental distribution over a square subgrid in order to achieve much higher performance than has been demonstrated in blocked distributions. While Sears et al. limited themselves to square grids, we demonstrate that the same techniques easily extend to nonsquare grids, thereby eliminating the need for the extra memory needed for the redistribution, although with less of a performance improvement than when redistributing to and from a square process grid.

3. REPRESENTING PARALLEL ALGORITHMS IN CODE

We now briefly discuss how the different libraries represent algorithms in code.

3.1. ScaLAPACK

The fundamental design decision behind ScaLAPACK can be found on the ScaLAPACK webpage [ScaLAPACK 2010]:

> "Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. (For such machines, the memory hierarchy includes the off-processor memory of other processors, in addition to the hierarchy of registers, cache, and local memory on each processor.) The fundamental building blocks of the ScaLAPACK library are distributed memory versions of the Level 1, 2 and 3 Basic Linear Algebra Subprograms (BLAS) called the Parallel BLAS (PBLAS), and a set of Basic Linear Algebra Communication Subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, all interprocessor communication occurs within the PBLAS and the BLACS. One of the design goals of ScaLAPACK was to have the ScaLAPACK routines resemble their LAPACK equivalents as much as possible."

In Figure 2 we show the ScaLAPACK Cholesky factorization routine. A reader who is familiar with the LAPACK Cholesky factorization will notice the similarity of coding style.

3.2. PLAPACK

As mentioned, PLAPACK already supports independent algorithmic and distribution block sizes. While $m_b = 1$ is supported, the current implementation would require that $n_b = rm_b = r$, so many of the benefits of an elemental distribution would not be available without rewriting much of the library.

Since the inception of PLAPACK, additional insights into dense matrix library development were exposed as part of the FLAME project and incorporated into the `libflame` library. To also incorporate all those insights, a complete rewrite of PLAPACK made more sense, yielding Elemental. For this reason we do not show code samples from PLAPACK.

3.3. Elemental

Elemental is written in the formal style of the `libflame` library but also incorporates a new notation and API for handling distributed matrices. In addition, Elemental takes advantage of C++ through its use of generic programming, const-correctness, and templated datatypes.[2] Like its predecessors, PLAPACK and `libflame`, Elemental attempts to provide a simple user interface by hiding details about matrices and vectors, such as leading dimensions and datatype, within objects rather than individually passing attributes in the style of (Sca)LAPACK. As a result, the vast majority of indexing details that exist in (Sca)LAPACK code disappear, leading to (in our experience) much simpler interfaces and faster development.

Let us examine how the code in Figure 3 implements the algorithm described in Section 2.5. The command

```
PartitionDownDiagonal( A, ATL, ATR,
                          ABL, ABR, 0 );
```

sets $A_{BR} = A$ and A_{TL} as a 0×0 placeholder matrix; for this algorithm, A_{TL} can be thought of as the portion of the matrix that we have completely factored, while the remaining three quadrants represent pieces of A that still require updates. We then run a `while` loop which continues until the entire matrix has been factored, i.e., when

[2]While Elemental currently provides distributed BLAS and LAPACK type routines for the `float`, `double`, `std::complex<float>`, and `std::complex<double>` datatypes, arbitrary real or complex field will be supported as soon as MPI and local BLAS interfaces are made available for them.

```
      SUBROUTINE PZPOTRF( UPLO, N, A, IA, JA, DESCA, INFO )
*
*         < deleted code >
*
         DO 10 J = JN+1, JA+N-1, DESCA( NB_ )
            JB = MIN( N-J+JA, DESCA( NB_ ) )
            I = IA + J - JA
*
*           Perform unblocked Cholesky factorization on JB block
*
            CALL PZPOTF2( UPLO, JB, A, I, J, DESCA, INFO )
            IF( INFO.NE.0 ) THEN
               INFO = INFO + J - JA
               GO TO 30
            END IF
*
            IF( J-JA+JB+1.LE.N ) THEN
*
*              Form the row panel of U using the triangular solver
*
               CALL PZTRSM( 'Left', UPLO, 'Conjugate transpose',
     $                      'Non-Unit', JB, N-J-JB+JA, CONE, A, I, J,
     $                      DESCA, A, I, J+JB, DESCA )
*
*              Update the trailing matrix, A = A - U'*U
*
               CALL PZHERK( UPLO, 'Conjugate transpose', N-J-JB+JA, JB,
     $                      -ONE, A, I, J+JB, DESCA, ONE, A, I+JB,
     $                      J+JB, DESCA )
            END IF
   10    CONTINUE
*        < deleted code >
```

Fig. 2. Excerpt from ScaLAPACK 2.0.0 Cholesky factorization. Parallelism is hidden inside calls to parallel implementations of BLAS operations, which limits the possibility of reusing communication steps between several such operations.

A_{BR} is empty. Within the `while` loop, progress through the matrix is controlled by two commands: The first of which is

```
RepartitionDownDiagonal( ATL, /**/ ATR,  A00, /**/ A01, A02,
                        /*************/ /******************/
                              /**/       A10, /**/ A11, A12,
                         ABL, /**/ ABR,  A20, /**/ A21, A22 );
```

which refines the four quadrants into nine submatrices, of which A_{11} is a square diagonal block whose dimension is equal to the algorithmic block size. The second command is

```
SlidePartitionDownDiagonal( ATL, /**/ ATR,  A00, A01, /**/ A02,
                                 /**/       A10, A11, /**/ A12,
                           /*************/ /******************/
                            ABL, /**/ ABR,  A20, A21, /**/ A22 );
```

which redefines the four quadrants such that A_{TL}, the completely factored portion of A, has been augmented by the A_{01}, A_{10}, and A_{11} submatrices.

```
template<typename F> // F is any representation of a real or complex field
void CholUVar3( DistMatrix<F,MC,MR>& A )
{
    const Grid& g = A.Grid();
    DistMatrix<F,MC,MR> ATL(g), ATR(g),  A00(g), A01(g), A02(g),
                        ABL(g), ABR(g),  A10(g), A11(g), A12(g),
                                         A20(g), A21(g), A22(g);
    DistMatrix<F,STAR,STAR> A11_STAR_STAR(g);
    DistMatrix<F,STAR,VR  > A12_STAR_VR(g);
    DistMatrix<F,STAR,MC  > A12_STAR_MC(g);
    DistMatrix<F,STAR,MR  > A12_STAR_MR(g);

    PartitionDownDiagonal( A, ATL, ATR,
                              ABL, ABR, 0 );
    while( ABR.Height() > 0 )
    {
        RepartitionDownDiagonal( ATL, /**/ ATR,  A00, /**/ A01, A02,
                                /*************/ /*****************/
                                      /**/       A10, /**/ A11, A12,
                                 ABL, /**/ ABR,  A20, /**/ A21, A22 );

        A12_STAR_MC.AlignWith( A22 );
        A12_STAR_MR.AlignWith( A22 );
        A12_STAR_VR.AlignWith( A22 );
        //------------------------------------------------------------------//
        A11_STAR_STAR = A11;
        internal::LocalChol( UPPER, A11_STAR_STAR );
        A11 = A11_STAR_STAR;

        A12_STAR_VR = A12;
        internal::LocalTrsm
        ( LEFT, UPPER, ADJOINT, NON_UNIT, (F)1, A11_STAR_STAR, A12_STAR_VR );

        A12_STAR_MC = A12_STAR_VR;
        A12_STAR_MR = A12_STAR_VR;
        A12 = A12_STAR_MR;
        internal::LocalTrrk
        ( UPPER, ADJOINT, (F)-1, A12_STAR_MC, A12_STAR_MR, (F)1, A22 );
        //------------------------------------------------------------------//
        A12_STAR_MC.FreeAlignments();
        A12_STAR_MR.FreeAlignments();
        A12_STAR_VR.FreeAlignments();

        SlidePartitionDownDiagonal( ATL, /**/ ATR,  A00, A01, /**/ A02,
                                         /**/       A10, A11, /**/ A12,
                                   /*************/ /*****************/
                                    ABL, /**/ ABR,  A20, A21, /**/ A22 );
    }
}
```

Fig. 3. Elemental upper-triangular Variant 3 Cholesky factorization. Unlike the ScaLAPACK code, this one code can accomodate any datatype that represents a real or complex field.

All communication and floating point computation occurs between the two horizontal lines of hyphens. Within this section, submatrices of A are redistributed, locally updated, and then redistributed back to their original form. For instance, redistributing A_{11} so that all processes have a copy is achieved by first creating the container

```
    DistMatrix<F,STAR,STAR> A11_STAR_STAR(g);
```

which indicates that `A11_STAR_STAR` is to be used for holding matrices which are replicated on all processes. The lines

```
        A11_STAR_STAR = A11;
        internal::LocalChol( UPPER, A11_STAR_STAR );
        A11 = A11_STAR_STAR;
```

then redistribute $A11$ by performing an allgather of the data, redundantly factor each local copy of A_{11}, and then locally (without communication) substitute the new values back into the distributed matrix.

The parallel computation of $A_{12} := A_{11}^{-H} A_{12}$ is accomplished by constructing a container for A_{12},

```
    DistMatrix<F,STAR,VR> A12_STAR_VR(g);
```

which describes what in PLAPACK would have been called a multivector distribution, followed by

```
        A12_STAR_VR = A12;
        internal::LocalTrsm
        ( LEFT, UPPER, ADJOINT, NON_UNIT, (F)1, A11_STAR_STAR, A12_STAR_VR );
```

which redistributes the data via an all-to-all communication within columns and performs the local portion of the update $A_{12} := A_{11}^{-H} A_{12}$ (TRSM). The subsequent redistributions of A_{12}, so that $A_{22} := A_{22} - A_{12}^{H} A_{12}$ can be performed through local computation, are accomplished by first constructing two temporary distributions,

```
    DistMatrix<F,STAR,MC> A12_STAR_MC(g);
    DistMatrix<F,STAR,MR> A12_STAR_MR(g);
```

and then the redistributions themselves are accomplished by the commands

```
        A12_STAR_MC = A12_STAR_VR;
        A12_STAR_MR = A12_STAR_VR;
```

The first command performs a permutation of data among all processes followed by an allgather of data within rows, and the second performs an allgather of data within columns. The local updates of A_{22} are then accomplished by

```
        internal::LocalTrrk
        ( UPPER, ADJOINT, (F)-1, A12_STAR_MC, A12_STAR_MR, (F)1, A22 );
```

Finally, the modified A_{12} resulting from the `LocalTrsm` is redistributed into its original form (without communication) by the command

```
        A12 = A12_STAR_MR;
```

This example shows that the Elemental framework allows the partitioning, distributions, communications, and local computations to be elegantly captured in code. The distributions and data movements that are incurred are further illustrated in Appendix A.

4. PERFORMANCE EXPERIMENTS

The scientific computing community has always been willing to accept complexity if it means attaining better performance. In this section, we give preliminary performance numbers that suggest that a focus on abstraction and elegance does not need to come at the cost of performance.

4.1. Platform details

The performance experiments were carried out on Argonne National Laboratory's IBM Blue Gene/P architecture. Each compute node consists of four 850 MHz PowerPC 450 processors for a combined theoretical peak performance of 13.6 GFlops in double-precision arithmetic per node. Nodes are interconnected by a three-dimensional torus topology and a collective network that each support a per node bidirectional bandwidth of 2.25 GB/s. Our experiments were performed on two racks (2048 compute nodes, or 8192 cores), which have an aggregate theoretical peak of 27.85 TFlops. For this configuration the X, Y, and Z dimensions of the torus are 8, 8, and 32, respectively, while the intranode dimension, T, is 4. For the majority of our experiments, the optimal decomposition into a two-dimensional topology was found to be either $(X, Y) \times (Z, T)$ or $(Z, T) \times (X, Y)$, with the former case meaning that the process grid is constructed with its first dimension containing both the X and Y dimensions of the torus, and its second dimension containing both the Z and T dimensions. Avoiding irregular communicators, i.e., those that only partially span one or more torus dimensions, is crucial in achieving high bandwidths in collective communication routines on Blue Gene/P. The $(X, Y) \times (Z, T)$ decomposition leads to a 64×128 process grid, while $(Z, T) \times (X, Y)$ clearly yields the reverse.

We compare the performance of a preliminary version of Elemental with the latest release of ScaLAPACK available from `netlib` (Release 1.8). Both packages were extensively tested for block sizes between $24, 32, ..., 256$ and for various process grid sizes and mappings to Blue Gene/P's torus. For each problem size, only the result from the best-performing block size and process grid combination is reported in the performance graphs.

4.2. Operations

Solution of the Hermitian-definite generalized eigenvalue problem, given by $Ax = \lambda Bx$ where A and B are known, A is Hermitian, and B is Hermitian positive-definite, is of importance to a wide class of fields, including quantum chemistry [Ford and Hall 1974] and structural dynamics [Bennighof and Lehoucq 2003]. When A and B are dense, the Cholesky-Wilkinson algorithm performs the following six steps [Wilkinson 1965; Golub and Van Loan 1989]:

— **Cholesky factorization.**
$B \to LL^H$ where L is lower triangular.

— **Reduction to Hermitian standard form.**
Form $C := L^{-1}AL^{-H}$, since $Cz = \lambda z$ holds if and only if $Ax = \lambda(LL^H)x$, where $z \equiv L^H x$.

— **Householder reduction to tridiagonal form.**
Compute a unitary Q (as a sequence of Householder transformations) and a real tridiagonal T such that $T = QCQ^H$.

— **Spectral decomposition of a tridiagonal matrix.**
Compute a unitary V and a real diagonal matrix D (or subsets thereof) such that $T = VDV^H$. The most common approaches are based upon: the MRRR algorithm [Dhillon 1997], Cuppen's divide-and-conquer [Cuppen 1981], and the shifted QR algorithm [Stewart 1970].

— **Back transformation.**
Compute $Z := Q^H V$ by applying the Householder transformations that represent Q.

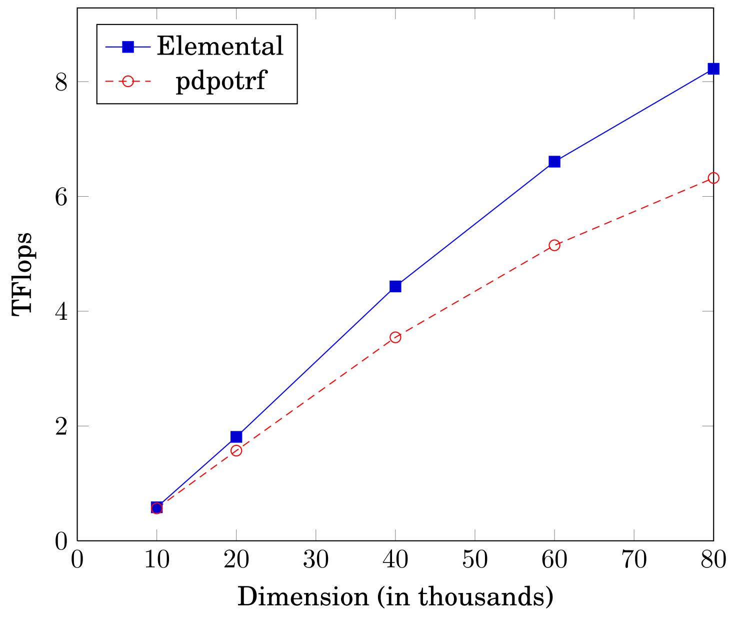

Fig. 4. Real double-precision Cholesky factorization on 8192 cores of Blue Gene/P. The top of the graph is set to one third of theoretical peak performance, 9.284 TFlops.

— **Solution of a triangular system with multiple right-hand sides.** Form the eigenvectors of A by solving $Z = L^H X$ for X.

Since the performance of the symmetric tridiagonal eigensolution is, in contrast with the other five steps, both problem and accuracy dependent, we have not included results from the extensive experiments that would be required for an honest comparison. However, we note that Elemental incorporates a scalable parallel implementation of the MRRR algorithm [Bientinesi et al. 2005a] that was implemented by Matthias Petschow and Paolo Bientinesi.

4.3. Results

Algorithms for complex matrices typically require four times as many floating point operations as their real equivalent, yet only require twice as much storage. Thus, translating a matrix algorithm from real into complex arithmetic almost universally implies an increase in the rate of floating point operations per second (flops). We thus restrict our performance experiments to the more challenging (from an efficiency standpoint) real symmetric-definite generalized eigenvalue problem.

Figure 4: Cholesky factorization. The performance in teraflops (TFlops) for both Elemental and ScaLAPACK's real Cholesky factorization routines are shown for problem sizes up to $n = 80,000$, which, in double precision, only requires roughly 6.25 MB of per process storage for the full matrix. This range of matrix sizes was chosen because applications that make use of dense eigensolvers often target problem sizes that are in the vicinity of $n = 30,000$, and extremely large problem sizes are ill-suited for dense linear algebra due to its characteristic $O(n^3)$ computational complexity.

The best performing parameters for ScaLAPACK's `pdpotrf` were found to be one of two cases: for all but the $n = 80,000$ problem size, an $(Z,T) \times (X,Y)$ process grid configuration was the fastest, yet for $n = 80,000$ an $(X,Y) \times (Z,T)$ mapping of the torus to the process grid led to nearly two teraflops of performance improvement. For

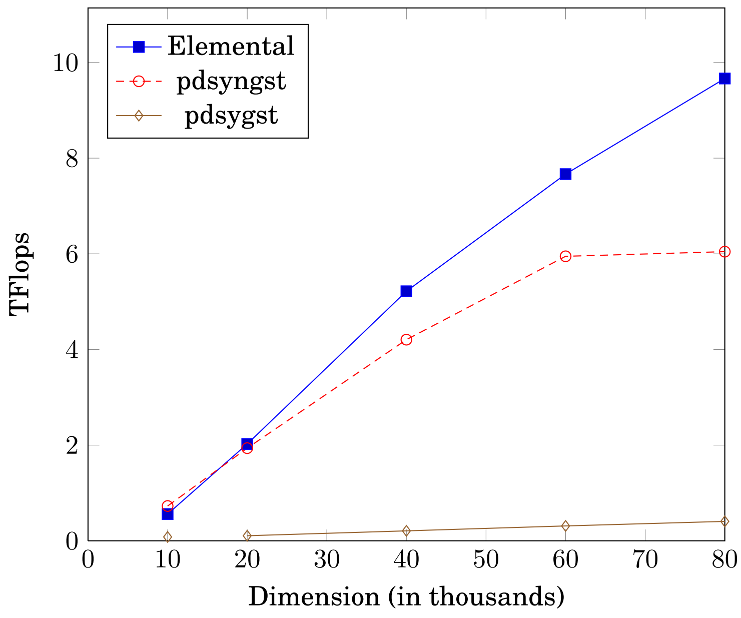

Fig. 5. Real double-precision reduction of generalized-definite eigenvalue problem $AX = BX\Lambda$ to symmetric standard form on 8192 cores. The top of the graph is set to 40% of theoretical peak performance, 11.14 TFlops.

Elemental, the best mapping of the process grid onto the torus was found to be $(Z, T) \times (X, Y)$ in all cases[3].

That Elemental achieved modest performance improvements over ScaLAPACK's `pdpotrf` agrees with our analysis that the communication costs between the two different approaches are approximately equal, but, on the other hand, ScaLAPACK has (perhaps suboptimal) hardcoded logical block sizes for the Hermitian rank-k update of A_{22} that dominates the computational cost of right-looking Cholesky factorization.

Figure 5: Reduction to Hermitian standard form. ScaLAPACK includes two implementations of this operation:

— `pdsygst`, which is a parallelization of the algorithm used by LAPACK's `dsygst`. This algorithm casts more computation in terms of `dtrsm` (triangular solve with multiple right-hand sides, but with only a few right-hand sides), which greatly limits the opportunity for parallelism.
— `pdsyngst`, which is based on an algorithm described in [Sears et al. 1998]. It casts most computation in terms of rank-k updates, which are easily parallelizable[4].

For the Elemental implementation, we independently derived an algorithm that is similar to the one used in `pdsyngst`, as described in our paper [Poulson et al. 2011].

[3]Since Cholesky factorization requires a relatively small amount of work for a dense linear algebra routine, its performance is particularly sensitive to the achieved communication speeds, hence the discussed teraflops of difference in performance due to different process grid choices. Indeed, experiments with Elemental revealed a bug in the Blue Gene/P-specific modifications of MPICH2's `MPI_Allgather` that, when corrected, led to twenty-times faster allgathers within rows of the process grid and increased performance by several teraflops. The logic for routing between different allgather algorithms implicitly assumed that each entry of data required one byte of storage. Fixing this mistake only requires trivial modifications to the source code. Alternatively, one can implicitly cast all data to a byte-sized datatype, e.g., `MPI_CHAR`.
[4]We note that `pdsyngst` only supports lower-triangular matrices.

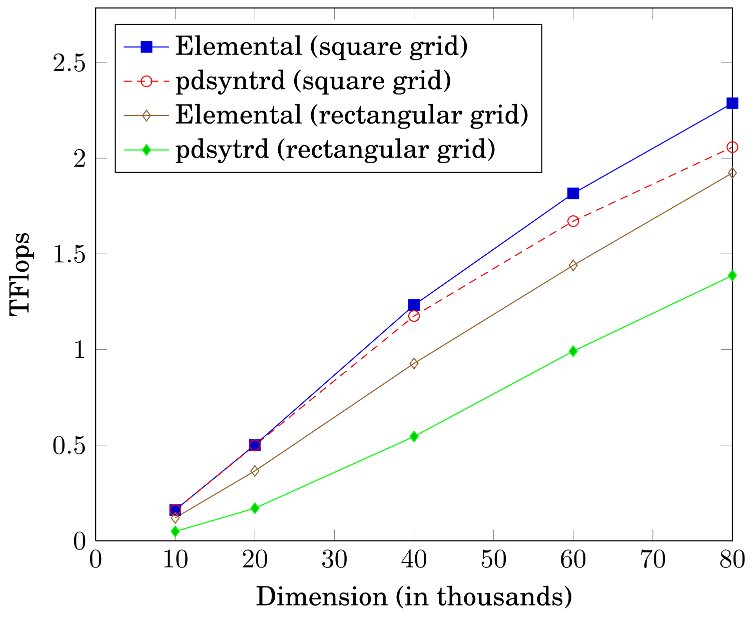

Fig. 6. Real double-precision Householder tridiagonalization on two racks of Blue Gene/P. The top of the graph is set to 10% of theoretical peak performance, 2.785 TFlops.

The performance results clearly demonstrate the effects of the mentioned scalability problem in `pdsygst`, which is shown to be an order of magnitude slower than the other two approaches.

Although the routine `pdsyngst` performs essentially the same sequence of updates as the Elemental implementation, it contains redundant communications since layering on top of the PBLAS does not allow for the reuse of temporary matrix distributions. Unlike Cholesky factorization, there are numerous updates to perform for each iteration of the algorithm and thus the cost of the redundant communication is more pronounced.

Figure 6: Householder reduction to tridiagonal form. As with the previous operation, ScaLAPACK includes two different implementations for the reduction to tridiagonal form:

— `pdsytrd` is a straightforward parallelization of the corresponding LAPACK algorithm, `dsytrd`, which accumulates and applies many Householder updates at a time in order to cast roughly half of the total work into matrix-matrix multiplication.

— `pdsyntrd` is based on the work discussed in [Sears et al. 1998; Hendrickson et al. 1999] and is a parallelization of a slightly different algorithm than that of `dsytrd`; in both algorithms the Householder updates are accumulated during the tridiagonalization of a column panel, but as `dsytrd` traverses through the columns of the panel, it adopts a lazy approach to updating the panel while `pdsyntrd` parallelizes a greedy algorithm for updating the panel. More importanty, rather than using a block cyclic matrix distribution, it exploits an elemental distribution in order to set up a communication pipeline within rows of the process grid. In addition, `pdsyntrd` requires a square process grid since it allows for faster data transposition and symmetric matrix vector multiplies.
Since ScaLAPACK uses block cyclic matrix distributions for every other routine, `pdsyntrd` redistributes the matrix from a rectangular process grid into an elemen-

tal distribution over a square process grid that is as large as possible, performs the fast tridiagonalization, and then redistributes back to the original block cyclic matrix distribution[5].

Elemental also contains two different approaches for the reduction to tridiagonal form, one is a direct analogue to the algorithm discussed in [Sears et al. 1998], and the other is a generalization to rectangular process grids so that the redistribution to a square process grid can be avoided. The penalty is that matrix distributions can no longer be transposed via a simple pairwise exchange and the local computation for the distributed symmetric matrix vector multiplies is slightly less efficient, but many communications can still be combined in the same manner as described in Sears et al.

The performance results demonstrate that the Elemental and ScaLAPACK implementations of the square process grid algorithm (which both use elemental distributions) achieve nearly identical performance; this is not surprising considering that the two algorithms are essentially identical. More interestingly, Elemental's generalization of this specialized approach to rectangular process grids is shown to achieve substantial speedups over the straightforward parallelization of LAPACK's `dsytrd`, `pdsytrd`. We note that both Elemental and ScaLAPACK's redistributions to a square process grid currently use roughly twice the required amount of memory since they essentially make a copy of the input matrix, though it would be possible to do an in-place redistribution of the input matrix if the associated buffers were guaranteed to be large enough to store either matrix distribution. It is thus currently preferable to use the rectangular grid algorithms when memory usage must be kept at a minimum.

The setup necessary for maximizing the tridiagonalization performance is quite different from that of the other operations in that it does not involve evenly dividing the torus dimensions between the row and column communicators: all four of the ScaLAPACK and Elemental implementations performed best with a $TXYZ$ ordering of the torus organized into either 64×128 or 128×64 process grids, both of which split the Y dimension of the torus between the row and column subcommunicators.

Figure 7: Back transformation. This operation requires the Householder transformations that form Q to be applied to the eigenvectors of the tridiagonal matrix T. ScaLAPACK uses compact WY-transforms (a transform that applies multiple Householder transformations simultaneously in order to cast computation in terms of `dgemm`). The Elemental implementation employs a variant on the compact WY-transform, the block UT transform [Joffrain et al. 2006]. This variant has the advantage that forming the block transform requires less communication, and nearly a 100% performance improvement is demonstrated for the larger problem sizes.

Figure 8: Solution of a triangular system with multiple right-hand sides. The first published parallelization of this operation is given in [Chtchelkanova et al. 1997], and Elemental's implementations closely follow the algorithms suggested therein. While ScaLAPACK incorporates the same approach, it also includes several other approaches, and which algorithm to execute is chosen at runtime based upon estimates of communication volume. That Elemental only implements a single algorithm and still achieves substantial performance improvements suggests to us that the algorithmic selection logic in the ScaLAPACK implementation is far from optimal and needlessly complex.

5. CONCLUSION

The goal of this paper is to demonstrate that parallel dense linear algebra libraries do not need to sacrifice performance in order to maintain a high level of abstraction.

[5]`pdsyntrd` only supports lower-triangular storage of the symmetric matrix.

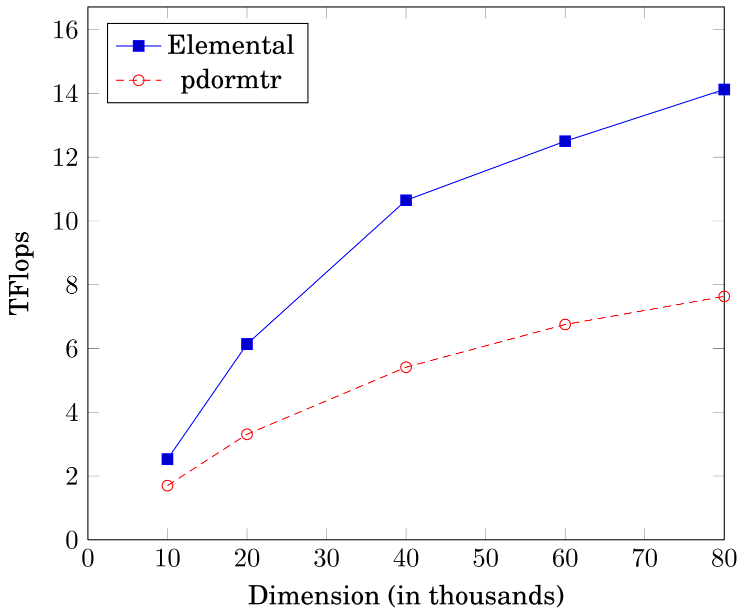

Fig. 7. Real double-precision application of the back transformation on 8192 cores of Blue Gene/P. The top of the graph is set to 60% of theoretical peak performance, 16.71 TFlops.

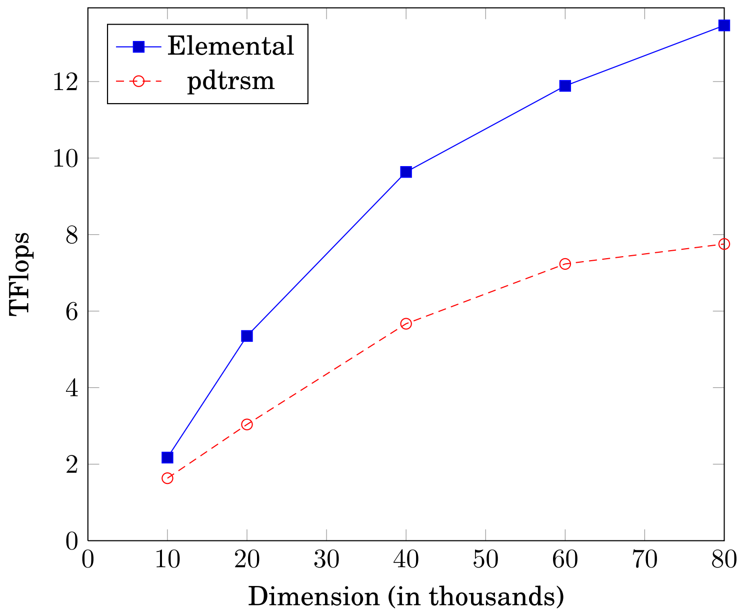

Fig. 8. Real double-precision $B := L^{-T}B$ (**trsm**) on 8192 cores of Blue Gene/P. The top of the graph is set to 50% of theoretical peak performance, 13.96 TFlops.

As of this writing, Elemental supports the following functionality, for both real and complex datatypes:

— All level-3 BLAS (and many extensions)
— Determinants, traces, and various norms
— (Semi-definite) Cholesky, LDL^T, LDL^H, LQ, LU, and QR factorization
— In-place inversion of general, HPD, and triangular matrices
— Reduction to tridiagonal and bidiagonal form
— (Skew-)Hermitian eigenvalue problems, including subset computations
— SVD, polar decompositions, and pseudoinverses
— Hermitian pseudoinverses, square roots, and matrix functions
— Hermitian generalized-definite eigenvalue problems, including subset computations

We envision a string of additional papers in the near future.

— In Appendix A, we hint at a set-based notation for describing the data distributions that underly Elemental. The key insight is that relationships between the sets that define different distributions dictate the communications that are required for the redistribution. A full paper on this topic is being written.
— A paper that discusses in detail the parallel implementation of $A := L^{-1}AL^{-H}$ and $A := L^HAL$ has been submitted for publication [Poulson et al. 2011]. That paper more clearly explains how Elemental benefits from the FLAME approach to deriving algorithms and more explicitly addresses the question of how much of the performance improvement is due to algorithm choice and how much is due to better implementation (which attribute in part to the fact that the API used to program algorithms makes it easier to implement more complex algorithms).
— As mentioned in the abstract and introduction, a major reason for creating a new distributed memory dense matrix library and framework is the arrival of many-core architectures that can be viewed as clusters on a single chip, like the SCC architecture. The Elemental library has already been ported to the SCC processor by replacing the MPI collective communication library with calls to a custom collective communication library for that architecture. Preliminary results of that experiment are reported in [Marker et al. 2011a].
— One of the goals of the FLAME project is to make the translation of sequential algorithms coded using the FLAME/C API [Bientinesi et al. 2005b; van de Geijn and Quintana-Ortí 2008], used to develop the `libflame` library [Van Zee 2009], to optimized Elemental code mechanical. Preliminary work towards this goal is reported in [Marker et al. 2011b].

Together these, and other such papers, will build a body of evidence in support of some of the less substantiated claims in this introductory paper.

Availability

The Elemental package is available under the New BSD License at `http://code.google.com/p/elemental`.

A. ELEMENTAL DISTRIBUTION AND CHOLESKY FACTORIZATION: MORE DETAILS

In this appendix, we describe the basics of distribution and redistribution in Elemental through a parallel Cholesky factorization.

$$
\begin{array}{|cccc|cccc|cccc|}
\hline
\multicolumn{4}{|c|}{\text{Process (0,0)}} & \multicolumn{4}{c|}{\text{Process (0,1)}} & \multicolumn{4}{c|}{\text{Process (0,2)}} \\
\alpha_{0,0} & \alpha_{0,3} & \alpha_{0,6} & \cdots & \alpha_{0,1} & \alpha_{0,4} & \alpha_{0,7} & \cdots & \alpha_{0,2} & \alpha_{0,5} & \alpha_{0,8} & \cdots \\
\alpha_{2,0} & \alpha_{2,3} & \alpha_{2,6} & \cdots & \alpha_{2,1} & \alpha_{2,4} & \alpha_{2,7} & \cdots & \alpha_{2,2} & \alpha_{2,5} & \alpha_{2,8} & \cdots \\
\alpha_{4,0} & \alpha_{4,3} & \alpha_{4,6} & \cdots & \alpha_{4,1} & \alpha_{4,4} & \alpha_{4,7} & \cdots & \alpha_{4,2} & \alpha_{4,5} & \alpha_{4,8} & \cdots \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots \\
\hline
\multicolumn{4}{|c|}{\text{Process (1,0)}} & \multicolumn{4}{c|}{\text{Process (1,1)}} & \multicolumn{4}{c|}{\text{Process (1,2)}} \\
\alpha_{1,0} & \alpha_{1,3} & \alpha_{1,6} & \cdots & \alpha_{1,1} & \alpha_{1,4} & \alpha_{1,7} & \cdots & \alpha_{1,2} & \alpha_{1,5} & \alpha_{1,8} & \cdots \\
\alpha_{3,0} & \alpha_{3,3} & \alpha_{3,6} & \cdots & \alpha_{3,1} & \alpha_{3,4} & \alpha_{3,7} & \cdots & \alpha_{3,2} & \alpha_{3,5} & \alpha_{3,8} & \cdots \\
\alpha_{5,0} & \alpha_{5,3} & \alpha_{5,6} & \cdots & \alpha_{5,1} & \alpha_{5,4} & \alpha_{5,7} & \cdots & \alpha_{5,2} & \alpha_{5,5} & \alpha_{5,8} & \cdots \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots \\
\hline
\end{array}
$$

Fig. 9. Distribution $A[M_C, M_R]$ when $r \times c = 2 \times 3$, $M_C^s = \{s, s+r, \ldots\}$, and $M_R^t = \{t, t+c, \ldots\}$.

A.1. Assumptions

In our discussion, we assume that the p processes form a (logical) $p = r \times c$ mesh. We let $A \in \mathbb{F}^{m \times n}$, where $\mathbb{F}$ is any field, equal

$$
A = \begin{pmatrix}
\alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\
\alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
\alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1}
\end{pmatrix}.
$$

We will use the first iteration of the Cholesky factorization algorithm discussed in the main body of the paper for illustration, with algorithmic blocks size $b_{\text{alg}} = 3$, so that

$$
\left(\begin{array}{c|c|c}
A_{00} & A_{01} & A_{02} \\
\hline
A_{10} & A_{11} & A_{12} \\
\hline
A_{20} & A_{21} & A_{22}
\end{array}\right)
=
\left(\begin{array}{c|ccc|cccc}
 & & & & & & & \\
\hline
 & \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} & \alpha_{04} & \alpha_{05} & \cdots \\
 & \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} & \alpha_{15} & \cdots \\
 & \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} & \alpha_{25} & \cdots \\
\hline
 & \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} & \alpha_{35} & \cdots \\
 & \alpha_{40} & \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} & \alpha_{45} & \cdots \\
 & \alpha_{50} & \alpha_{51} & \alpha_{52} & \alpha_{53} & \alpha_{54} & \alpha_{55} & \cdots \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}\right) \tag{1}
$$

A.2. Distribution $A[M_C, M_R]$

If $M_C = \{M_C^0, \cdots, M_C^{r-1}\}$ and $M_R = \{M_R^0, \cdots, M_R^{c-1}\}$ are partitionings of the natural numbers, then $A[M_C, M_R]$ will be used to denote the distribution of matrix A that assigns to process (s,t) the submatrix of A where the row indices and column indices are selected from M_C^s and M_R^t, respectively.

In the case of the elemental distribution, $M_C^s = \{s, s+r, s+2r, \ldots\}$ and $M_R^t = \{t, t+c, t+2c, \ldots\}$, which means that $A[M_C, M_R]$ assigns

$$
\begin{pmatrix}
\alpha_{s,t} & \alpha_{s,t+c} & \cdots \\
\alpha_{s+r,t} & \alpha_{s+r,t+c} & \cdots \\
\vdots & \vdots &
\end{pmatrix}
$$

to process (s,t) of an $r \times c$ mesh. We illustrate this in Figure 9 for $r \times c = 2 \times 3$.

A.3. Distribution $A[\star,\star]$

In our parallel Cholesky factorization, A_{11} is duplicated to all processes before being redundantly factored. We denote duplication of the entire matrix A by $A[\star,\star]$, which is meant to indicate that each process keeps all row and column indices. Duplicating only submatrix A_{11} is denoted by $A_{11}[\star,\star]$, as illustrated by

Process (0,0)	Process (0,1)	Process (0,2)
$\begin{matrix}\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2}\\ \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}\\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2}\end{matrix}$	$\begin{matrix}\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2}\\ \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}\\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2}\end{matrix}$	$\begin{matrix}\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2}\\ \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}\\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2}\end{matrix}$
Process (1,0)	**Process (1,1)**	**Process (1,2)**
$\begin{matrix}\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2}\\ \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}\\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2}\end{matrix}$	$\begin{matrix}\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2}\\ \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}\\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2}\end{matrix}$	$\begin{matrix}\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2}\\ \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}\\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2}\end{matrix}$

for the matrix in (1).

The communication that takes $A_{11}[M_C,M_R]$ to $A_{11}[\star,\star]$ is an allgather involving all processes. In our code, this is accomplished by the command

```
A11_STAR_STAR = A11_MC_MR;
```

The inverse operation that redistributes $A_{11}[\star,\star]$ back to $A_{11}[M_C,M_R]$ requires no communication, since all processes have a copy, and is accomplished by the command

```
A11_MC_MR = A11_STAR_STAR;
```

A.4. Distribution $A[\star,V_R]$

In our Cholesky factorization, we must compute $A_{12} := A_{11}^{-H}A_{12}$, where A_{11} is upper triangular part of matrix A_{11}. By the time this computation occurs, A_{11} is available as $A_{11}[\star,\star]$ (duplicated to all nodes) but A_{12} is distributed as part of A: $A_{12}[M_C,M_R]$, complicating the parallelization of this operation. The question is how to redistribute the data so that the computation can easily parallelized.

Consider our prototypical matrix in (1). If we redistribute A_{12} like

Process 0	Process 1	Process 2
$\begin{matrix}\alpha_{0,6} & \alpha_{0,12} & \cdots\\ \alpha_{1,6} & \alpha_{1,12} & \cdots\\ \alpha_{2,6} & \alpha_{2,12} & \cdots\end{matrix}$	$\begin{matrix}\alpha_{0,7} & \alpha_{0,13} & \cdots\\ \alpha_{1,7} & \alpha_{1,13} & \cdots\\ \alpha_{2,7} & \alpha_{2,13} & \cdots\end{matrix}$	$\begin{matrix}\alpha_{0,8} & \alpha_{0,14} & \cdots\\ \alpha_{1,8} & \alpha_{1,14} & \cdots\\ \alpha_{2,8} & \alpha_{2,14} & \cdots\end{matrix}$
Process 3	**Process 4**	**Process 5**
$\begin{matrix}\alpha_{0,3} & \alpha_{0,9} & \alpha_{0,15} & \cdots\\ \alpha_{1,3} & \alpha_{1,9} & \alpha_{1,15} & \cdots\\ \alpha_{2,3} & \alpha_{2,9} & \alpha_{2,15} & \cdots\end{matrix}$	$\begin{matrix}\alpha_{0,4} & \alpha_{0,10} & \alpha_{0,16} & \cdots\\ \alpha_{1,4} & \alpha_{1,10} & \alpha_{1,16} & \cdots\\ \alpha_{2,4} & \alpha_{2,10} & \alpha_{2,16} & \cdots\end{matrix}$	$\begin{matrix}\alpha_{0,5} & \alpha_{0,11} & \alpha_{0,17} & \cdots\\ \alpha_{1,5} & \alpha_{1,11} & \alpha_{1,17} & \cdots\\ \alpha_{2,5} & \alpha_{2,11} & \alpha_{2,17} & \cdots\end{matrix}$

then

$$\begin{pmatrix}\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2}\\ 0 & \alpha_{1,1} & \alpha_{1,2}\\ 0 & 0 & \alpha_{2,2}\end{pmatrix}^{-1}\begin{pmatrix}\alpha_{0,3} & \alpha_{0,4} & \cdots\\ \alpha_{1,3} & \alpha_{1,4} & \cdots\\ \alpha_{2,3} & \alpha_{2,4} & \cdots\end{pmatrix}$$

can be parallized trivially because A_{11} is already available to all processes and locally the computation

$$\begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} \\ 0 & \alpha_{1,1} & \alpha_{1,2} \\ 0 & 0 & \alpha_{2,2} \end{pmatrix}^{-1} \begin{pmatrix} \alpha_{0,k} & \alpha_{0,k+p} & \cdots \\ \alpha_{1,k} & \alpha_{1,k+p} & \cdots \\ \alpha_{2,k} & \alpha_{2,k+p} & \cdots \end{pmatrix}$$

can commense on what is now labeled as process k (except that the first three columns are skipped in this example).

We now construct a distribution that facilitates the above insight. To do so, we view the processes as a linear array indexed in row-major order so that process (s,t) in the 2D mesh is process $k = sc + t$ in our 1D view of the processes. Next, we create a partitioning of the natural numbers $\{V_R^0, \cdots, V_R^{p-1}\}$ where $V_R^k = \{k, k+p, k+2p, \ldots\}$. The distribution $A[\star, V_R]$ now assigns the columns V_R^k of matrix A to process k, viewing the processes as a 1D array. In other words, it assigns the columns V_R^{sc+t} to process (s,t), viewing the processes as a 2D array. Or, one can view this as the set that determines what part of A is assigned to process $k = sc+t$ selects all rows of A (hence the $\star$) and the columns of A in the set V_R^k.

Now, by design, for our elemental distribution $M_R^t = \cup_{s=0,r-1} V_R^{sc+t}$. What this means is that to redistribute $A_{12}[M_C, M_R]$ to $A_{12}[\star, V_R]$, elements of A_{12} need only be communicated within columns of processes, via an all-to-all collective communication. The command that redistributes A_{12} in this fashion is given by

```
    A12_STAR_VR = A12_MC_MR;
```

A.5. Distributions $A[\star, M_R]$ and $A[\star, M_C]$

Finally, let us look at the update $A_{22} := A_{22} - A_{12}^H A_{21}$. The resulting matrix overwrites A_{22} and hence should be distributed like A_{22}: $(A_{22} - A_{12}^H A_{21})[M_C, M_R]$. To make all of the computation local, we note that on process (s,t)

$$\begin{aligned} (A_{22} - A_{12}^H A_{12})[M_C^s, M_R^t] &= A_{22}[M_C^s, M_R^t] - A_{12}^H A_{12}[M_C^s, M_R^t] \\ &= A_{22}[M_C^s, M_R^t] - (A_{12}^H)[M_C^s, \star] A_{12}[\star, M_R^t] \\ &= A_{22}[M_C^s, M_R^t] - A_{12}[\star, M_C^s]^H A_{12}[\star, M_R^t]. \end{aligned}$$

These three distributions are illustrated in Figure 10.

Let us focus on process $(1,2)$ where the following update must happen as part of $A_{22} := A_{22} - A_{12}^H A_{12}$:

$$\begin{pmatrix} \alpha_{3,5} & \alpha_{3,8} & \alpha_{3,11} & \cdots \\ \alpha_{5,5} & \alpha_{5,8} & \alpha_{5,11} & \cdots \\ \alpha_{7,5} & \alpha_{7,8} & \alpha_{7,11} & \cdots \\ \alpha_{9,5} & \alpha_{9,8} & \alpha_{9,11} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} - := \begin{pmatrix} \alpha_{0,3} & \alpha_{0,5} & \alpha_{0,7} & \cdots \\ \alpha_{1,3} & \alpha_{1,5} & \alpha_{1,7} & \cdots \\ \alpha_{2,3} & \alpha_{2,5} & \alpha_{2,7} & \cdots \end{pmatrix}^H \begin{pmatrix} \alpha_{0,5} & \alpha_{0,8} & \alpha_{0,11} & \cdots \\ \alpha_{1,5} & \alpha_{1,8} & \alpha_{1,11} & \cdots \\ \alpha_{2,5} & \alpha_{2,8} & \alpha_{2,11} & \cdots \end{pmatrix}. \qquad (2)$$

(Here the gray entries are those that are not updated due to symmetry.) *If* the elements of A_{12} are distributed as illustrated in Figure 10 *then* each process could locally update its part of A_{22}.

The commands

```
    A12_STAR_MC = A12_STAR_VR;
    A12_STAR_MR = A12_STAR_VR;
```

redistribute (the updated) A_{12} as required. The second command requires an allgather within column of processes. The first requires a more complicated, but equally systematic, sequence of collective communications (namely, a permutation followed by an allgather within rows). The details go beyond the scope of this paper.

Process (0,0)	Process (0,1)	Process (0,2)
$\alpha_{0,0}$ $\alpha_{0,3}$ $\alpha_{0,6}$ $\cdots$	$\alpha_{0,1}$ $\alpha_{0,4}$ $\alpha_{0,7}$ $\cdots$	$\alpha_{0,2}$ $\alpha_{0,5}$ $\alpha_{0,8}$ $\cdots$
$\alpha_{2,0}$ $\alpha_{2,3}$ $\alpha_{2,6}$ $\cdots$	$\alpha_{2,1}$ $\alpha_{2,4}$ $\alpha_{2,7}$ $\cdots$	$\alpha_{2,2}$ $\alpha_{2,5}$ $\alpha_{2,8}$ $\cdots$
$\alpha_{4,0}$ $\alpha_{4,3}$ $\alpha_{4,6}$ $\cdots$	$\alpha_{4,1}$ $\alpha_{4,4}$ $\alpha_{4,7}$ $\cdots$	$\alpha_{4,2}$ $\alpha_{4,5}$ $\alpha_{4,8}$ $\cdots$
$\vdots$ $\vdots$ $\vdots$ $\ddots$	$\vdots$ $\vdots$ $\vdots$ $\ddots$	$\vdots$ $\vdots$ $\vdots$ $\ddots$
Process (1,0)	**Process (1,1)**	**Process (1,2)**
$\alpha_{1,0}$ $\alpha_{1,3}$ $\alpha_{1,6}$ $\cdots$	$\alpha_{1,1}$ $\alpha_{1,4}$ $\alpha_{1,7}$ $\cdots$	$\alpha_{1,2}$ $\alpha_{1,5}$ $\alpha_{1,8}$ $\cdots$
$\alpha_{3,0}$ $\alpha_{3,3}$ $\alpha_{3,6}$ $\cdots$	$\alpha_{3,1}$ $\alpha_{3,4}$ $\alpha_{3,7}$ $\cdots$	$\alpha_{3,2}$ $\alpha_{3,5}$ $\alpha_{3,8}$ $\cdots$
$\alpha_{5,0}$ $\alpha_{5,3}$ $\alpha_{5,6}$ $\cdots$	$\alpha_{5,1}$ $\alpha_{5,4}$ $\alpha_{5,7}$ $\cdots$	$\alpha_{5,2}$ $\alpha_{5,5}$ $\alpha_{5,8}$ $\cdots$
$\vdots$ $\vdots$ $\vdots$ $\ddots$	$\vdots$ $\vdots$ $\vdots$ $\ddots$	$\vdots$ $\vdots$ $\vdots$ $\ddots$

Distributions $A[M_C, M_R]$ and $A_{22}[M_C, M_R]$ (highlighted).

Process (0,0)	Process (0,1)	Process (0,2)
$\alpha_{0,4}$ $\alpha_{0,6}$ $\alpha_{0,8}$ $\cdots$	$\alpha_{0,4}$ $\alpha_{0,6}$ $\alpha_{0,8}$ $\cdots$	$\alpha_{0,4}$ $\alpha_{0,6}$ $\alpha_{0,8}$ $\cdots$
$\alpha_{1,4}$ $\alpha_{1,6}$ $\alpha_{1,8}$ $\cdots$	$\alpha_{1,4}$ $\alpha_{1,6}$ $\alpha_{1,8}$ $\cdots$	$\alpha_{1,4}$ $\alpha_{1,6}$ $\alpha_{1,8}$ $\cdots$
$\alpha_{2,4}$ $\alpha_{2,6}$ $\alpha_{2,8}$ $\cdots$	$\alpha_{2,4}$ $\alpha_{2,6}$ $\alpha_{2,8}$ $\cdots$	$\alpha_{2,4}$ $\alpha_{2,6}$ $\alpha_{2,8}$ $\cdots$
Process (1,0)	**Process (1,1)**	**Process (1,2)**
$\alpha_{0,3}$ $\alpha_{0,5}$ $\alpha_{0,7}$ $\cdots$	$\alpha_{0,3}$ $\alpha_{0,5}$ $\alpha_{0,7}$ $\cdots$	$\alpha_{0,3}$ $\alpha_{0,5}$ $\alpha_{0,7}$ $\cdots$
$\alpha_{1,3}$ $\alpha_{1,5}$ $\alpha_{1,7}$ $\cdots$	$\alpha_{1,3}$ $\alpha_{1,5}$ $\alpha_{1,7}$ $\cdots$	$\alpha_{1,3}$ $\alpha_{1,5}$ $\alpha_{1,7}$ $\cdots$
$\alpha_{2,3}$ $\alpha_{2,5}$ $\alpha_{2,7}$ $\cdots$	$\alpha_{2,3}$ $\alpha_{2,5}$ $\alpha_{2,7}$ $\cdots$	$\alpha_{2,3}$ $\alpha_{2,5}$ $\alpha_{2,7}$ $\cdots$

Distribution $A_{12}[\star, M_C]$.

Process (0,0)	Process (0,1)	Process (0,2)
$\alpha_{0,3}$ $\alpha_{0,6}$ $\alpha_{0,9}$ $\cdots$	$\alpha_{0,4}$ $\alpha_{0,7}$ $\alpha_{0,10}$ $\cdots$	$\alpha_{0,5}$ $\alpha_{0,8}$ $\alpha_{0,11}$ $\cdots$
$\alpha_{1,3}$ $\alpha_{1,6}$ $\alpha_{1,9}$ $\cdots$	$\alpha_{1,4}$ $\alpha_{1,7}$ $\alpha_{1,10}$ $\cdots$	$\alpha_{1,5}$ $\alpha_{1,8}$ $\alpha_{1,11}$ $\cdots$
$\alpha_{2,3}$ $\alpha_{2,6}$ $\alpha_{2,9}$ $\cdots$	$\alpha_{2,4}$ $\alpha_{2,7}$ $\alpha_{2,10}$ $\cdots$	$\alpha_{2,5}$ $\alpha_{2,8}$ $\alpha_{2,11}$ $\cdots$
Process (1,0)	**Process (1,1)**	**Process (1,2)**
$\alpha_{0,3}$ $\alpha_{0,6}$ $\alpha_{0,9}$ $\cdots$	$\alpha_{0,4}$ $\alpha_{0,7}$ $\alpha_{0,10}$ $\cdots$	$\alpha_{0,5}$ $\alpha_{0,8}$ $\alpha_{0,11}$ $\cdots$
$\alpha_{1,3}$ $\alpha_{1,6}$ $\alpha_{1,9}$ $\cdots$	$\alpha_{1,4}$ $\alpha_{1,7}$ $\alpha_{1,10}$ $\cdots$	$\alpha_{1,5}$ $\alpha_{1,8}$ $\alpha_{1,11}$ $\cdots$
$\alpha_{2,3}$ $\alpha_{2,6}$ $\alpha_{2,9}$ $\cdots$	$\alpha_{2,4}$ $\alpha_{2,7}$ $\alpha_{2,10}$ $\cdots$	$\alpha_{2,5}$ $\alpha_{2,8}$ $\alpha_{2,11}$ $\cdots$

Distribution $A_{12}[\star, M_R]$.

Fig. 10. The distribution $A_{22}[M_C, M_R]$ (top), $A_{12}[\star, M_C]$ (middle) and $A_{12}[\star, M_R]$ (bottom) so that locally on each process $A_{22} := A_{22} - A_{12}^H A_{12}$ can be computed. For example, process $(1, 2)$ can then update its local elements as indicated in Equation (2).

In the Cholesky factorization, due to the fact that $A_{12}[\star, M_R]$ gives each process a superset of the data required for $A_{12}[M_C, M_R]$, the updated values of A_{12} can be stored using the command

```
A12_MC_MR = A12_STAR_MR;
```

which simply performs local copies.

Acknowledgements

This research was partially sponsored by NSF grants OCI-0850750 and CCF-0917167, grants from Microsoft, an unrestricted grant from Intel, and a fellowship from the Institute of Computational Engineering and Sciences. Jack Poulson was also partially supported by a fellowship from the Institute of Computational Engineering and Sciences, and Bryan Marker was partially supported by a Sandia National Laboratory fellowship. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357; early experiments were performed on the Texas Advanced Computing Center's Ranger Supercomputer.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

We would like to thank John Lewis (Cray) and John Gunnels (IBM T.J. Watson Research Center) for their constructive comments on this work and Brian Smith (IBM Rochester) for his help in avoiding performance problems in Blue Gene/P's collective communication library.

REFERENCES

ALPATOV, P., BAKER, G., EDWARDS, C., GUNNELS, J., MORROW, G., OVERFELT, J., VAN DE GEIJN, R., AND WU, Y.-J. J. 1997. PLAPACK: Parallel Linear Algebra Package – design overview. In *Proceedings of SC97*.

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, L. S., DEMMEL, J., DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., GREENBAUM, A., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK Users' Gide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

ANDERSON, E., BENZONI, A., DONGARRA, J., MOULTON, S., OSTROUCHOV, S., TOURANCHEAU, B., AND VAN DE GEIJN, R. 1992. LAPACK for distributed memory architectures: Progress report. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, 625–630.

BENNIGHOF, J. K. AND LEHOUCQ, R. 2003. An automated multilevel substructuring method for eigenspace computation in linear elastodynamics. *SIAM. J. Sci. Comput 25*, 2004.

BIENTINESI, P., DHILLON, I. S., AND VAN DE GEIJN, R. A. 2005a. A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. *SIAM Journal on Scientific Computing 27,* 1, 43–66.

BIENTINESI, P., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005b. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft. 31,* 1, 27–59.

BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. 1997. *ScaLAPACK Users' Guide*. SIAM.

CHAN, E., HEIMLICH, M., PURKAYASTHA, A., AND VAN DE GEIJN, R. 2007a. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience 19,* 13, 1749–1783.

CHAN, E., QUINTANA-ORTÍ, E., QUINTANA-ORTÍ, G., AND VAN DE GEIJN, R. 2007b. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*. 116–126.

CHOI, J., DONGARRA, J. J., OSTROUCHOV, L. S., PETITET, A. P., WALKER, D. W., AND WHALEY, R. C. 1994. The design and implementation of the ScaLAPACK LU, QR and Cholesky factorization routines. LAPACK Working Note 80 UT-CS-94-246, University of Tennessee. Sept.

CHTCHELKANOVA, A., GUNNELS, J., MORROW, G., OVERFELT, J., AND VAN DE GEIJN, R. A. 1997. Parallel implementation of BLAS: General techniques for level 3 BLAS. *Concurrency: Practice and Experience 9,* 9, 837–857.

CUPPEN, J. J. M. 1981. A divide and conquer method for the symmetric tridiagonal eigenvalue problem. *Numer. Math. 36*, 177–195.

DHILLON, I. S. 1997. A new $O(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem. Ph.D. thesis, EECS Department, University of California, Berkeley.

DONGARRA, J. AND OSTROUCHOV, S. 1990. LAPACK block factorization algorithms on the Intel iPSC/860. LAPACK Working Note 24, Technical Report CS-90-115, University of Tennessee. Oct.

DONGARRA, J. AND VAN DE GEIJN, R. 1992. Reduction to condensed form on distributed memory architectures. *Parallel Computing 18*, 973–982.

DONGARRA, J., VAN DE GEIJN, R., AND WALKER, D. 1994. Scalability issues affecting the design of a dense linear algebra library. *J. Parallel Distrib. Comput. 22,* 3.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND DUFF, I. 1990. A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft. 16,* 1, 1–17.

EDWARDS, C., GENG, P., PATRA, A., AND VAN DE GEIJN, R. 1995. Parallel matrix distributions: have we been doing it all wrong? Tech. Rep. TR-95-40, Department of Computer Sciences, The University of Texas at Austin.

FORD, B. AND HALL, G. 1974. The generalized eigenvalue problem in quantum chemistry. *Computer Physics Communications 8,* 5, 337 – 348.

GOLUB, G. H. AND VAN LOAN, C. F. 1989. *Matrix Computations* 2nd Ed. The Johns Hopkins University Press, Baltimore.

GOTO, K. AND VAN DE GEIJN, R. A. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft. 34,* 3: Article 12, 25 pages.

GUNNELS, J. A., GUSTAVSON, F. G., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software 27,* 4, 422–455.

HENDRICKSON, B., JESSUP, E., AND SMITH, C. 1999. Toward an efficient parallel eigensolver for dense symmetric matrices. *SIAM J. Sci. Comput. 20,* 3, 1132–1154.

HENDRICKSON, B. A. AND WOMBLE, D. E. 1994. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput. 15,* 5, 1201–1226.

HOWARD, J., DIGHE, S., HOSKOTE, Y., VANGAL, S., FINAN, D., RUHL, G., JENKINS, D., WILSON, H., BORKAR, N., SCHROM, G., PAILET, F., JAIN, S., JACOB, T., YADA, S., MARELLA, S., SALIHUNDAM, P., ERRAGUNTLA, V., KONOW, M., RIEPEN, M., DROEGE, G., LINDEMANN, J., GRIES, M., APEL, T., HENRISS, K., LUND-LARSEN, T., STEIBL, S., BORKAR, S., DE1, V., WIJNGAART, R. V. D., AND MATTSON, T. 2010. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the International Solid-State Circuits Conference*.

JOFFRAIN, T., LOW, T. M., QUINTANA-ORTÍ, E. S., VAN DE GEIJN, R., AND VAN ZEE, F. G. 2006. Accumulating Householder transformations, revisited. *ACM Trans. Math. Softw. 32,* 2, 169–179.

JOHNSSON, S. L. 1987. Communication efficient basic linear algebra computations on hypercube architectures. *J. of Par. Distr. Comput. 4*, 133–172.

MARKER, B., CHAN, E., POULSON, J., VAN DE GEIJN, R., VAN DER WIJNGAART, R. F., MATTSON, T. G., AND KUBASKA, T. E. 2011a. Programming many-core architectures - a case study: Dense matrix computations on the intel scc processor. *Concurrency and Computation: Practice and Experience*. To appear.

MARKER, B., TERREL, A., POULSON, J., BATORY, D., AND VAN DE GEIJN, R. 2011b. Mechanizing the expert dense linear algebra developer. FLAME Working Note #58 TR-11-18, The University of Texas at Austin, Department of Computer Sciences. April.

MATTSON, T. G., VAN DER WIJNGAART, R., AND FRUMKIN, M. 2008. Programming the Intel 80-core network-on-a-chip terascale processor. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, Piscataway, NJ, USA, 1–11.

PETITET, A., WHALEY, R. C., DONGARRA, J., AND CLEARY, A. HPL Algorithm. `http://netlib.org/benchmark/hpl/algorithm.html`.

POULSON, J., VAN DE GEIJN, R., AND BENNIGHOF, J. 2011. Parallel algorithms for reducing the generalized hermitian-definite eigenvalue problem. FLAME Working Note #56. Technical Report TR-11-05, The University of Texas at Austin, Department of Computer Sciences. February.

QUINTANA-ORTÍ, G., QUINTANA-ORTÍ, E. S., VAN DE GEIJN, R. A., VAN ZEE, F. G., AND CHAN, E. 2009. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software 36,* 3, 14:1–14:26.

ScaLAPACK 2010. Home Page. `http://www.netlib.org/scalapack/scalapack_home.html`.

SCHREIBER, R. 1992. Scalability of sparse direct solvers. *Graph Theory and Sparse Matrix Computations 56*.

SEARS, M. P., STANLEY, K., AND HENRY, G. 1998. Application of a high performance parallel eigensolver to electronic structure calculations. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. Supercomputing '98. IEEE Computer Society, Washington, DC, USA, 1–1.

STEWART, G. 1990. Communication and matrix computations on large message passing systems. *Parallel Computing 16*, 27–40.

STEWART, G. W. 1970. Incorporating origin shifts into the qr algorithm for symmetric tridiagonal matrices. *Commun. ACM 13*, 365–367.

STRAZDINS, P. E. 1998. Optimal load balancing techniques for block-cyclic decompositions for matrix factorization. In *Proceedings of PDCN'98 2nd International Conference on Parallel and Distributed Computing and Networks*.

van de Geijn, R. Feb. 24–28, 1992. Dense linear solve on the Intel touchstone delta system. In *Digest of Papers: CompCon92, 37th IEEE Computer Society International Conference*.

van de Geijn, R. A. 1997. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press.

van de Geijn, R. A. and Quintana-Ortí, E. S. 2008. *The Science of Programming Matrix Computations*. `http://www.lulu.com/content/1911788`.

Van Zee, F. G. 2009. `libflame`*: The Complete Reference*. `www.lulu.com`.

Whaley, R. C. and Dongarra, J. J. 1998. Automatically tuned linear algebra software. In *Proceedings of SC'98*.

Wilkinson, J. H. 1965. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, England.

Wu, Y.-J. J., Alpatov, P. A., Bischof, C., and van de Geijn, R. A. 1996. A parallel implementation of symmetric band reduction using PLAPACK. In *Proceedings of Scalable Parallel Library Conference, Mississippi State University*. PRISM Working Note 35.

Received Received Month Year; revised revised Month Year; accepted accepted Month Year