Elemental: A New Framework for Distributed Memory Dense Matrix Computations

FLAME Working Note #44

Jack Poulson
Bryan Marker
Robert van de Geijn
Institute for Computational Engineering and Sciences
and Department of Computer Science
The University of Texas at Austin
Austin, Texas 78712

June 11, 2010

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 what lessons we have learned since the introduction of these two packages and to propose a simple yet effective alternative. Preliminary performance results show the new solution achieves considerably better performance than the previously developed libraries.

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 [8, 5, 14, 31] and PLAPACK [32], were created in the mid-1990s, both of which provide a substantial part of the functionality offered by the widely used LAPACK library [3]. Both of these packages still each enjoy a loyal following.

One of the authors of the present paper contributed to the early design of ScaLAPACK [5, 15, 13, 12, 5, 14] and was the primarily architect of PLAPACK [33, 2, 27, 32, 6]. 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 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 processors and multi-threaded architectures (including SMP and multicore), yielding the FLAME project [21, 20], `libflame` library [34], and SuperMatrix runtime system for scheduling dense linear algebra algorithms to multicore architectures [10, 28]. 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 [26] and the recently announced Intel SCC research processor with 48 cores in one processor [24], the research comes full circle: distributed memory libraries will 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

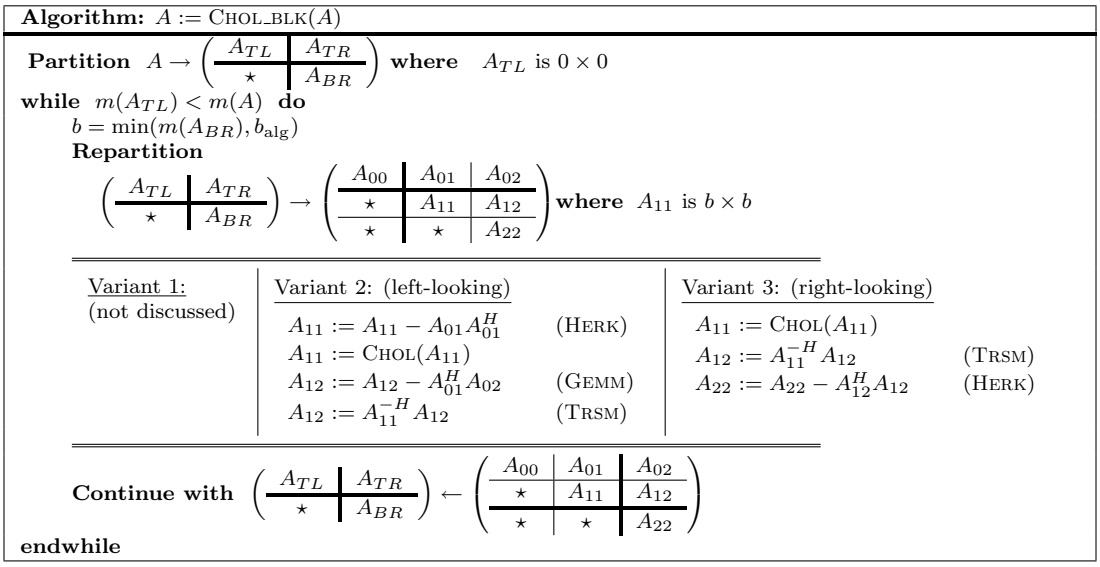

Figure 1: Blocked algorithms for computing the Cholesky factorization. (Note that the algorithm is for both real and complex valued matrices.)

based on more than 20 years of experience by one of the authors and a fresh look provided by the other authors, who more recently entered the field. 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 potential, and 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.

2 Of 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 processing nodes (nodes hereafter) using a two-dimensional data distribution [30, 22]. The p nodes 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 nodes, with an occasional collective communication that involves all nodes.

2.1 Motivating Example

In much of this paper, we will use the Cholesky factorization as our motivating example. Algorithms for this operation that lend themselves to parallelization are 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 node (s, t).

2.3 ScaLAPACK

In theory ScaLAPACK allows the distribution block size to be any choice of row and column dimension (including rectangular). In practice, the block size is chosen to be square. What is important is that 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). This is a problem since the algorithmic block size is currently often in the $128 - 256$ range, and the larger the distribution block size, the worse the load balance. If the algorithmic block size is decreased, then the local computation performs worse. In other words, there is a tension between wanting the distribution block size to be small versus wanting the algorithmic block size to be larger.

The benefit of linking the two is that, for example, the A_{11} block in the right-looking Cholesky factorization is owned by a single node, thus requiring only local computation on that node. After this A_{11} needs only be broadcast within the row of nodes that owns A_{12}, and the nodes in that row of nodes can then independently update A_{12}. Finally, A_{12} can be duplicated within rows and columns of nodes after which A_{22} can be updated independently on each node.

2.4 PLAPACK

In PLAPACK there is a notion of a vector distribution that induces the matrix distribution. Vectors are subdivided into subvectors of length b_{distr} which are wrapped in a cyclic fashion to all nodes. 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 $r \cdot b_{\text{distr}}$. This in turn means that algorithms that operate with the matrix are mildly nonscalable in the sense that if the number of nodes p gets large enough, efficiency will start to suffer even in the matrix is chosen to fill all of available memory. In PLAPACK the distribution and algorithmic block sizes are not linked so the distribution block size can be chosen to be small while the algorithmic blocksize can equal the block size that makes local computation efficient.

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

2.5 Elemental

In principle Elemental, like ScaLAPACK, can accomodate any distribution block size. However, unlike ScaLAPACK and like PLAPACK, the distribution block size is not linked to the algorithmic block size. Now, load balance is optimal when the distribution block size is as small as possible, leading to the choice to initially only support $b_{\text{distr}} = 1$, unlike PLAPACK which is not implemented to be efficient when $b_{\text{distr}} = 1$.

The insight to use $b_{\text{distr}} = 1$ is not new. On early distributed memory architectures such "elemental" distribution was the norm [22, 25]. But that was before the advent of cache based processors that favored blocked algorithms. In [23] it is noted that

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

This may have been prophetic but has not become relevant until recently. The reason is that the algorithmic block size used for blocked algorithms used to be related to the (square root of the) size of the L1 cache, which was relatively small. Kazushige Goto [18] showed that alternative 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 [24]. 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 true for dense matrix computations, consider the following observations regarding parallelization of blocked Cholesky factorization Variant 3:

- In the ScaLAPACK implementation A_{11} is factored by a single node after which it must be broadcast within the column of nodes that owns it after its factorization.
- If the matrix is distributed using $b_{\text{distr}} = 1$, then A_{11} must be gathered to at least one node if it is to be factored by only one node (which is beneficial since otherwise a lot of communication is necessary during that smaller computation).
- If done correctly, an allgather to all nodes is comparable in cost to the broadcast of A_{11} performed by ScaLAPACK.
- Thus, if $b_{\text{distr}} = 1$, A_{11} can be first gathered to all nodes and then redundantly factored (after which locally the updated values can be placed back in the original matrix since all nodes have a copy).

The point is that for the suboperation that factors A_{11} there is little or no price to be paid, in term of communication cost and computation in the critical path, if $b_{\text{distr}} = 1$.

Next, consider the update of A_{12}:

- In the ScaLAPACK implementation, A_{12} is updated by the row of nodes that owns it, requiring the broadcast of A_{11} within that row of nodes. Upon completion, the updated A_{12} is then broadcast within rows and columns of nodes.
- 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 $A_{12} := A_{11}^{-H} A_{12}$. This can be implemented as an all-to-all collective communication within columns (details go beyond the scope of this paper). After this, if A_{11} is already redundantly factored by each node, that update of A_{12} can happen completely in parallel (with all nodes participating).
- An allgather within rows and columns then duplicates the elements of A_{12} so that A_{22} can be updated in parallel, much like it is by ScaLAPACK except that ScaLAPACK would use, again, a broadcast.

It is important to realize that for large amounts of data an allgather is *cheaper* than a broadcast [9].

The point is that for little or no extra cost the update of A_{12} and subsequent communication of the result can be accomodated if $b_{\text{distr}} = 1$, after which the load balance during the update of A_{22} is much better, which can then yield higher performance.

2.6 Recap

If one understands that (virtually) all communications in dense matrix computations are collective in nature, then one realizes that partitioning and distributing using a small block size is not a problem because gathering data from many nodes incurs a communication cost that is comparable to the collective communications that are executed anyway by algorithms that assume a coarser partitioning.

3 Programmability

A major concern when designing Elemental was that the same code should support distributed memory parallelism at both the "exascale" (clusters with hundreds of thousands or even millions of cores) and distributed memory clusters 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 [1]:

> 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) [8], and a set of Basic Linear Algebra Communication Subprograms (BLACS) [4, 17] 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.

The question could be asked whether the code in Figure 2 could simply be modified to accomodate a block size of $b_{\text{distr}} = 1$ while allowing a large algorithmic block size b_{alg}. The answer is that it cannot:

- Major modifications would be needed that would violate the prime directive that the code must closely resemble the LAPACK equivalent routine.

  Consider what would need to be added to the code to perform the communication necessary to duplicate A_{11} to all nodes and to redistribute A_{12}.

- The communication layer of ScaLAPACK, the BLACS, inherently does not support the all-to-all and allgather communications that underly the described Cholesky factorization for a matrix that is distributed with $b_{\text{distr}} = 1$.

  There is a reason for this. The BLACS interface for communicating matrices was meant to closely resemble the BLAS that perform computation in LAPACK. Consider a representative call to the BLACS broadcast routine:

```
CALL DGEBS2D( ICONTXT, SCOPE, TOP, M, N, A, LDA )
```

```
      SUBROUTINE PDPOTRF( 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 20 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 PDPOTF2( 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 column panel of L using the triangular solver
*
               CALL PDTRSM( 'Right', UPLO, 'Transpose', 'Non-Unit',
     $                      N-J-JB+JA, JB, ONE, A, I, J, DESCA, A, I+JB,
     $                      J, DESCA )
*
*              Update the trailing matrix, A = A - L*L'
*
               CALL PDSYRK( UPLO, 'No Transpose', N-J-JB+JA, JB, -ONE,
     $                      A, I+JB, J, DESCA, ONE, A, I+JB, J+JB,
     $                      DESCA )
*
            END IF
   20    CONTINUE
```

Figure 2: Excerpt from ScaLAPACK Cholesky factorization.

Here `ICONTXT` describes the two-dimensional mesh of nodes, `SCOPE` indicates whether the broadcast involves rows, columns, or all nodes, `TOP` indicates the topology (algorithm) to be used for the broadcast, `M` and `N` indicate the local row and column dimensions of the matrix, which is stored locally at this node at address `A` with leading dimension `lda`. Now, consider what a call to allgather might look like. The indexing required how data being collected from all nodes is stored before and after the allgather operation is extremely difficult to specify. Most communications that inherently are encountered when accomodating a distribution with $b_{\text{distr}} = 1$ (or any distribution that divorces the algorithmic block size for the distribution block size) would be equally difficult to specify: gather, scatter, allgather, reduce-scatter, and all-to-all. It is for this reason[1] that the BLACS only included broadcast (`_xxBS2D` and `_xxBR2D`), global combine (`_xSUM2D`, `_xAMX2D`, and `_xAMN2D`), and point-to-point communication (`_xxSD2D`, `_xxRV2D`).

The point is that without a major redesign of ScaLAPACK, $b_{\text{alg}} \neq b_{\text{distr}} = 1$ cannot be accomodated by that package given the design contraints that were put in place early in the development of that package.

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

[1] We can say this with confidence since one of the authors of the current paper was the original designer of the BLACS [4, 11, 4, 17].

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++[2]. Other than that, the coding style resembles that used by `libflame`. Like its predecessors PLAPACK and libflame, it hides details regarding a matrix or vector in an object. 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. Imagine for a moment that there one of the occurances of `N-J-JB+JA` in Figure 2 was changed to `N-J-JB`. This would be a very hard error to track down.

Now, 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 nodes own a copy is achieved by

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

  which indicates that `A11_Star_Star` describes a matrix duplicated on all nodes, and

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

  which performs an allgather of the data, factors the matrix redundantly on all nodes, and then local substitutes the new values into the original (distributed) matrix.

- The parallel solve of A_{12} against the conjugate-transpose of the upper triangle of A_{11} 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

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

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

```
template<typename T>
void
elemental::lapack::internal::CholUVar3
( 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: Excerpt from Elemental Cholesky factorization.

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 three temporary distributions,

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

  which respectively describe an intermediate vector distribution and the two distributions needed to make the update of A_{22} local. The redistributions themselves are accomplished by the commands

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

  which perform a permutation of data among all nodes, an allgather of data within rows, and an allgather of data within columns, respectively. (Details of why and how these communications 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 (re)partitioning (indexing), distributions, communications, and local computations to be elegantly captured in code.

3.4 Recap

Regardless of whether someone prefers the code in Figure 2 over that in Figure 3, high-level decisions made during the early design stage of ScaLAPACK inherently prevent the kinds of communications that support an elemental distribution ($b_{\text{distr}} = 1$). The abstractions that are part of the Elemental framework do. Equally importantly: these abstractions hide details of how the underlying collective communications are implemented, allowing the code to be easily ported from a conventional cluster where MPI [29] is used to environments with other support for communicating between nodes.

4 Performance Experiments

The scientific computing community has always been willing to give up programmability if it meant attaining better performance. In this section we give preliminary performance numbers that suggest that one can have one's cake and eat it too.

4.1 Platform details

The performance experiments were carried out on The University of Texas at Austin Texas Advanced Computing Center's Ranger Supercomputer. At the time of this paper Ranger consisted of 3,936 nodes. Each node is a SunBlade x6420 blode running a 2.6.16 Linux kernel, with four AMD Opteron Quad-Core (2.3 GHz) 64-bit processors. Each core can perform four floating-point operations per clock cycle for a peak performance of 9.2 GLFOPS/core. The nodes are connected via a fill-CLOS InfiniBand interconnect providing a 1 GB/sec point-to-point bandwidth. Since this is a substantial resource we chose to only perform experiments with 15 nodes, for a total of 240 cores.

We compare the performance of a preliminary version of Elemental to ScaLAPACK Release 1.8 and PLAPACK Release R3.22. For our experiments we used MVAPICH2 Release 1.2 for MPI and for the BLAS

GotoBLAS Version 1.30. Each core hosted one MPI process, meaning that the machine was viewed as a 240 processor distributed memory architecture. For all tested implementationsntal the cores were viewed as a 16×15 mesh of MPI processes. Elemental uses the described elemental cyclic two-dimensional distribution of the matrices while for ScaLAPACK the distribution block size was always chosen to equal the algorithmic block size. For PLAPACK the distribution block size was chosen to equal 32. A simple experiment on a single core yielded an optimal block size of 224, which is the block size for which the rank-k that is at the core of many high-performance dense matrix computations achieves near-peak performance.

All computations were performed in double precision (64-bit) arithmetic.

4.2 Results

In Figures 4–7 we compare the performance attained by Elemental and ScaLAPACK implementations of various frequently encountered matrix computations. We limited the size of the problems we tests so as not to unnecessarily tie up the valuable resource that Ranger represents. We believe the presented data establishes the trend that can be expected had larger problems been used.

Level-3 BLAS ScaLAPACK, PLAPACK, and Elemental support a full implementation of the level-3 BLAS [16] operations. Here we discuss representative performance for these implementations. In Figure 4 we show performance attained when computing $C := AB$, the "no transpose-no transpose" case of the BLAS general matrix-matrix multiplication (`gemm`) operation. We believe that the reason that the ScaLAPACK performance is inferior is that that there are three main algorithmic variants for parallel matrix-matrix multiplication [19] and ScaLAPACK chooses the wrong algorithmic variant (although all three are implemented). In Figure 5 we show performance for solving $LX = B$ where L is a lower triangular matrix and X overwrites B. This operations is called the triangular solve with multiple right-hand sides (`trsm`). The large difference in performance is caused by the excessive communication required for their broadcast-based implementation.

Matrix factorization ScaLAPACK and PLAPACK include implementations of all three (one-sided) matrix factorizations: LU with partial pivoting, Cholesky, and QR factorization. Elemental as of this writing includes the first two, although implementation of QR factorization is a relatively straight-forward exercise, especially given that we have an implementation of reduction to tridiagonal form (discussed later).

In Figure 6 we report the performance for the Cholesky factorization. The "jagged" performance of the ScaLAPACK Cholesky factorization can be contributed to load-imbalance because of the relatively large distribution block size (which, recall, equals the algorithmic block size). It is interesting to note that performance is notably better near $m = 15 \times 16 \times 224 = 53760$. For Elemental we show performance for both the left-looking (var2) and right-looking (var3) variants. Overall the performance for this operation ramps up considerably slower because the ratio between communication and computation is less favorable.

In Figure 7 we report the performance for the LU factorization with partial pivoting. There are a number of reasons why the Elemental implementation outperforms the ScaLAPACK implementation: Better load balance because of the elemental distribution, the ability of Elemental to use all MPI processes to collaborate in the factorization of the "current panel" in contrast to the one column of processes used by the ScaLAPACK implementation, and a more efficient approach to pivoting rows outside the current panel. Details of how these different issues contribute to the better performance will be investigated in a future paper.

Two-sided factorization ScaLAPACK and PLAPACK include implementations of all three reductions to condensed form (two-sided factorization) operations: Reduction to tridiagonal, upperHessenberg, and bidiagonal form. Elemental as of this writing includes only the first, although implementation the other two is a relatively straight-forward exercise given the implementation of reduction to tridiagonal form.

In Figure 8 we report the performance for the reduction to tridiagonal form. Notice that 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 memory intensive.

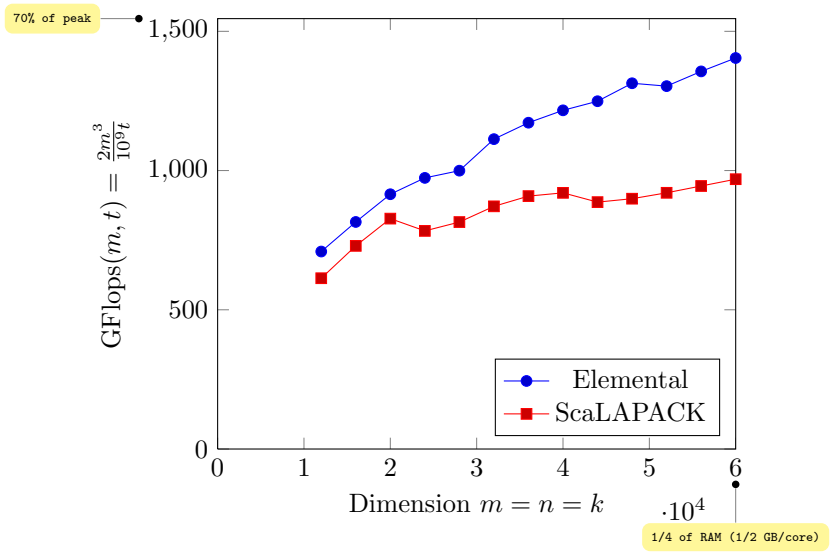

Figure 4: Performance of the `gemm` operation ($C = AB$) with square matrices on 240 cores.

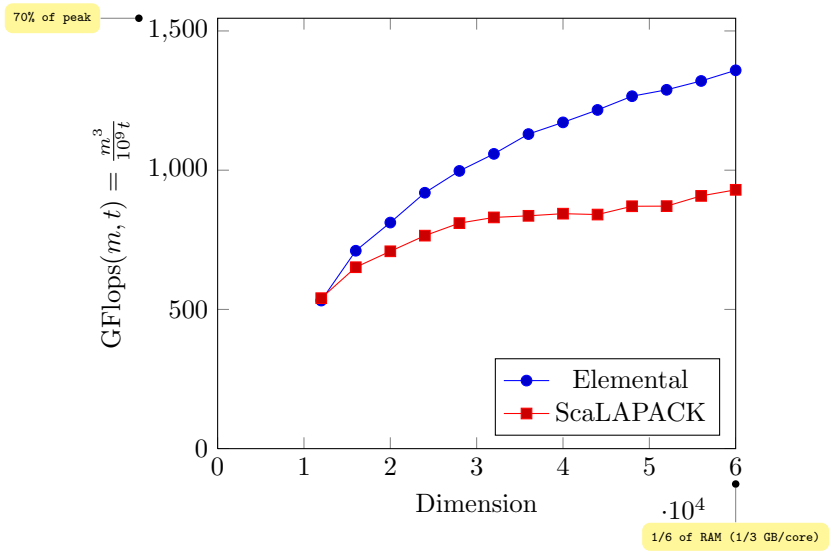

Figure 5: Performance of the `trsm` operation ($B = L^{-1}B$) with square matrices on 240 cores.

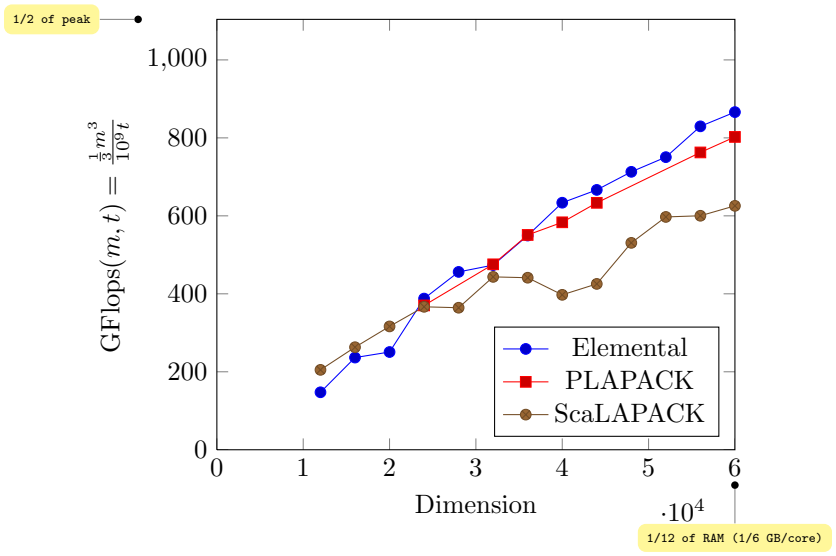

Figure 6: Performance of Cholesky factorization on 240 cores.

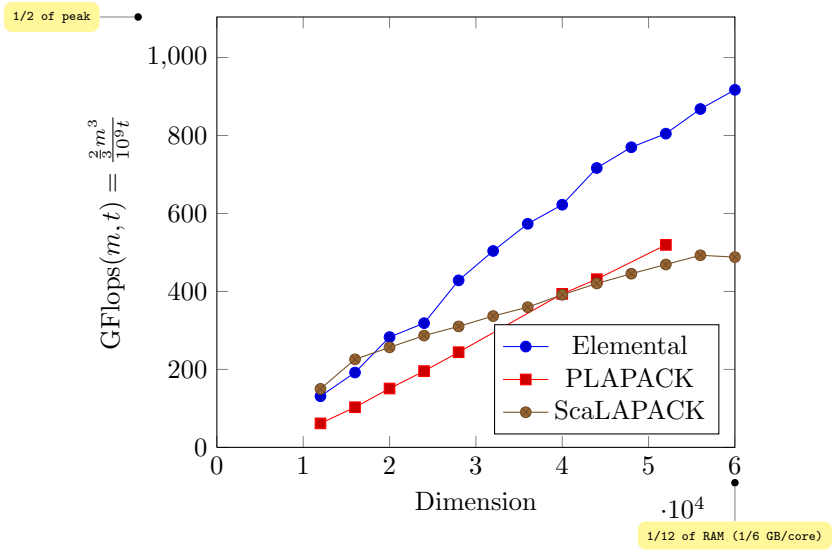

Figure 7: Performance of LU factorization with partial pivoting on 240 cores.

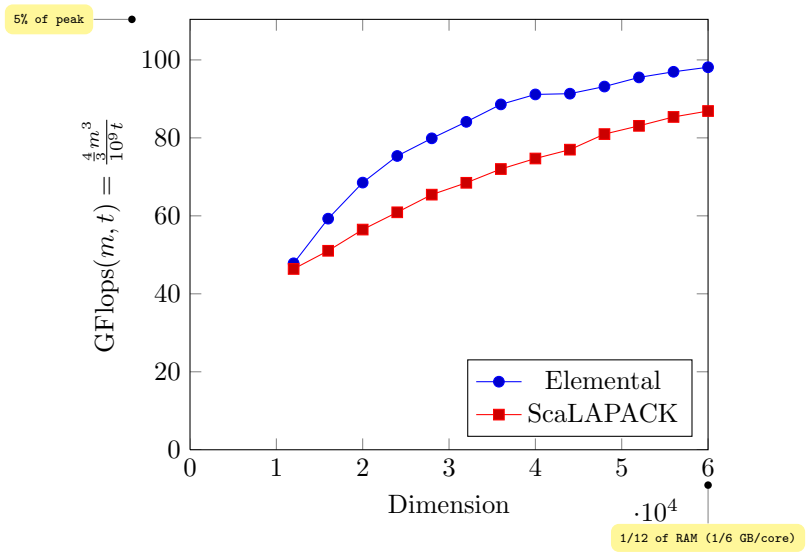

Figure 8: Performance of reduction to tridiagonal form on 240 cores.

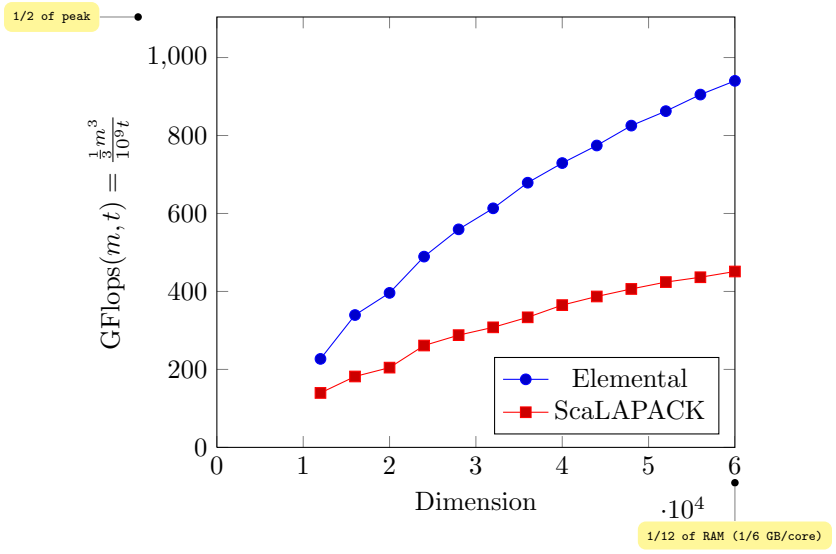

Figure 9: Performance of inversion of a triangular matrix 240 cores.

Inversion of a triangular matrix Inversion of a triangular matrix is an operation encountered when inverting a symmetric positive definite matrix. A thorough discussion of that operation and its parallelization can be found in [7]. The poor performance of the ScaLAPACK implementation is due to a call to a parallel triangular matrix-multiplication with n_b right-hand sides. While this operation is embarrassingly parallel with many right-hand sides, it is inherently unscalable if there are not a sufficient number of right-hand sides to distribute to each node.

4.3 Tuning

A somewhat legitimate criticism of the performance experiments reported in this section is that we did not extensively tune ScaLAPACK. The ONLY tuning we did for either package was to check if viewing the processes as a 15×16 mesh versus a 16×15 mesh was better. One would expect that there would be a considerable discrepency in performance since the target machine has 16 cores on a single node.

We believe that Elemental inherently requires little tuning: the elemental partitioning does a good job of balancing the matrices among processes for load balance and the collective communications that are inherently encountered in the implementations do not tend to favor row vs. column communications, meaning that an approximately square mesh of processes should perform well. Thus, we tend not to tune other than a very simple test to determine a good algorithmic block size.

ScaLAPACK on the other hand has many tunable parameters. The choice of distribution block size (which, recall, is tied to the algorithmic block size) is very much a function of the problem (matrix) size. If the matrix is small, a smaller block size should be chosen for better load balance, but a small block size impedes the performance of the local computations. Moreover, there tends to be an imbalance in the cost of communication within rows and columns. For example, Cholesky and LU factorizations in ScaLAPACK pipeline communication within rows while communication with columns is more synchronous in nature. (These choices of how to communicate themselves are parameters that can be tuned in ScaLAPACK.) Nonetheless, one would have expected that if the block size were chosen to be equal the best algorithmic block size, as the matrix size increases in the graphs, the plotted curves should have started to converge. One could argue that perhaps we did not examine large enough problem sizes, but it is undeniable that something good is happening with the performance of the Elemental implementations.

5 Conclusion

The point of this paper is to demonstrate once again 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.

One major impetus behind the creation of Elemental was the prospect of chips with many cores on a single chip where communication between the cores will be achieved through message-passing. Our project is funded in part by Intel to develop techniques for porting dense linear algebra libraries to Intel's experimental SCC processor [24], which consists of 48 (Pentium P54C) cores on a single chip that can communicate on-chip via communication buffers. While one approach to programming this chip is to treat it as a distributed memory architecture, for many reasons it is impractical to use MPI as the communication layer. As a result, Intel Labs created the experimental light-weight communication layer RCCE, which at the moment only supports synchronous communication between nodes, including a few collective communications. Fortunately, Elemental performs *all* communication via collective communication. Thus it turned out to be a relatively easy task (a matter of a few weeks of time for one of the authors of this paper) to port Elemental to RCCE. The purpose of the experimental processor is to examine programmability rather than raw performance and the successful port of Elemental to this unusual architecture demonstrates how well it addresses that issue. A future paper will report performance attained on that novel architecture.

Availability

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

Acknowledgments

This research was partially sponsored by NSF grant OCI-0850750, a grant from Microsoft, and a grant from Intel. Access to Ranger was arranged by the Texas Advanced Computing Center. As always, we thank the other members of the FLAME team for their support.

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).

References

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

[2] 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.

[3] 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.

[4] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic linear algebra communication subprograms. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 287–290. IEEE Computer Society Press, 1991.

[5] 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.

[6] Greg Baker, John Gunnels, Greg Morrow, Beatrice Riviere, and Robert van de Geijn. PLAPACK: High performance through high level abstraction. In *Proceedings of ICCP98*, 1998.

[7] Paolo Bientinesi, Brian Gunter, and Robert A. Van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Transactions on Mathematical Software*, 35(1).

[8] 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.

[9] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[10] 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.

[11] Jack Dongarra and Robert van de Geijn. Two dimensional basic linear algebra subprograms. LAPACK Working Note 37 CS-91-138, UTK, Oct. 1991.

[12] Jack Dongarra and Robert van de Geijn. A parallel dense linear solve library routine. In *Proceedings of the 1992 Intel Supercomputer Users' Group Meeting*, Oct. 1992.

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

[14] 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.

[15] Jack Dongarra, Robert van de Geijn, and David Walker. A look at scalable dense linear algebra libraries. In *Proceedings of Scalable High Performance Concurrent Computing '92*, April 27-29, 1992.

[16] 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.

[17] Jack J. Dongarra, Robert A. van de Geijn, and R. Clint Whaley. Two dimensional basic linear algebra communication subprograms. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, March 1993.

[18] 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.

[19] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.

[20] 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.

[21] John A. Gunnels and Robert A. van de Geijn. Formal methods for high-performance linear algebra libraries. In Ronald F. Boisvert and Ping Tak Peter Tang, editors, *The Architecture of Scientific Software*, pages 193–210. Kluwer Academic Press, 2001.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2001.

[28] 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.

[29] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

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

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

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

[33] 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.

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