Elemental: A New Framework for Distributed Memory Dense Matrix Computations

Jack Poulson[*]

Bryan Marker[†]

Jeff R. Hammond[‡]

Nichols A. Romero[§]

Robert A. van de Geijn[¶]

June 11, 2010
Revised Jan. 24, 2011

Abstract

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 it will likely not be practical to use MPI-based implementations. 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 (i.e., on two racks of Blue Gene/P).

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 [5, 3, 8, 3, 9] and PLAPACK [29, 1, 26], were created in the mid-1990s, both of which provide a substantial part of the functionality offered by the widely used LAPACK library [2]. 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 primarily architect of PLAPACK. This second package resulted from a desire to solve the programmability crisis that faced computational scientists in the early days of massively parallel computing much like the programmability problem that now faces us as multicore architectures evolve into many-core 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 [13], `libflame` library [27], and SuperMatrix runtime system for scheduling dense linear algebra algorithms to multicore architectures [6, 21]. 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

---

[*]Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712, `jack.poulson@gmail.com`.

[†]Department of Computer Science, The University of Texas at Austin, Austin, TX 78712, `bamarker@gmail.com`.

[‡]Argonne National Laboratory, 9700 South Cass Avenue, LCF/Building 240, Argonne, IL 60439, `jhammond@anl.gov`.

[§]Argonne National Laboratory, 9700 South Cass Avenue, LCF/Building 240, Argonne, IL 60439, `naromero@anl.gov`.

[¶]Department of Computer Science, The University of Texas at Austin, Austin, TX 78712, `rvdg@cs.utexas.edu`.
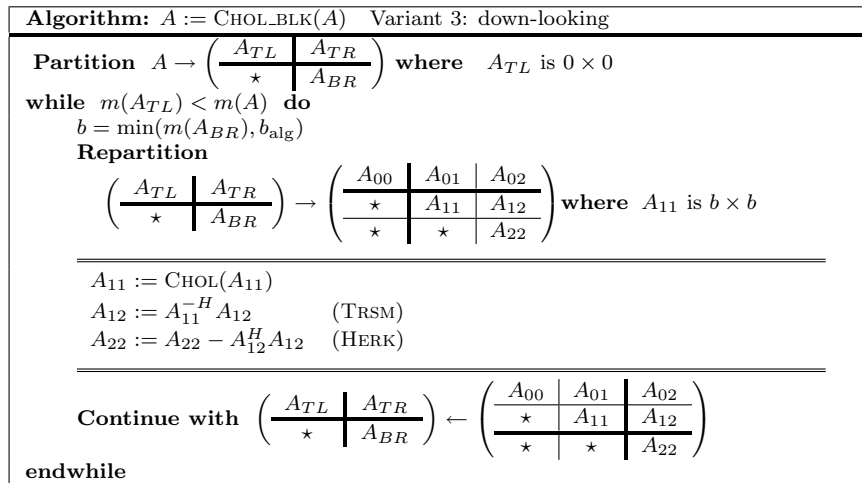
Algorithm: $A := \text{CHOL\_BLK}(A)$ Variant 3: down-looking

$$\textbf{Partition}\quad A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \quad \textbf{where} \quad A_{TL} \text{ is } 0 \times 0$$

while $m(A_{TL}) < m(A)$ **do**

    $b = \min(m(A_{BR}), b_{\text{alg}})$

    Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right) \textbf{where} \quad A_{11} \text{ is } b \times b$$

    $A_{11} := \text{CHOL}(A_{11})$

    $A_{12} := A_{11}^{-H} A_{12}$ (TRSM)

    $A_{22} := A_{22} - A_{12}^{H} A_{12}$ (HERK)

$$\textbf{Continue with}\quad \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$$

endwhile

Figure 1: Blocked algorithms for computing the Cholesky factorization.

research processor [19] and the recently announced Intel Single-chip Cloud Computer (SCC) research processor with 48 cores in one processor [16], the research comes full circle: distributed memory libraries may need to be mapped to single-chip environments.

    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. 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. It is structured as follows: in Section 2, we review how matrices are distributed to the memories of a distributed memory architecture using two-dimensional cyclic data distribution as well as the communications that are inherently encountered in parallel dense matrix computations. In Section 3, we discuss how distributed memory code can be written so as to hide many of the indexing details that traditionally make libraries for distributed memory difficult to develop and maintain. In Section 4, we show that elegance does not mean that performance must be sacrificed. Concluding remarks follow in the final section. Two appendices are included. One describes the data distributions that underly Elemental in more detail while the other illustrates the parallel Cholesky factorization that is the model example used throughout the paper.

2 Distribution 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 [23, 24, 14]. The p processes in a distributed memory architecture are logically viewed as a two-dimensional $r \times c$ mesh with $p = rc$. 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.

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 given in Figure 1.

2.2 Two-dimensional (block) cyclic distribution

Matrix $A \in \mathbb{R}^{m \times n}$ is partitioned 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 $A_{i,j}$ is of a chosen (uniform) block size. A two-dimensional (Cartesian) block-cyclic matrix distribution assigns

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

to process (s, t).

2.3 ScaLAPACK

While in theory ScaLAPACK allows blocks $A_{i,j}$ to be rectangular, in practice they are chosen to be square. The design decisions that underly ScaLAPACK link the distribution block size, b_{distr}, to the algorithmic block size, b_{alg} (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) [10] efficiency, yet small enough to avoid inefficiency due to load-imbalance.

The benefit of linking the two is that, for example, the A_{11} block in Figure 1 is owned by a single process allowing it to be factored by one process. After this factoring, it only needs to be broadcast within the row of processes that owns A_{12}, and then those processes can independently perform their part of $A_{12} := U_{11}^{-H} A_{12}$ with local calls to `trsm` (meaning only c processes participate in this operations). Finally, A_{12} is duplicated within rows and columns of processes after which A_{22} can be updated independently by each process. For some operations, the simplicity of the distribution also allows for limited overlapping of communication with computation.

2.4 PLAPACK

In PLAPACK there is a notion of a vector distribution that induces the matrix distribution [11]. Vectors are subdivided into subvectors of length b_{distr} which are wrapped in a cyclic fashion to all processes. This vector distribution then induces the distribution of columns and rows of a matrix to the mesh. The net effect is that the submatrices $A_{i,j}$ are of size b_{distr} by $r \cdot b_{\text{distr}}$. Algorithms that operate with the matrix are, as a result of this "unbalanced" distribution, mildly nonscalable in the sense that if the number of processes p gets large enough, efficiency will start to suffer even if the matrix is chosen to fill all of available memory. In PLAPACK the distribution block size can be chosen to be small while the algorithmic block size can equal the block size that makes local computation efficient since the two block sizes are not linked.

The mild nonscalability of PLAPACK was the result of a conscious choice made to simplify the implementation at a time when the number of processes was relatively small. There was always the intention to fix this eventually. The new package described in this paper is that fix, but it also incorporates other insights made in the last decade.

2.5 Elemental

In principle Elemental, like ScaLAPACK, can accomodate any distribution block size. Unlike ScaLAPACK and like PLAPACK, the distribution block size is not linked to the algorithmic block size. Load balance is optimal when the distribution block size is as small as possible, leading to the choice to initially (and possibly permanently) only support $b_{\text{distr}} = 1$, unlike PLAPACK which is not implemented to be efficient when $b_{\text{distr}} = 1$. This choice also greatly simplifies routines that pack and unpack messages before and after communication, since $b_{\text{distr}} \neq 1$ requires careful attention to be paid to partial blocks, etc.

The insight to use $b_{\text{distr}} = 1$ is not new. On early distributed memory architectures, before the advent of cache-based processors that favor blocked algorithms like the one in Figure 1, such an "elemental" distribution was the norm [18, 14]. In [15] 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 [25] it was experimentally shown that small distribution block sizes that were independent of the algorithmic block size were beneficial. This may have been prophetic but has not become particularly relevant until recently. The reason is that the algorithmic block size used to be related to the (square root of the) size of the L1 cache [28], which was relatively small. Kazushige Goto [12] 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 L2 cache. However, by the time this was discovered distributed memory architectures had so much local memory that load balance could still be achieved for the very large problem sizes that could be stored. More recently, the advent of GPU accelerators push the block size higher yet, into the $b_{\text{alg}} = 1000$ range, so that $b_{\text{distr}} = b_{\text{alg}}$ will likely become problematic. Moreover, one path towards many-core (hundreds or even thousands of cores on one chip) is to create distributed memory architectures on a chip [16]. In that scenario, the problem size will likely not be huge due to an inability to have very large memories close to the chip and/or because the problems that will be targeted to those kinds of processors will be relatively small.

To some the choice of $b_{\text{distr}} = 1$ may seem to be in contradition to conventional wisdom that says that the more processor 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:

- 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 $b_{\text{distr}} = 1$, then A_{11} can be gathered to all processes and factored redundantly. 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 done correctly, an allgather to all processes is comparable in cost to the broadcast of A_{11} performed by ScaLAPACK. (If the process mesh is $p = r \times c$, under reasonable assumptions, the former requires $\log_2(p)$ relatively short messages while the latter requires $\log_2(r)$ such messages.)

The point is that, for the suboperation that factors A_{11}, there is a small price to be paid for switching to an elemental distribution. 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 $b_{\text{distr}} = 1$, then rows of A_{12} must be brought together so that they can be updated as part of $A_{12} := U_{11}^{-H} A_{12}$. This can be implemented as an all-to-all collective communication within columns (details of which are illustrated in Appendix B). After this, since A_{11} was redundantly factored by each process, the update $A_{12} := U_{11}^{-H} A_{12}$ is shared among all p processes (again, details are illustrated in Appendix B).

Finally, consider the update of A_{22}:

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

The point is that an elemental distribution requires different communications that are comparable in cost to those incurred by ScaLAPACK while enhancing load-balance for operations like $A_{12} := U_{11}^{-H} A_{12}$ and $A_{22} := A_{22} - A_{12}^H A_{12}$.

```
          SUBROUTINE PZPOTRF( UPLO, N, A, IA, JA, DESCA, INFO )
*
*  -- ScaLAPACK routine (version 1.7) --
*     University of Tennessee, Knoxville, Oak Ridge National Laboratory,
*     and University of California, Berkeley.
*     May 25, 2001
*
          < 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 >
```

Figure 2: Excerpt from ScaLAPACK Cholesky factorization. Parallelism is hidden inside calls to parallel implementations of BLAS operations, which limits the possiblity of combining communication required for individual such operations. (The header refers to ScaLAPACK 1.7 but this excerpt is from the code in the latest release, ScaLAPACK 1.8. dated April 5 2007.)

3 Programmability

A major concern when designing Elemental was that the same code should support distributed memory parallelism on both large-scale clusters and for many cores on a single chip. Thus, the software must be flexibly retargetable to both of these extremes.

3.1 ScaLAPACK

The fundamental design decision behind ScaLAPACK can be found on the ScaLAPACK webpage [22]:

> "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 (PBLAS) of the Level 1, 2 and 3 Basic Linear Algebra Subprograms (BLAS), 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 $b_{\text{alg}} \neq b_{\text{distr}}$. While $b_{\text{distr}} = 1$ is supported, the communication layer of PLAPACK would need to be rewritten and the nonscalability of the underlying distribution would need to be fixed.

Since the inception of PLAPACK, additional insights into solutions to the programmability problem for dense matrix computations 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.

3.3 Elemental

Elemental goes one step beyond `libflame` in that it is coded in C++[1]. Other than this small detail, the coding style resembles that used by `libflame`. Like its predecessors PLAPACK and `libflame`, it hides the details of matrices and vectors within objects. As a result, much of the indexing clutter that exists in LAPACK and ScaLAPACK code disappears, leading to much easier to develop and maintain code.

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

- The tracking of submatrices in Figure 1 translates to

```
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 );
        [...]

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

- Redistributing A_{11} so that all processes have a copy is achieved by

```
DistMatrix<T,Star,Star> A11_Star_Star(g);
```

which indicates that `A11_Star_Star` describes a matrix replicated on all processes, and

```
A11_Star_Star = A11;
lapack::internal::LocalChol( Upper, A11_Star_Star );
A11 = A11_Star_Star;
```

which performs an allgather of the data, has every process redundantly factor the matrix, and then locally substitutes the new values into the distributed matrix.

- The parallel computation of $A_{12} := U_{11}^{-H} A_{12}$ is accomplished by first constructing an object for holding a temporary distribution of A_{12},

```
DistMatrix<T,Star,VR> A12_Star_VR(g);
```

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

---

[1]In the future, the library will be accessible from Fortran or C via wrappers.

```cpp
template<typename T>
void CholU( DistMatrix<T,MC,MR>& A )
{
    const Grid& g = A.GetGrid();

    DistMatrix<T,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<T,Star,Star> A11_Star_Star(g);
    DistMatrix<T,Star,VR  > A12_Star_VR(g);
    DistMatrix<T,Star,MC  > A12_Star_MC(g);
    DistMatrix<T,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;
        lapack::internal::LocalChol( Upper, A11_Star_Star );
        A11 = A11_Star_Star;

        A12_Star_VR = A12;
        blas::internal::LocalTrsm
          ( Left, Upper, ConjugateTranspose, NonUnit,
            (T)1, A11_Star_Star, A12_Star_VR );

        A12_Star_MC = A12_Star_VR;
        A12_Star_MR = A12_Star_VR;
        blas::internal::LocalTriangularRankK
          ( Upper, ConjugateTranspose,
            (T)-1, A12_Star_MC, A12_Star_MR, (T)1, A22 );
        A12 = A12_Star_MR;
        //--------------------------------------------------------------//
        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 );
    }
}
```

Figure 3: Elemental upper-triangular variant 3 Cholesky factorization.

```
A12_Star_VR = A12;
blas::internal::LocalTrsm
  ( Left, Upper, ConjugateTranspose, NonUnit,
    (T)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 redistribution of A_{12} so that $A_{22} := A_{22} - A_{12}^H A_{12}$ is accomplished by first constructing two temporary distributions,

```
DistMatrix<T,Star,MC> A12_Star_MC(g);
DistMatrix<T,Star,MR> A12_Star_MR(g);
```

which describe the two distributions needed to make the update of A_{22} local. The redistributions themselves are accomplished by the commands

```
A12_Star_VR = A12;
A12_Star_MC = A12_Star_VR;
A12_Star_MR = A12_Star_VR;
```

which perform a permutation of data among all processes, an allgather of data within rows, and an allgather of data within columns, respectively. (Details of how and why these communications are performed will be given in a future, more comprehensive paper.) The local update of A_{22} is accomplished by

```
blas::internal::LocalTriangularRankK
  ( Upper, ConjugateTranspose,
    (T)-1, A12_Star_MC, A12_Star_MR,(T)1, A22 );
```

- Finally, the updated A_{12} is placed back into the distributed matrix, without requiring any communication, by the command

```
A12 = A12_Star_MR;
```

The point is that the Elemental framework allows the partitioning, distributions, communications, and local computations to be elegantly captured in code. The data movements that are incurred are further illustrated in Appendix B while an explanation of the data distributions is given in Appendix A.

4   Performance Experiments

The scientific computing community has always been willing to give up programmability if it means attaining better performance. In this section, we give preliminary performance numbers that suggest that a focus on programmability 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 2.55 GB/s. Our experiments were performed on two racks (2048 compute nodes, or 8192 cores), which have an aggregate theoretical peak of just over 27 TFlops For this configuration the X, Y, and Z dimensions of the torus are 8, 8, and 32, respectively. The optimal decomposition into a two-dimensional topology was almost always $Z \times (X, Y)$, which was used for the below experiments.

We compare the performance of a preliminary version of Elemental ported to Blue Gene/P (Elemental-BG/P) with the latest release of ScaLAPACK available from `netlib` (Release 1.8). Since experiments showed that ScaLAPACK performs worse in SMP mode, its performance results are reported with one MPI

process per core with local computation performed by calls to BLAS provided by IBM's serial ESSL library. Elemental-BG/P performance is reported with one MPI process per core and with one MPI process per node (SMP mode). The shared memory parallelism simply consists of using IBM's threaded ESSL BLAS library and adding simple OpenMP directives for parallelizing the packing and unpacking steps surrounding MPI calls. Both packages were tested for all block sizes in the range $32, 64, 96, ..., 256$, with only the best-performing block size for each problem size reported in the graphs. All reported computations were performed in double-precision (64-bit) arithmetic.

4.2   Operations

Solution of the Hermitian 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 the coauthors of this paper from Argonne National Lab. This operations is typically broken down into six steps:

- **Cholesky factorization.** $B \rightarrow LL^H$ where L is lower triangular.

- **Reduction of the generalized problem to Hermitian standard form.** The transformation $C := L^{-1}AL^{-H}$.

- **Householder reduction to tridiagonal form.** Computes unitary Q (as a sequence of Householder transformations) so that $T = QCQ^H$ is tridiagonal.

- **Spectral decomposition of a tridiagonal matrix.** Computes unitary V such that $T = VDV^H$ where D is diagonal. This part of the problem is not included in the performance experiments. It could employ, for example, the PMRRR algorithms [4].

- **Back transformation.** This operations computes $Z = Q^H V$ by applying the Householder transformations that represent Q to V.

- **Solution of a triangular system of equations with multiple right-hand sides.** Now $A(L^{-H}Z) = C(L^{-H}Z)$ and hence $X = L^{-H}Z$ equal the desired generalized eigenvectors. This requires the solution of the triangular system of equations $LX = Z$.

Since complex arithmetic requires four times as many floating point operations, we tuned and measured performance for double-precision real matrices. The represented performance is qualitatively representative of that observed for the complex case. To further reduce the time required for collecting data we limited the largest problem size for which performance is reported to $100,000 \times 100,000$, which requires less than one percent of the available main memory of the 2048 nodes for each distributed matrix.

4.3   Results

The point of this section is to show that preliminary experience with Elemental supports the claim that it provides a solution to the programmability problem without sacrificing performance. Potential users of ScaLAPACK and Elemental are encouraged to perform and report their own comparisons.

Cholesky factorization. The performance of Cholesky factorization is reported in Figure 4. The performance improvements for Cholesky factorization are largely due to inefficiencies in ScaLAPACK's approach to applying the symmetric update to A_{22}. More specifically, in order to take advantage of symmetry in the update of A_{22}, ScaLAPACK breaks the lower triangle of A_{22} into a series of column panels of a fixed width and updates each using a call to gemm. This width is fixed as 32 within the routine pilaenv and preliminary experiments show significant increases in performance when the library is recompiled with larger choice of the parameter, e.g., 128. We did not test ScaLAPACK over a range of these parameters because each choice requires one to recompile the library and this greatly complicates the experiments. We thus used the version that has been in use on Argonne's BG/P for the past several years. Elemental avoids the need for this tuning parameter by updating A_{22} via recursive partitioning of the local computation. This approach results in the vast majority of the work lying in dgemm updates of large square matrices. The level of abstraction at which
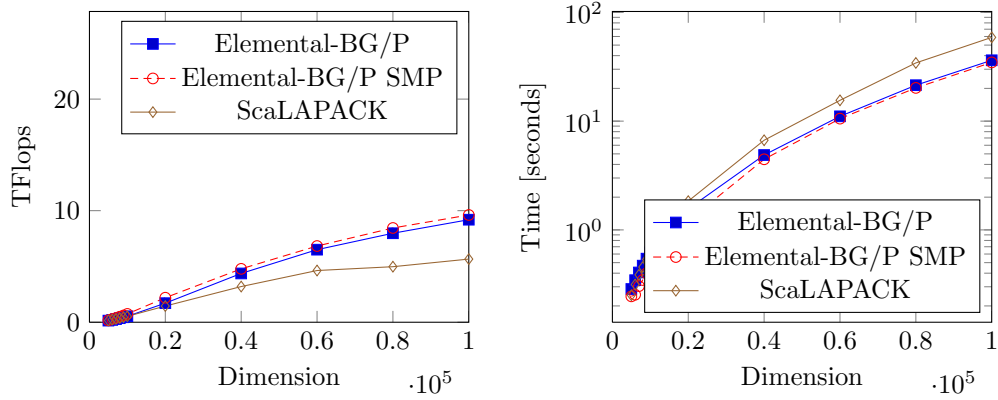
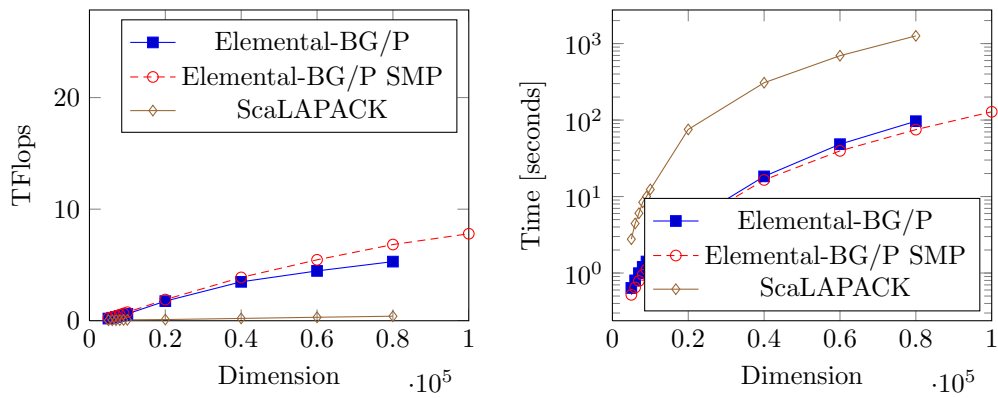Figure 4: Real double-precision Cholesky factorization on 8192 cores.

Figure 5: Real double-precision reduction of generalized eigenvalue problem to symmetric standard form on 8192 cores.
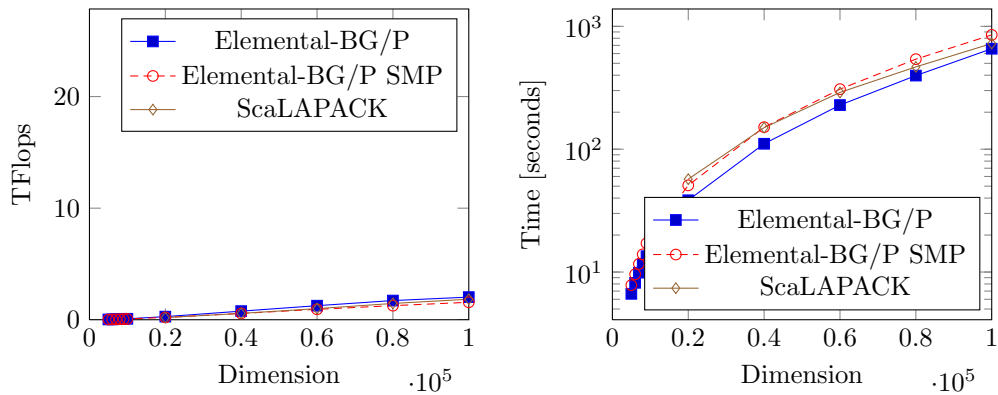

Figure 6: Real double-precision Householder tridiagonalization on 8192 cores.
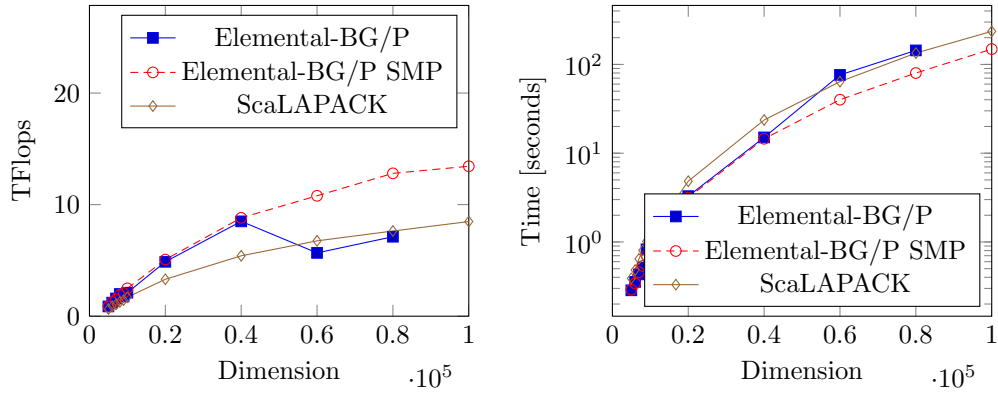
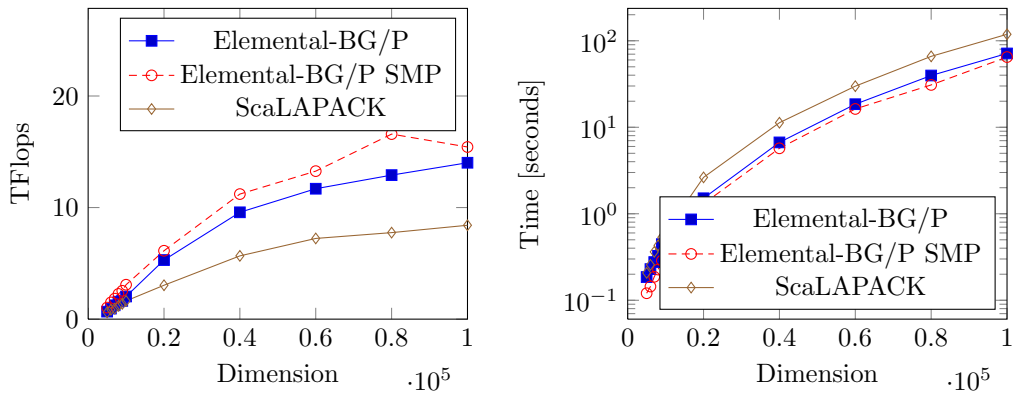Figure 7: Real double-precision application of the back transformation on 8192.

Figure 8: Real double-precision $B := L^{-T}B$ (**trsm**) on 8192 cores.

11

Elemental routines are coded greatly simplifies the incorporation of such strategies for eliminating tuning parameters.

Reduction of generalized eigenvalue problem to Hermitian standard form. Performance of this operation is reported in Figure 5. For this operation, ScaLAPACK utilizes an algorithm that is nonscalable by casting most computation in terms of the parallel solution of a triangular system with multiple right-hand sides (`trsm`) and parallel triangular matrix-matrix multiplication (`trmm`). These operations work with only a narrow panel of right-hand sides (with width equal to the block size), which greatly limits the opportunity for parallelism. By contrast Elemental uses a new algorithm [20] that casts most computation in terms of rank-k updates, which are easily parallelizable. The vast performance improvement observed in this routine can be entirely attributed to the new algorithm avoiding unscalable updates.

Householder reduction to tridiagonal form. In Figure 6, we report the performance for the reduction to tridiagonal form. This operation is the most time consuming step towards finding all eigenvalues and eigenvectors of a symmetric or Hermitian matrix. Neither package attains the same level of performance as do the other operations. The reason for this is that a substantial part of the computation is in a (local) matrix-vector multiplication, which is inherently constrained by memory bandwidth. The algorithms used by both packages are essentially the same.

Back transformation. This operation requires the Householder transformations that form Q to be applied to the eigenvectors of the tridiagonal matrix T. Our parallel implementation employs block UT transforms [17] in order to cast most of the computation in terms of efficient matrix-matrix multiplications. Results are reported in Figure 7. The dramatic drop in the performance curve for Elemental is due to a large drop in efficiency of IBM's MPI_Reduce_scatter implementation over the (Z, T) subtori once the message sizes are larger than a particular value. We are in the process of working out the problem with IBM.

Solution of a triangular system of equations with multiple right-hand sides. Parallelization of this operation was first published in [7] and it is part of ScaLAPACK's Parallel BLAS (PBLAS). Results are reported in Figure 7.

4.4 Discussion

Both ScaLAPACK and Elemental have considerably large searchspaces of tuning parameters, so our experiments always included testing both packages over the same large range of blocksizes, process grid dimensions, and matrix sizes. Unfortunately an exhaustive search is infeasible, so we make no claims that we have found the optimal parameters for each package. However, we believe that our experiments were sufficiently detailed for qualitative comparison. In particular, we would like to point out that the factor of 20 performance improvement observed in the reduction of the generalized eigenvalue problem resulted from a smarter algorithm rather than a smarter implementation; all other performance differences are insignificant in comparison.

5 Conclusion

The point of this paper is to demonstrate that, for the domain of dense linear algebra libraries, neither abstraction nor elegance needs to stand in the way of performance, even on distributed memory architectures. Therefore, it is time for the community to embrace notations, techniques, algorithms, abstractions, and APIs that help solve the programmability problem in preparation for exascale computing, rather than remaining fixated on performance at the expense of sanity. On future architectures that also incorporate GPUs, the effect of programmability on performance is expected to be even more pronounced.

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

- All level-3 BLAS.

- All three (one-sided) matrix factorizations: LU with partial pivoting, Cholesky, and QR factorization.

- All operations for computing the inverse of a symmetric/Hermitian positive-definite matrix.

- All operations for computing the solution of the generalized symmetric/Hermitian positive-definite eigenvalue problem, with the exception of the parallel solution of the symmetric tridiagonal eigensolver.

Soon, we expect to have functionality that rivals that of ScaLAPACK, with the possible exception of solvers for the nonsymmetric eigenvalue problem.

We envision a string of additional papers in the near future. In Appendix A, we hint at a general set-based notation for describing the data distributions that underly Elemental. The key insight is that relations between sets, via the union operator, that link the different destributions dictate the communications that are required for parallelization. A full paper on this topic is being written. Another paper will give details regarding the parallel computation of the solution of the generalized Hermitian eigenvalue problem.

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. The results of that experiment will also be reported in a future publication.

Availability

The Elemental package is available at `http://code.google.com/p/elemental` and its port to Blue Gene/P at `http://code.google.com/p/elemental-bgp`.

References

[1] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*, 1997.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[3] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Lapack for distributed memory architectures: Progress report. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 625–630, Philadelphia, 1992. SIAM.

[4] Paolo Bientinesi, Inderjit S. Dhillon, and Robert A. van de Geijn. A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. *SIAM Journal on Scientific Computing*, 27(1):43–66, 2005.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[6] Ernie Chan, Enrique Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. 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*, pages 116–126, 2007.

[7] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, and Robert A. van de Geijn. Parallel implementation of BLAS: General techniques for level 3 BLAS. *Concurrency: Practice and Experience*, 9(9):837–857, Sept. 1997.

[8] Jack Dongarra and Robert van de Geijn. Reduction to condensed form on distributed memory architectures. *Parallel Computing*, 18:973–982, 1992.

[9] Jack Dongarra, Robert van de Geijn, and David Walker. Scalability issues affecting the design of a dense linear algebra library. *J. Parallel Distrib. Comput.*, 22(3), Sept. 1994.

[10] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[11] C. Edwards, P. Geng, A. Patra, and R. van de Geijn. Parallel matrix distributions: have we been doing it all wrong? Technical Report TR-95-40, Department of Computer Sciences, The University of Texas at Austin, 1995.

[12] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3: Article 12, 25 pages), May 2008.

[13] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[14] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput.*, 15(5):1201–1226, 1994.

[15] Bruce Hendrickson, Elizabeth Jessup, and Christopher Smith. Toward an efficient parallel eigensolver for dense symmetric matrices. *SIAM J. Sci. Comput.*, 20(3):1132–1154, 1999.

[16] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De1, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the International Solid-State Circuits Conference*, February 2010.

[17] Thierry Joffrain, Tze Meng Low, Enrique S. Quintana-Ortí, Robert van de Geijn, and Field G. Van Zee. Accumulating householder transformations, revisited. *ACM Trans. Math. Softw.*, 32(2):169–179, 2006.

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

[19] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[20] Jack Poulson. Formalized parallel dense linear algebra and its application to the generalized eigenvalue problem. Master's thesis, Department of Aerospace Engineering, The University of Texas, May 2009.

[21] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3):14:1–14:26, July 2009.

[22] Home Page, 2010. `http://www.netlib.org/scalapack/scalapack_home.html`.

[23] R. Schreiber. Scalability of sparse direct solvers. *Graph Theory and Sparse Matrix Computations*, 56, 1992.

[24] G. W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990.

[25] P. E. Strazdins. 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*, Dec. 1998.

[26] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997.

[27] Field G. Van Zee. `libflame`: *The Complete Reference.* `www.lulu.com`, 2009.

[28] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

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

Note to the referees: We envision Appendix A and B as electronic appendices so that color has meaning.

A   Elemental Distribution Details

In this appendix, we describe the basics of the distribution used by Elemental and how it facilitates parallel Cholesky factorization.

In our discussion, we assume that the p processes from a (logical) $r \times c$ mesh with $p = rc$. We let $A \in \mathbb{R}^{m \times n}$ equal

$$A = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \cdots & \alpha_{0(n-1)} \\ \alpha_{10} & \alpha_{11} & \cdots & \alpha_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{(m-1)0} & \alpha_{(m-1)1} & \cdots & \alpha_{(m-1)(n-1)} \end{pmatrix}.$$

A.1   Distribution $A(\mathcal{M}_C, \mathcal{M}_R)$

The basic elemental matrix distribution of matrix A assigns to process (s,t) of an $r \times c$ mesh of processes the submatrix

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

In Matlab notation (starting indexing at zero) this means that process (s,t) owns submatrix $\alpha_{s:r:m,t:c:n}$. We will denote this distribution by $A(\mathcal{M}_C, \mathcal{M}_R)$ where $\mathcal{M}_C$ can be thought of as the set of sets of integers $\{M_C^{s,t}\}$ with $s = 0, \ldots, r-1$ and $t = 0, \ldots, c-1$. The sets $\{M_C^{s,t}\}$ and $\{M_R^{s,t}\}$ indicate two "filters" that determine the row and column indices, respectively, of the matrix that are assigned to process (s,t). Figure 9 illustrates this.

The $\mathcal{M}_C$ and $\mathcal{M}_R$ are meant to represent a partitioning of the natural numbers (which include zero since we are computer scientists) into r and c nonoverlapping subsets, respectively: $\mathcal{M}_C = \left(\mathcal{M}_C^0, \mathcal{M}_C^1, \ldots, \mathcal{M}_C^{r-1}\right)$ and $\mathcal{M}_R = \left(\mathcal{M}_R^0, \mathcal{M}_R^1, \ldots, \mathcal{M}_R^{c-1}\right)$. Now, $A(\mathcal{M}_C^{s,t}, \mathcal{M}_R^{s,t})$ represents the submatrix of A that is formed by only choosing the row indices from $\mathcal{M}_C^{s,t}$ and column indices from $\mathcal{M}_R^{s,t}$. The notation $A(\mathcal{M}_C, \mathcal{M}_R)$ is meant to indicate the distribution that assigns $A(\mathcal{M}_C^{s,t}, \mathcal{M}_R^{s,t})$ to each process (s,t).

While this notation captures a broad family of distributions, in the elemental distribution $\mathcal{M}_C^s = \{s, s+r, s+2r, \ldots\}$ and $\mathcal{M}_R^t = \{t, t+c, t+2c, \ldots\}$ creating the desired round-robin distribution in Figure 9.

A.2   Distribution $A(\mathcal{V}_C, \star)$

This second distribution can be described as follows: The processes still forms an $r \times c$ mesh, but now the processes are numbered in column-major order: process $(0,0)$ is process 0. process $(1,0)$ is process 1, etc. process u in this numbering is assigned elements $\alpha_{u,*}, \alpha_{u+p,*}, \ldots$ where $*$ indicates all valid column indices. Another way of describing this is that the processes are now viewed as forming a one-dimensional array and rows of the matrix are wrapped onto this array in a round-robin fashion. This is illustrated in Figure 10.

What is important is that it is very easy to redistribute from $A(\mathcal{M}_C, \mathcal{M}_R)$ to $A(\mathcal{V}_C, \star)$. Focus on the first row of matrix A in Figures 9 and 10. The elements of row 0 of A in Figure 9 need to be *gathered* within the first row of processes to process $(0,0)$ so that it becomes distributed as in Figure 10. Similarly, the elements of row 1 of A need to be gathered to process $(1,0)$. A careful comparison of the two figures shows that all-to-all communications within rows of processes will redistribute $A(\mathcal{M}_C, \mathcal{M}_R)$ to $A(\mathcal{V}_C, \star)$.

The $\mathcal{V}_C$ represents a partitioning of the natural numbers into a 2D array of p nonoverlapping subsets:

$$\mathcal{V}_C = \begin{pmatrix} \mathcal{V}_C^{0,0}, \ldots, \mathcal{V}_C^{0,c-1} \\ \vdots \\ \mathcal{V}_C^{r-1,0}, \ldots, \mathcal{V}_C^{r-1,c-1} \end{pmatrix}$$

Figure 9:

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

Figure 9: Illustration of distribution $A(\mathcal{M}_C, \mathcal{M}_R)$ where $r = c = 3$, $\mathcal{M}_C^{s,t} = \{s, s+r, \ldots\}$, and $\mathcal{M}_R^{s,t} = \{t, t+c, \ldots\}$. Here the (s,t) tile represents process (s,t).

Figure 10:

Process 0				Process 3				Process 6			
$\alpha_{0,0}$	$\alpha_{0,1}$	$\alpha_{0,2}$	$\cdots$	$\alpha_{3,0}$	$\alpha_{3,1}$	$\alpha_{3,2}$	$\cdots$	$\alpha_{6,0}$	$\alpha_{6,1}$	$\alpha_{6,2}$	$\cdots$
$\alpha_{9,0}$	$\alpha_{9,1}$	$\alpha_{0,2}$	$\cdots$	$\alpha_{12,0}$	$\alpha_{12,1}$	$\alpha_{12,2}$	$\cdots$	$\alpha_{15,0}$	$\alpha_{15,1}$	$\alpha_{15,2}$	$\cdots$
$\alpha_{18,0}$	$\alpha_{18,1}$	$\alpha_{18,2}$	$\cdots$	$\alpha_{21,0}$	$\alpha_{21,1}$	$\alpha_{21,2}$	$\cdots$	$\alpha_{24,0}$	$\alpha_{24,1}$	$\alpha_{24,2}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
Process 1				Process 4				Process 7			
$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$	$\cdots$	$\alpha_{4,0}$	$\alpha_{4,1}$	$\alpha_{4,2}$	$\cdots$	$\alpha_{7,0}$	$\alpha_{7,1}$	$\alpha_{7,2}$	$\cdots$
$\alpha_{10,0}$	$\alpha_{10,1}$	$\alpha_{10,2}$	$\cdots$	$\alpha_{13,0}$	$\alpha_{13,1}$	$\alpha_{13,2}$	$\cdots$	$\alpha_{16,0}$	$\alpha_{16,1}$	$\alpha_{16,2}$	$\cdots$
$\alpha_{19,0}$	$\alpha_{19,1}$	$\alpha_{19,2}$	$\cdots$	$\alpha_{22,0}$	$\alpha_{22,1}$	$\alpha_{22,2}$	$\cdots$	$\alpha_{25,0}$	$\alpha_{25,1}$	$\alpha_{25,2}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
Process 2				Process 5				Process 8			
$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$	$\cdots$	$\alpha_{5,0}$	$\alpha_{5,1}$	$\alpha_{5,2}$	$\cdots$	$\alpha_{8,0}$	$\alpha_{8,1}$	$\alpha_{8,2}$	$\cdots$
$\alpha_{11,0}$	$\alpha_{11,1}$	$\alpha_{11,2}$	$\cdots$	$\alpha_{14,0}$	$\alpha_{14,1}$	$\alpha_{14,2}$	$\cdots$	$\alpha_{17,0}$	$\alpha_{17,1}$	$\alpha_{17,2}$	$\cdots$
$\alpha_{20,0}$	$\alpha_{20,1}$	$\alpha_{20,2}$	$\cdots$	$\alpha_{23,0}$	$\alpha_{23,1}$	$\alpha_{23,2}$	$\cdots$	$\alpha_{26,0}$	$\alpha_{26,1}$	$\alpha_{26,2}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 10: Illustration of distribution $A(\mathcal{V}_C, *)$ where $r = c = 3$ and $\mathcal{V}_C^{s,t} = \{u, u+p, \ldots\}$, with $u = (formula)$.

Now, $A(\mathcal{V}_C^{s,t}, \star)$ represents the submatrix of A that is formed by only choosing the rows with indices found in $\mathcal{V}_C^{s,t}$. The notation $A(\mathcal{V}_C, \star)$ is meant to indicate the distribution that assigns $A(\mathcal{V}_C^{s,t}, \star)$ to each process (s,t). In the case where A is a vector, x, the notation $x(\mathcal{V}_C)$ can be used.

This notation also captures a broad family of distributions. In the elemental distribution $\mathcal{V}_C^{s,t} = \{u, u+p, u+2p, \ldots\}$, where $u = s + rt$, creating the round-robin distribution in Figure 10.

The very important relation between $\mathcal{M}_C^s$ and $\mathcal{V}_C^{s,t}$ is that $\mathcal{M}_C^s = \cup_{t=0}^{c-1} \mathcal{V}_C^{s,t}$. It is this property that

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

Figure 11: (Repeat) Illustration of distribution $A(\mathcal{M}_C, \mathcal{M}_R)$ where $r = c = 3$, $\mathcal{M}_{s,t}^C = \{s, s+r, \ldots\}$, and $\mathcal{M}_R^{s,t} = \{t, t+c, \ldots\}$.

Process 0				Process 1				Process 2			
$\alpha_{0,0}$	$\alpha_{0,9}$	$\alpha_{0,18}$	$\cdots$	$\alpha_{0,1}$	$\alpha_{0,10}$	$\alpha_{0,19}$	$\cdots$	$\alpha_{0,2}$	$\alpha_{0,11}$	$\alpha_{0,20}$	$\cdots$
$\alpha_{1,0}$	$\alpha_{1,9}$	$\alpha_{1,18}$	$\cdots$	$\alpha_{1,1}$	$\alpha_{1,10}$	$\alpha_{1,19}$	$\cdots$	$\alpha_{1,2}$	$\alpha_{1,11}$	$\alpha_{1,20}$	$\cdots$
$\alpha_{2,0}$	$\alpha_{2,9}$	$\alpha_{2,18}$	$\cdots$	$\alpha_{2,1}$	$\alpha_{2,10}$	$\alpha_{2,19}$	$\cdots$	$\alpha_{2,2}$	$\alpha_{2,11}$	$\alpha_{2,20}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
Process 3				Process 4				Process 5			
$\alpha_{0,3}$	$\alpha_{0,12}$	$\alpha_{0,21}$	$\cdots$	$\alpha_{0,4}$	$\alpha_{0,13}$	$\alpha_{0,22}$	$\cdots$	$\alpha_{0,5}$	$\alpha_{0,14}$	$\alpha_{0,23}$	$\cdots$
$\alpha_{1,3}$	$\alpha_{1,12}$	$\alpha_{1,21}$	$\cdots$	$\alpha_{1,4}$	$\alpha_{1,13}$	$\alpha_{1,22}$	$\cdots$	$\alpha_{1,5}$	$\alpha_{1,14}$	$\alpha_{1,23}$	$\cdots$
$\alpha_{2,3}$	$\alpha_{2,12}$	$\alpha_{2,21}$	$\cdots$	$\alpha_{2,4}$	$\alpha_{2,13}$	$\alpha_{2,22}$	$\cdots$	$\alpha_{2,5}$	$\alpha_{2,14}$	$\alpha_{2,23}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
Process 6				Process 7				Process 8			
$\alpha_{0,6}$	$\alpha_{0,15}$	$\alpha_{0,24}$	$\cdots$	$\alpha_{0,7}$	$\alpha_{0,16}$	$\alpha_{0,25}$	$\cdots$	$\alpha_{0,8}$	$\alpha_{0,17}$	$\alpha_{0,26}$	$\cdots$
$\alpha_{1,6}$	$\alpha_{1,15}$	$\alpha_{1,24}$	$\cdots$	$\alpha_{1,7}$	$\alpha_{1,16}$	$\alpha_{1,25}$	$\cdots$	$\alpha_{1,8}$	$\alpha_{1,17}$	$\alpha_{1,26}$	$\cdots$
$\alpha_{2,6}$	$\alpha_{2,15}$	$\alpha_{2,24}$	$\cdots$	$\alpha_{2,7}$	$\alpha_{2,16}$	$\alpha_{2,25}$	$\cdots$	$\alpha_{2,8}$	$\alpha_{2,17}$	$\alpha_{2,26}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 12: Illustration of distribution $A(\star, \mathcal{V}_R)$ where $r = c = 3$ and $\mathcal{V}_R^{s,t} = \{u, u+p, \ldots\}$, with $u = (formula)$.

guarantees that redistributing from $A(\mathcal{V}_C, \star)$ to $A(\mathcal{M}_C, \mathcal{M}_R)$ and vise versa requires only simultaneous all-to-all communications within rows.

A.3 Distribution $A(\star, \mathcal{V}_R)$

This distribution can be similarly described: The processes still forms an $r \times c$ mesh, but now the processes are numbered in row-major order: process $(0,0)$ is process 0. process $(0,1)$ is process 1, etc. process u in this numbering is assigned elements $\alpha_{*,u}, \alpha_{*,u+p}, \ldots$ where $*$ indicates all valid row indices. Another way of describing this is that the processes are now viewed as forming a one-dimensional array and columns of the matrix are wrapped onto this array in a round-robin fashion. This is illustrated in Figure 12.

Again, what is important is that it is very easy to redistribute from $A(\mathcal{M}_C, \mathcal{M}_R)$ to $A(\star, \mathcal{V}_R)$. Focus on the first column of matrix A in Figure 11 (which is just a repeat of Figure 9 for easy comparison) and Figure 12. The elements of column 0 of A in Figure 11 need to be *gathered* within the first column of processes to process $(0,0)$ so that it becomes distributed as in Figure 12. Similarly, the elements of column 1 of A need to be gathered to process $(0,1)$. A careful comparison of the two figures shows that all-to-all communications within columns of processes will redistribute $A(\mathcal{M}_C, \mathcal{M}_R)$ to $A(\star, \mathcal{V}_R)$.

The $\mathcal{V}_R$ similarly represents a partitioning into the 2D array of nonoverlapping subsets

$$
\mathcal{V}_R = \begin{pmatrix} \mathcal{V}_R^{0,0}, \ldots, \mathcal{V}_R^{0,c-1} \\ \vdots \\ \mathcal{V}_R^{r-1,0}, \ldots, \mathcal{V}_R^{r-1,c-1} \end{pmatrix}
$$

Now, $A(\star, \mathcal{V}_R^{s,t})$ represents the submatrix of A that is formed by only choosing the columns from $\mathcal{V}_R^{s,t}$. The notation $A(\star, \mathcal{V}_R)$ is meant to indicate the distribution that assigns $A(\star, \mathcal{V}_R^{s,t})$ to each process (s,t). In the case where A is a vector, x, the notation $x(\mathcal{V}_R)$ can be used.

This notation again captures a broad family of distributions. In the elemental distribution $\mathcal{V}_C^{s,t} = \{u, u+p, u+2p, \ldots\}$, where $u = sc + t$ creating the round-robin distribution in Figure 12.

The very important relation between $\mathcal{M}_R^t$ and $\mathcal{V}_R^{s,t}$ is that $\mathcal{M}_R^t = \cup_{s=0}^{r-1} \mathcal{V}_R^{s,t}$. It is this property that guarantees that redistributing from $A(\star, \mathcal{V}_R)$ to $A(\mathcal{M}_C, \mathcal{M}_R)$ and vise versa requires only simultaneous all-to-all communications within columns.

A.4 Distribution $A(\mathcal{M}_C, \star)$

Let $\mathcal{M}_C$ be as before. Then $A(\mathcal{M}_C, \star)$ assigns to process (s,t) the elements $A(\mathcal{M}_C^s, \star)$. In the case of the elemental distribution, this means that process (s,t) is assigned the submatrix

$$
\begin{pmatrix} \alpha_{s,\star} \\ \alpha_{s+r,\star} \\ \vdots \end{pmatrix}
$$

as illustrated in Figure 13.

It is important to note that redistributing from $A(\mathcal{V}_C, \star)$ to $A(\mathcal{M}_C, \star)$ requires simultaneous allgathers within columns, as is obvious from comparing Figures 10 and 13. Similarly, redistributing from $A(\mathcal{M}_C, \mathcal{M}_R)$ to $A(\mathcal{M}_C, \star)$ requires simultaneous allgathers within columns, as is obvious from comparing Figures 9 and 13.

A.5 Distribution $A(\star, \mathcal{M}_R)$

Let $\mathcal{M}_R$ be as before. Then $A(\star, \mathcal{M}_R)$ assigns to process (s,t) the elements $A(\star, \mathcal{M}_R^t)$. In the case of the elemental distribution, this means that process (s,t) is assigned the submatrix

$$
\begin{pmatrix} \alpha_{\star,t} & \alpha_{\star,t+c} & \cdots \end{pmatrix}
$$

as illustrated in Figure 14.

It is important to note that redistributing from $A(\star, \mathcal{V}_R)$ to $A(\star, \mathcal{M}_R)$ requires simultaneous allgathers within columns, as is obvious from comparing Figures 12 and 14. Similarly, redistributing from $A(\mathcal{M}_C, \mathcal{M}_R)$ to $A(\star, \mathcal{M}_R)$ requires simultaneous allgathers within rows, as is obvious from comparing Figures 9 and 14.

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

Figure 13: Illustration of distribution $A(\mathcal{M}_C, \star)$ where $r = c = 3$.

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

Figure 14: Illustration of distribution $A(\star, \mathcal{M}_R)$ where $r = c = 3$.

B   An Illustrated Guide to Elemental Cholesky Factorization

In this appendix, we illustrate the Cholesky factorization routine in Figure 3. We will work through the commands in the loop body one by one for the first iteration of the algorithm. For each command, we explain how data is redistributed and what computation is performed where. We assume that the reader is reasonably familiar with Cholesky factorization, the BLAS, and LAPACK. While we illustrate the algorithm on a 3×3 mesh of processes and use a distribution block size of 3, there is nothing special about the mesh being square and the block size being equal to the mesh row and column size.

We start with matrix A distributed among the processes:

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

a distribution denoted by $A(\mathcal{M}_C, \mathcal{M}_R)$. Here the elements of A_{11}, A_{12}, and A_{22} are highlighted in blue, green, and red, respectively.

B.1 Parallel factorization of A_{11}

The command
```
A11_Star_Star = A11;
```
creates a copy of A_{11} on each process as illustrated by

Process (0,0)			Process (0,1)			Process (0,2)		
$\alpha_{0,0}$	$\alpha_{0,1}$	$\alpha_{0,2}$	$\alpha_{0,0}$	$\alpha_{0,1}$	$\alpha_{0,2}$	$\alpha_{0,0}$	$\alpha_{0,1}$	$\alpha_{0,2}$
$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$	$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$	$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$
$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$	$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$	$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$
Process (1,0)			Process (1,1)			Process (1,2)		
$\alpha_{0,0}$	$\alpha_{0,1}$	$\alpha_{0,2}$	$\alpha_{0,0}$	$\alpha_{0,1}$	$\alpha_{0,2}$	$\alpha_{0,0}$	$\alpha_{0,1}$	$\alpha_{0,2}$
$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$	$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$	$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$
$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$	$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$	$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$
Process (2,0)			Process (2,1)			Process (2,2)		
$\alpha_{0,0}$	$\alpha_{0,1}$	$\alpha_{0,2}$	$\alpha_{0,0}$	$\alpha_{0,1}$	$\alpha_{0,2}$	$\alpha_{0,0}$	$\alpha_{0,1}$	$\alpha_{0,2}$
$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$	$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$	$\alpha_{1,0}$	$\alpha_{1,1}$	$\alpha_{1,2}$
$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$	$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$	$\alpha_{2,0}$	$\alpha_{2,1}$	$\alpha_{2,2}$

This requires an allgather among all processes. Next, the command
```
lapack::internal::LocalChol( Upper, A11_Star_Star );
```
factors A_{11}, redundantly, on each process. The command
```
A11 = A11_Star_Star;
```
places the updated entries of A_{11} back where they belong in the distributed matrix, which does not require communication since each process owns a copy.

B.2 Parallel update $A_{12} := U_{11}^{-H} A_{12}$

Now, let us highlight A_{12} for the first iteration, but with multiple colors so that some of the required data movements can be tracked:

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

Here the colored elements represent the distribution $A_{12}(\star, \mathcal{M}_R)$. Consider the computation $A_{12} := U_{11}^{-H} A_{12}$. Partition A_{12} by columns:

$$A_{12} \rightarrow \begin{pmatrix} \alpha_{12}^3 & \alpha_{12}^4 & \cdots \end{pmatrix}$$

Then $\begin{pmatrix} \alpha_{12}^3 & \alpha_{12}^4 & \cdots \end{pmatrix} := U_{11}^{-H} \begin{pmatrix} \alpha_{12}^3 & \alpha_{12}^4 & \cdots \end{pmatrix} = \begin{pmatrix} U_{11}^{-H} \alpha_{12}^3 & U_{11}^{-H} \alpha_{12}^4 & \cdots \end{pmatrix}$. In other words, each column is updated by a triangular solve with U_{11}. So, to conveniently compute $A_{12} := U_{11}^{-H} A_{12}$ we note that (1) whole columns of A_{12} should exist on the same node and (2) those columns should be wrapped to processes so that they can all participate, as such:

Process 0				Process 1				Process 2			
	$\alpha_{0,9}$	$\alpha_{0,18}$	$\cdots$		$\alpha_{0,10}$	$\alpha_{0,19}$	$\cdots$		$\alpha_{0,11}$	$\alpha_{0,20}$	$\cdots$
	$\alpha_{1,9}$	$\alpha_{1,18}$	$\cdots$		$\alpha_{1,10}$	$\alpha_{1,19}$	$\cdots$		$\alpha_{1,11}$	$\alpha_{1,20}$	$\cdots$
	$\alpha_{2,9}$	$\alpha_{2,18}$	$\cdots$		$\alpha_{2,10}$	$\alpha_{2,19}$	$\cdots$		$\alpha_{2,11}$	$\alpha_{2,20}$	$\cdots$
Process 3				Process 4				Process 5			
$\alpha_{0,3}$	$\alpha_{0,12}$	$\alpha_{0,21}$	$\cdots$	$\alpha_{0,4}$	$\alpha_{0,13}$	$\alpha_{0,22}$	$\cdots$	$\alpha_{0,5}$	$\alpha_{0,14}$	$\alpha_{0,23}$	$\cdots$
$\alpha_{1,3}$	$\alpha_{1,12}$	$\alpha_{1,21}$	$\cdots$	$\alpha_{1,4}$	$\alpha_{1,13}$	$\alpha_{1,22}$	$\cdots$	$\alpha_{1,5}$	$\alpha_{1,14}$	$\alpha_{1,23}$	$\cdots$
$\alpha_{2,3}$	$\alpha_{2,12}$	$\alpha_{2,21}$	$\cdots$	$\alpha_{2,4}$	$\alpha_{2,13}$	$\alpha_{2,22}$	$\cdots$	$\alpha_{2,5}$	$\alpha_{2,14}$	$\alpha_{2,23}$	$\cdots$
Process 6				Process 7				Process 8			
$\alpha_{0,6}$	$\alpha_{0,15}$	$\alpha_{0,24}$	$\cdots$	$\alpha_{0,7}$	$\alpha_{0,16}$	$\alpha_{0,25}$	$\cdots$	$\alpha_{0,8}$	$\alpha_{0,17}$	$\alpha_{0,26}$	$\cdots$
$\alpha_{1,6}$	$\alpha_{1,15}$	$\alpha_{1,24}$	$\cdots$	$\alpha_{1,7}$	$\alpha_{1,16}$	$\alpha_{1,25}$	$\cdots$	$\alpha_{1,8}$	$\alpha_{1,17}$	$\alpha_{1,26}$	$\cdots$
$\alpha_{2,6}$	$\alpha_{2,15}$	$\alpha_{2,24}$	$\cdots$	$\alpha_{2,7}$	$\alpha_{2,16}$	$\alpha_{2,25}$	$\cdots$	$\alpha_{2,8}$	$\alpha_{2,17}$	$\alpha_{2,26}$	$\cdots$

Now, compare this distribution $A_{12}(\star, \mathcal{V}_R)$ to that of $A_{12}(\star, \mathcal{M}_R)$ in the previous picture. We notice that all-to-all communications within process columns are needed to achieve the required redistribution. The command

```
A12_Star_VR = A12;
```

hides all the required communication in the assignment operator. After this the command

```
blas::internal::LocalTrsm
( Left, Upper, ConjugateTranspose, NonUnit,
  (T)1, A11_Star_Star, A12_Star_VR );
```

updates the local content of $A_{12}(\star, \mathcal{V}_R)$ with $U_{11}^{-H} A_{12}$, using the duplicated copy of A_{11} (i.e., $A_{11}(\star, \star)$, which holds U_{11}). We will delay placing the updated entries of A_{12} back where they belong until later.

B.3 Parallel update $A_{22} := A_{22} - A_{12}^{-H}A_{12}$

Finally, consider the update $A_{22} := A_{22} - A_{12}^H A_{12}$. In the following, we highlight the elements of A that are affected:

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

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

$$
\begin{pmatrix}
\alpha_{4,5} & \alpha_{4,8} & \alpha_{4,11} & \cdots \\
\alpha_{7,5} & \alpha_{7,8} & \alpha_{7,11} & \cdots \\
\alpha_{10,5} & \alpha_{10,8} & \alpha_{10,11} & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{pmatrix}
- :=
\begin{pmatrix}
\alpha_{0,4} & \alpha_{0,7} & \alpha_{0,10} & \cdots \\
\alpha_{1,4} & \alpha_{1,7} & \alpha_{1,10} & \cdots \\
\alpha_{2,4} & \alpha_{2,7} & \alpha_{2,10} & \cdots
\end{pmatrix}^H
\begin{pmatrix}
\alpha_{0,5} & \alpha_{0,8} & \alpha_{0,11} & \cdots \\
\alpha_{1,5} & \alpha_{1,8} & \alpha_{1,11} & \cdots \\
\alpha_{2,5} & \alpha_{2,8} & \alpha_{2,11} & \cdots
\end{pmatrix}. \qquad (1)
$$

(Here the gray entries are those that are not updated due to symmetry.) *If* the elements of A_{12} were distributed as illustrated in Figure 15 *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 communications required for these redistributions are explained illustrated in Figures 16 and 17. After this, the simultaneous local computations are accomplished by the call

```
blas::internal::LocalTriangularRankK
( Upper, ConjugateTranspose,
  (T)-1, A12_Star_MC, A12_Star_MR, (T)1, A22 );
```

Finally, we observe that $A_{12}(\star, \mathcal{M}_R)$ duplicates the updated submatrix A_{12} in such a way that placing this data back in the original matrix requires only local copying of data. The command

```
A12 = A12_Star_MR;
```

accomplishes this.

Top figure

Process (0,0)				Process (0,1)				Process (0,2)			
$\alpha_{0,3}$	$\alpha_{0,6}$	$\alpha_{0,9}$	$\cdots$	$\alpha_{0,3}$	$\alpha_{0,6}$	$\alpha_{0,9}$	$\cdots$	$\alpha_{0,3}$	$\alpha_{0,6}$	$\alpha_{0,9}$	$\cdots$
$\alpha_{1,3}$	$\alpha_{1,6}$	$\alpha_{1,9}$	$\cdots$	$\alpha_{1,3}$	$\alpha_{1,6}$	$\alpha_{1,9}$	$\cdots$	$\alpha_{1,3}$	$\alpha_{1,6}$	$\alpha_{1,9}$	$\cdots$
$\alpha_{2,3}$	$\alpha_{2,6}$	$\alpha_{2,9}$	$\cdots$	$\alpha_{2,3}$	$\alpha_{2,6}$	$\alpha_{2,9}$	$\cdots$	$\alpha_{2,3}$	$\alpha_{2,6}$	$\alpha_{2,9}$	$\cdots$

Process (1,0)				Process (1,1)				Process (1,2)			
$\alpha_{0,4}$	$\alpha_{0,7}$	$\alpha_{0,10}$	$\cdots$	$\alpha_{0,4}$	$\alpha_{0,7}$	$\alpha_{0,10}$	$\cdots$	$\alpha_{0,4}$	$\alpha_{0,7}$	$\alpha_{0,10}$	$\cdots$
$\alpha_{1,4}$	$\alpha_{1,7}$	$\alpha_{1,10}$	$\cdots$	$\alpha_{1,4}$	$\alpha_{1,7}$	$\alpha_{1,10}$	$\cdots$	$\alpha_{1,4}$	$\alpha_{1,7}$	$\alpha_{1,10}$	$\cdots$
$\alpha_{2,4}$	$\alpha_{2,7}$	$\alpha_{2,10}$	$\cdots$	$\alpha_{2,4}$	$\alpha_{2,7}$	$\alpha_{2,10}$	$\cdots$	$\alpha_{2,4}$	$\alpha_{2,7}$	$\alpha_{2,10}$	$\cdots$

Process (2,0)				Process (2,1)				Process (2,2)			
$\alpha_{0,5}$	$\alpha_{0,8}$	$\alpha_{0,11}$	$\cdots$	$\alpha_{0,5}$	$\alpha_{0,8}$	$\alpha_{0,11}$	$\cdots$	$\alpha_{0,5}$	$\alpha_{0,8}$	$\alpha_{0,11}$	$\cdots$
$\alpha_{1,5}$	$\alpha_{1,8}$	$\alpha_{1,11}$	$\cdots$	$\alpha_{1,5}$	$\alpha_{1,8}$	$\alpha_{1,11}$	$\cdots$	$\alpha_{1,5}$	$\alpha_{1,8}$	$\alpha_{1,11}$	$\cdots$
$\alpha_{2,5}$	$\alpha_{2,8}$	$\alpha_{2,11}$	$\cdots$	$\alpha_{2,5}$	$\alpha_{2,8}$	$\alpha_{2,11}$	$\cdots$	$\alpha_{2,5}$	$\alpha_{2,8}$	$\alpha_{2,11}$	$\cdots$

Bottom figure

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

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

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

Figure 15: If A_{12} is redistributed both as $A_{12}(\star, \mathcal{M}_C)$ (Top) and $A_{12}(\star, \mathcal{M}_R)$ (Bottom) then local parts of $A_{22} := A_{22} - A_{12}^H A_{12}$ can be computed independently on each process. For example, Process $(1,2)$ can then update its local elements as indicated in Eqn. (1).

Process 0

$\alpha_{0,9}$ $\alpha_{0,18}$ $\cdots$
$\alpha_{1,9}$ $\alpha_{1,18}$ $\cdots$
$\alpha_{2,9}$ $\alpha_{2,18}$ $\cdots$

Process 1

$\alpha_{0,10}$ $\alpha_{0,19}$ $\cdots$
$\alpha_{1,10}$ $\alpha_{1,19}$ $\cdots$
$\alpha_{2,10}$ $\alpha_{2,19}$ $\cdots$

Process 2

$\alpha_{0,11}$ $\alpha_{0,20}$ $\cdots$
$\alpha_{1,11}$ $\alpha_{1,20}$ $\cdots$
$\alpha_{2,11}$ $\alpha_{2,20}$ $\cdots$

Process 3

$\alpha_{0,3}$ $\alpha_{0,12}$ $\alpha_{0,21}$ $\cdots$
$\alpha_{1,3}$ $\alpha_{1,12}$ $\alpha_{1,21}$ $\cdots$
$\alpha_{2,3}$ $\alpha_{2,12}$ $\alpha_{2,21}$ $\cdots$

Process 4

$\alpha_{0,4}$ $\alpha_{0,13}$ $\alpha_{0,22}$ $\cdots$
$\alpha_{1,4}$ $\alpha_{1,13}$ $\alpha_{1,22}$ $\cdots$
$\alpha_{2,4}$ $\alpha_{2,13}$ $\alpha_{2,22}$ $\cdots$

Process 5

$\alpha_{0,5}$ $\alpha_{0,14}$ $\alpha_{0,23}$ $\cdots$
$\alpha_{1,5}$ $\alpha_{1,14}$ $\alpha_{1,23}$ $\cdots$
$\alpha_{2,5}$ $\alpha_{2,14}$ $\alpha_{2,23}$ $\cdots$

Process 6

$\alpha_{0,6}$ $\alpha_{0,15}$ $\alpha_{0,24}$ $\cdots$
$\alpha_{1,6}$ $\alpha_{1,15}$ $\alpha_{1,24}$ $\cdots$
$\alpha_{2,6}$ $\alpha_{2,15}$ $\alpha_{2,24}$ $\cdots$

Process 7

$\alpha_{0,7}$ $\alpha_{0,16}$ $\alpha_{0,25}$ $\cdots$
$\alpha_{1,7}$ $\alpha_{1,16}$ $\alpha_{1,25}$ $\cdots$
$\alpha_{2,7}$ $\alpha_{2,16}$ $\alpha_{2,25}$ $\cdots$

Process 8

$\alpha_{0,8}$ $\alpha_{0,17}$ $\alpha_{0,26}$ $\cdots$
$\alpha_{1,8}$ $\alpha_{1,17}$ $\alpha_{1,26}$ $\cdots$
$\alpha_{2,8}$ $\alpha_{2,17}$ $\alpha_{2,26}$ $\cdots$

Process (0,0)

$\alpha_{0,3}$ $\alpha_{0,6}$ $\alpha_{0,9}$ $\cdots$
$\alpha_{1,3}$ $\alpha_{1,6}$ $\alpha_{1,9}$ $\cdots$
$\alpha_{2,3}$ $\alpha_{2,6}$ $\alpha_{2,9}$ $\cdots$

Process (0,1)

$\alpha_{0,4}$ $\alpha_{0,7}$ $\alpha_{0,9}$ $\cdots$
$\alpha_{1,4}$ $\alpha_{1,7}$ $\alpha_{1,9}$ $\cdots$
$\alpha_{2,4}$ $\alpha_{2,7}$ $\alpha_{2,9}$ $\cdots$

Process (0,2)

$\alpha_{0,5}$ $\alpha_{0,8}$ $\alpha_{0,9}$ $\cdots$
$\alpha_{1,5}$ $\alpha_{1,8}$ $\alpha_{1,9}$ $\cdots$
$\alpha_{2,5}$ $\alpha_{2,8}$ $\alpha_{2,9}$ $\cdots$

Process (1,0)

$\alpha_{0,3}$ $\alpha_{0,6}$ $\alpha_{0,9}$ $\cdots$
$\alpha_{1,3}$ $\alpha_{1,6}$ $\alpha_{1,9}$ $\cdots$
$\alpha_{2,3}$ $\alpha_{2,6}$ $\alpha_{2,9}$ $\cdots$

Process (1,1)

$\alpha_{0,4}$ $\alpha_{0,7}$ $\alpha_{0,9}$ $\cdots$
$\alpha_{1,4}$ $\alpha_{1,7}$ $\alpha_{1,9}$ $\cdots$
$\alpha_{2,4}$ $\alpha_{2,7}$ $\alpha_{2,9}$ $\cdots$

Process (1,2)

$\alpha_{0,5}$ $\alpha_{0,8}$ $\alpha_{0,9}$ $\cdots$
$\alpha_{1,5}$ $\alpha_{1,8}$ $\alpha_{1,9}$ $\cdots$
$\alpha_{2,5}$ $\alpha_{2,8}$ $\alpha_{2,9}$ $\cdots$

Process (2,0)

$\alpha_{0,3}$ $\alpha_{0,6}$ $\alpha_{0,9}$ $\cdots$
$\alpha_{1,3}$ $\alpha_{1,6}$ $\alpha_{1,9}$ $\cdots$
$\alpha_{2,3}$ $\alpha_{2,6}$ $\alpha_{2,9}$ $\cdots$

Process (2,1)

$\alpha_{0,4}$ $\alpha_{0,7}$ $\alpha_{0,9}$ $\cdots$
$\alpha_{1,4}$ $\alpha_{1,7}$ $\alpha_{1,9}$ $\cdots$
$\alpha_{2,4}$ $\alpha_{2,7}$ $\alpha_{2,9}$ $\cdots$

Process (2,2)

$\alpha_{0,5}$ $\alpha_{0,8}$ $\alpha_{0,9}$ $\cdots$
$\alpha_{1,5}$ $\alpha_{1,8}$ $\alpha_{1,9}$ $\cdots$
$\alpha_{2,5}$ $\alpha_{2,8}$ $\alpha_{2,9}$ $\cdots$

Figure 16: An illustration of the result of command `A12_Star_MR = A12_Star_VR;`. The updated A_{12} starts distributed as $A_{12}(\star, \mathcal{V}_R)$ (Top) and needs to be redistributed to $A_{12}(\star, \mathcal{M}_R)$ (Bottom), which is also depicted in Figure 15 (Bottom). Comparing the two pictures, one notices that allgathers within process columns accomplish the required data movements.

Figure 17: An illustration of the result of command `A12_Star_MC = A12_Star_VR;`. The updated A_{12} starts distributed as $A_{12}(\star, \mathcal{V}_R)$ as in Figure 16 (Top). The command first triggers a redistribution to $A_{12}(\star, \mathcal{V}_C)$ depicted above in the top picture. Comparing the top picture in Figure 16 to the top picture above shows that the required data movement is a permutation: If the data on each process is viewed as a unit, these units are rearranged from being assigned to processes in row-major order to column-major order. Next, the data needs to be redistributed from $A_{12}(\star, \mathcal{V}_C)$ (Top) to $A_{12}(\star, \mathcal{M}_C)$ (Bottom), which is also depicted in Figure 15 (Top). Comparing the two pictures, one notices that allgathers within process rows accomplish the required data movements.