(Parallel) Algorithms for Two-sided Triangular Solves and Matrix Multiplication

JACK POULSON, ROBERT A. VAN DE GEIJN, and JEFFREY BENNIGHOF,
The University of Texas at Austin

We discuss the parallel implementation of two operations, $A := L^{-1}AL^{-H}$ and $A := L^{H}AL$, where A is Hermitian and L is lower triangular. We use the FLAME formalisms to derive and represent a family of algorithms which are then implemented using Elemental, a new C++ library for distributed memory architectures. It is shown that, provided the right algorithm is chosen, excellent performance is attained on a large cluster.

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

General Terms: Algorithms, Performance

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

ACM Reference Format:
Poulson, J., van de Geijn, R. A., Bennighof, J. 2012. Parallel Algorithms for Two-sided Triangular Solve and Matrix Multiplication ACM Trans. Math. Softw. 0, 0, Article 0 (0000), 12 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The two-sided triangular solve is an important operation that can be used to reduce a (well-conditioned) generalized Hermitian-definite eigenvalue problem, $Ax = \lambda Bx$ where A is Hermitian and B is Hermitian positive-definite, to a standard Hermitian eigenvalue problem [Sears et al. 1998; Anderson et al. 1999; Poulson et al.]. It is also utilized in a stable implementation of a new algorithm for factoring (with pivoting) an indefinite symmetric matrix A into LTL^T, where L is lower triangular and T is tridiagonal [Ballard et al.]. The two-sided triangular matrix multiplication is encountered when transforming the generalized Hermitian-definite eigenvalue problem $ABx = \lambda x$ to a standard Hermitian eigenvalue problem [Anderson et al. 1999; Poulson et al.]. This paper gives a thorough treatment of algorithms for these operations and discusses their implementation in our new Elemental library [Poulson et al. ; Poulson 2011] for dense linear algebra on distributed memory architectures, an alternative to ScaLAPACK [Blackford et al. 1997].

A second contribution of this paper is in its concise, start-to-finish demonstration of the FLAME methodology for representing, deriving, and implementing dense linear algebra algorithms. The FLAME project has pursued techniques for developing

Authors' addresses: Jack Poulson, Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712, jack.poulson@gmail.com. Robert A. van de Geijn, Department of Computer Science, The University of Texas at Austin, Austin, TX 78712, rvdg@cs.utexas.edu. Jeffrey Bennighof, Department of Aerospace Engineering and Engineering Mechanics, The University of Texas at Austin, Austin, TX 78712, bennighof@mail.utexas.edu.

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

dense linear algebra libraries for more than a decade, funded by the National Science Foundation and industry. The hallmarks of the approach are:

— **A notation for representing dense linear algebra algorithms that avoids indices and facilitates the comparing and contrasting of related algorithms.** A journal paper on parallel inversion of a general matrix [Quintana et al. 2001] was among the first papers to employ the FLAME notation for representing dense linear algebra algorithms. In that paper, it was shown that the new Gauss-Jordan based algorithm could be viewed as the classical three-step approach to inverting a matrix, merged into one loop. As a result, the stability analysis for the classic approach carried over to the new algorithm. It was also employed in a multitude of our papers published in the ACM Transactions on Mathematical Software and elsewhere. In this paper, the notation is again used to compare and contrast algorithms.
— **A methodology for deriving algorithms hand-in-hand with their proof of correctness.** The FLAME methodology [Gunnels 2001; Gunnels et al. 2001] has yielded a "worksheet" for deriving algorithms [Bientinesi et al. 2005a] that facilitates systematic and mechanical tools for automatically deriving algorithms [Bientinesi 2006]. A book [van de Geijn and Quintana-Ortí 2008] breaks the process down into steps that even undergraduates who have limited familiarity with linear algebra can follow. It is our belief that everyone who works in the field of dense linear algebra should be familiar with this work. The current paper shows how the methodology yields a family of algorithms for the studied operations.
— **Application Programming Interfaces (APIs) that allow algorithms to be cleanly translated into code.** We believe that code should closely reflect how algorithms are naturally represented [Bientinesi et al. 2005b]. Elemental uses such an API. A representative code excerpt can be found in the first journal paper on the Elemental library [Poulson et al.]. In a companion video [van de Geijn 2011], it is demonstrated how an API for use with Matlab [Moler 1980], *FLAME@lab* [Bientinesi et al. 2005b], can be used to rapidly implement the algorithms discussed in this paper.
— **A modern alternative for LAPACK.** The notation, methodology, and APIs have been used to develop a library, `libflame` [Van Zee et al. 2009; Van Zee 2009], that already encompasses the functionality of most of the Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979; Dongarra et al. 1988; Dongarra et al. 1990], and LAPACK [Anderson et al. 1999]. It seemless also facilitates algorithms-by-blocks that can target multicore [Quintana-Ortí et al. 009b] and multiGPU (and other accelerated) architectures [Igual et al. 2011]. The algorithms discussed in this paper are also available in `libflame`.

Thus, this paper also illustrates the benefits of a body of work that has been published in the ACM Transactions on Mathematical Software and elsewhere over the last decade.

2. ALGORITHMS FOR COMPUTING THE TWO-SIDED TRIANGULAR SOLVE OPERATION

In this section, we derive algorithms for computing $C := L^{-1}AL^{-H}$, overwriting the lower triangular part of Hermitian matrix A with the lower triangular part of Hermitian matrix C.

Derivation. We give the minimum information required so that those familiar with the FLAME methodology can understand how the algorithms were derived. Those not familiar with the methodology can simply take the resulting algorithms—presented in Figures 2 and 3—on face value and move on to the discussion at the end of this section.

Using $\hat{A}$ to denote the input state of the matrix A, and $\wedge$ for the logical AND operator, we express the computation $C := L^{-1}AL^{-H}$ by the constraint that $A = C \wedge LCL^H = \hat{A}$ upon completion of the program. This constraint is known as the *postcondition* in the FLAME methodology.

Next, we form the *Partitioned Matrix Expression* (PME), which can be viewed as a recursive definition of the operation. For this, we partition the matrices so that

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right), \quad C \rightarrow \left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right), \quad \text{and} \quad L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), \tag{1}$$

where A_{TL}, C_{TL}, and L_{TL} are square submatrices and $\star$ denotes the parts of the Hermitian matrices that are neither stored nor updated. Substituting these partitioned matrices into the postcondition yields

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) \wedge$$

$$\underbrace{\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right)\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)^H = \left(\begin{array}{c|c} \hat{A}_{TL} & \star \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)}_{\left(\begin{array}{c|c} L_{TL}C_{TL}L_{TL}^H = \hat{A}_{TL} & \star \\ \hline L_{BR}C_{BL} = \hat{A}_{BL}L_{TL}^{-H} - L_{BL}C_{TL} & \begin{array}{rl} L_{BR}C_{BR}L_{BR}^H = & \hat{A}_{BR} - L_{BL}C_{TL}L_{BL}^H \\ & - L_{BL}C_{BL}^H L_{BR}^H - L_{BR}C_{BL}L_{BL}^H \end{array} \end{array}\right)}.$$

This expresses all conditions that must be satisfied upon completion of the computation, in terms of the submatrices. The bottom-right quadrant can be further manipulated into

$$\begin{aligned} L_{BR}C_{BR}L_{BR}^H &= \hat{A}_{BR} - L_{BL}C_{TL}L_{BL}^H - L_{BL}C_{BL}^H L_{BR}^H - L_{BR}C_{BL}L_{BL}^H \\ &= \hat{A}_{BR} - L_{BL}\underbrace{\left(\frac{1}{2}C_{TL}L_{BL}^H + C_{BL}^H L_{BR}^H\right)}_{W_{BL}^H} - \underbrace{\left(\frac{1}{2}L_{BL}C_{TL} + L_{BR}C_{BL}\right)}_{W_{BL}} L_{BL}^H \end{aligned}$$

using a standard trick that casts three rank-k updates into a single symmetric rank-$2k$ update. Now, the PME can be rewritten as

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) \wedge Y_{BL} = L_{BL}C_{TL} \wedge W_{BL} = L_{BR}C_{BL} - \frac{1}{2}Y_{BL}$$
$$\wedge \left(\begin{array}{c|c} L_{TL}C_{TL}L_{TL}^H = \hat{A}_{TL} & \star \\ \hline L_{BR}C_{BL} = \hat{A}_{BL}L_{TL}^{-H} - Y_{BL} & L_{BR}C_{BR}L_{BR}^H = \hat{A}_{BR} - (L_{BL}W_{BL}^H + W_{BL}L_{BL}^H) \end{array}\right).$$

The next step of the methodology identifies *loop invariants* for algorithms. A loop invariant is a predicate that expresses the state of a matrix (or matrices) before and after each iteration of the loop. In the case of this operation, there are many such loop invariants. However, careful consideration for maintaining symmetry in the intermediate update and avoiding unnecessary computation leaves the five in Figure 1.

The methodology finishes by deriving algorithms that maintain these respective loop invariants. The resulting blocked algorithms are given in Figures 2 and 3 where Variant k corresponds to Loop Invariant k. Unblocked algorithms result if the block size is chosen to equal 1.

Discussion. All algorithms in Figures 2 and 3 incur a cost of about n^3 flops where n is the matrix size. There is a useful rule of thumb for determining which operations in the

Loop Invariant 1

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} C_{TL} & \star \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$$

Loop Invariant 2

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} C_{TL} & \star \\ \hline \hat{A}_{BL}L_{TL}^{-H} & \hat{A}_{BR} \end{array}\right)$$

Loop Invariant 3

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} C_{TL} & \star \\ \hline \hat{A}_{BL}L_{TL}^{-H} & \hat{A}_{BR} \end{array}\right) \wedge \left(\begin{array}{c|c} Y_{TL} & \\ \hline Y_{BL} & Y_{BR} \end{array}\right) = \left(\begin{array}{c|c} & \\ \hline L_{BL}C_{TL} & \end{array}\right)$$

Loop Invariant 4

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} C_{TL} & \star \\ \hline \hat{A}_{BL}L_{TL}^{-H} - L_{BL}C_{TL} & \hat{A}_{BR} - (L_{BL}W_{BL}^{H} + W_{BL}L_{BL}^{H}) \end{array}\right)$$

Loop Invariant 5

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & \hat{A}_{BR} - (L_{BL}W_{BL}^{H} + W_{BL}L_{BL}^{H}) \end{array}\right)$$

Fig. 1. Five loop invariants for computing $A := L^{-1}AL^{-H}$.

algorithms in this paper asymptotically require the most work: given the partitionings

$$\left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \quad \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \quad \text{and} \quad \left(\begin{array}{c|c|c} C_{00} & \star & \star \\ \hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right),$$

the operations that involve at least one highlighted submatrix contribute to an $O(n^3)$ (highest order) cost term while the others contribute to lower order terms. Thus, first and foremost, it is important that the highlighted operations in Figures 2 and 3 attain high performance.

On sequential architectures, all of the highlighted operations *can* attain high performance [Goto and van de Geijn 008a; Goto and van de Geijn 008b]. However, as we will demonstrate, there is a notable difference on parallel architectures. As was already pointed out in a paper by Sears, Stanley, and Henry [Sears et al. 1998], it is the parallel triangular solves with b right-hand sides (TRSM), $A_{10} := A_{10}L_{00}^{-H}$ in Variant 1 and $A_{21} := L_{22}^{-1}A_{21}$ in Variant 5, that inherently do *not* parallelize well yet account for about one third of the flops for Variants 1 and 5. The reason is that inherent dependencies exist within the TRSM operation, the details of which go beyond the scope of this paper. All of the other highlighted operations can, in principle, asymptotically attain near-peak performance when correctly parallelized on an architecture with reasonable communication [van de Geijn and Watts 1997; Chtchelkanova et al. 1997; Gunnels et al. 1998; van de Geijn 1997]. Thus, Variants 1 and 5 cast a substantial fraction of computation in terms of an operation that does not parallelize well, in contrast to Variants 2, 3, and 4. Variant 3 has the disadvantage that intermediate result Y_{BL} must be stored. (In the algorithm we show Y for all algorithms, but only Y_{10} or Y_{21} are needed for Variants 1, 2, 4, and 5.)

In Section 4 we will see that Variant 4 attains the highest performance. This is because its most computationally intensive operations parallelize most naturally when targeting distributed memory architectures. Variant 2 might be a good choice when

```
Algorithm: $A := L^{-1}AL^{-H}$ and $A := L^H AL$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $L \rightarrow \left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right)$, $Y \rightarrow \left(\begin{array}{c|c} Y_{TL} & Y_{TR} \\ \hline Y_{BL} & Y_{BR} \end{array}\right)$
    where $A_{TL}$, $L_{TL}$, and $Y_{TL}$ are $0 \times 0$.
while $m(A_{TL}) < m(A)$ do
    Determine block size $b$
    Repartition
    $\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$, $\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$,
    $\left(\begin{array}{c|c} Y_{TL} & 0 \\ \hline Y_{BL} & Y_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} Y_{00} & 0 & 0 \\ \hline Y_{10} & Y_{11} & 0 \\ \hline Y_{20} & Y_{21} & Y_{22} \end{array}\right)$
        where $A_{11}$, $L_{11}$, and $Y_{11}$ are $b \times b$
```

Variant 4 for $L^{-1}AL^{-H}$ (Section 2)	Variant 4 for $L^H AL$ (Section 3)
$A_{10} := L_{11}^{-1} A_{10}$	$Y_{10} := A_{11} L_{10}$
$A_{20} := A_{20} - L_{21} A_{10}$ (GEMM)	$A_{10} := W_{10} = A_{10} + \frac{1}{2} Y_{10}$
$A_{11} := L_{11}^{-1} A_{11} L_{11}^{-H}$	$A_{00} := A_{00} + (A_{10}^H L_{10} + L_{10}^H A_{10})$ (HER2K)
$Y_{21} := L_{21} A_{11}$	$A_{10} := A_{10} + \frac{1}{2} Y_{10}$
$A_{21} := A_{21} L_{11}^{-H}$	$A_{10} := L_{11}^H A_{10}$
$A_{21} := W_{21} = A_{21} - \frac{1}{2} Y_{21}$	$A_{11} := L_{11}^H A_{11} L_{11}$
$A_{22} := A_{22} - (L_{21} A_{21}^H + A_{21} L_{21}^H)$ (HER2K)	$A_{20} := A_{20} + A_{21} L_{10}$ (GEMM)
$A_{21} := A_{21} - \frac{1}{2} Y_{21}$	$A_{21} := A_{21} L_{11}$

```
    Continue with
    $\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$, $\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$,
    $\left(\begin{array}{c|c} Y_{TL} & 0 \\ \hline Y_{BL} & Y_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} Y_{00} & 0 & 0 \\ \hline Y_{10} & Y_{11} & 0 \\ \hline Y_{20} & Y_{21} & Y_{22} \end{array}\right)$
endwhile
```

Fig. 2. Blocked Variants 4 for computing $A := L^{-1}AL^{-H}$ and $A := L^H AL$. All blocked variants result by inserting the commands in Figure 3.

implementing an out-of-core algorithm, since its most expensive computations (which are highlighted) require the bulk of data (A_{00} and A_{20}) to be read but not written.

3. ALGORITHMS FOR COMPUTING THE TWO-SIDED MATRIX MULTIPLICATION OPERATION

In this section, we derive algorithms for computing $C := L^H AL$, overwriting the lower triangular part of Hermitian matrix A.

Derivation. We once again give the minimum information required so that those familiar with the FLAME methodology understand how the algorithms were derived.

The postcondition for this operation is given by $A = L^H \hat{A} L$ where, again, $\hat{A}$ represents the input matrix A. We again partition the matrices as in Eqn. (1). Substituting

$L^{-1}AL^{-H}$	L^HAL
Variant 1 $Y_{10} := L_{10}A_{00}$ (HEMM) $A_{10} := A_{10}L_{00}^{-H}$ (TRSM) $A_{10} := W_{10} = A_{10} - \frac{1}{2}Y_{10}$ $A_{11} := A_{11} - (A_{10}L_{10}^H + L_{10}A_{10}^H)$ $A_{11} := L_{11}^{-1}A_{11}L_{11}^{-H}$ $A_{10} := A_{10} - \frac{1}{2}Y_{10}$ $A_{10} := L_{11}^{-1}A_{10}$	Variant 1 $Y_{21} := A_{22}L_{21}$ (HEMM) $A_{21} := A_{21}L_{11}$ $A_{21} := W_{21} = A_{21} + \frac{1}{2}Y_{21}$ $A_{11} := L_{11}^H A_{11} L_{11}$ $A_{11} := A_{11} + (A_{21}^H L_{21} + L_{21}^H A_{21})$ $A_{21} := A_{21} + \frac{1}{2}Y_{21}$ $A_{21} := L_{22}^H A_{21}$ (TRMM)
Variant 2 $Y_{10} := L_{10}A_{00}$ (HEMM) $A_{10} := W_{10} = A_{10} - \frac{1}{2}Y_{10}$ $A_{11} := A_{11} - (A_{10}L_{10}^H + L_{10}A_{10}^H)$ $A_{11} := L_{11}^{-1}A_{11}L_{11}^{-H}$ $A_{21} := A_{21} - A_{20}L_{10}^H$ (GEMM) $A_{21} := A_{21}L_{11}^{-H}$ $A_{10} := A_{10} - \frac{1}{2}Y_{10}$ $A_{10} := L_{11}^{-1}A_{10}$	Variant 2 $A_{10} = L_{11}^H A_{10}$ $A_{10} = A_{10} + L_{21}^H A_{20}$ (GEMM) $Y_{21} = A_{22}L_{21}$ (HEMM) $A_{21} = A_{21}L_{11}$ $A_{21} = A_{21} + \frac{1}{2}Y_{21}$ $A_{11} = L_{11}^H A_{11} L_{11}$ $A_{11} = A_{11} + (A_{21}^H L_{21} + L_{21}^H * A_{21})$ $A_{21} = A_{21} + \frac{1}{2}Y_{21}$
Variant 3 $A_{10} := W_{10} = A_{10} - \frac{1}{2}Y_{10}$ $A_{11} = A_{11} - (A_{10}L_{10}^H + L_{10}A_{10}^H)$ $A_{11} = L_{11}^{-1}A_{11}L_{11}^{-H}$ $A_{21} = A_{21} - A_{20}L_{10}^H$ (GEMM) $A_{21} = A_{21}L_{11}^{-H}$ $A_{10} = A_{10} - \frac{1}{2}Y_{10}$ $A_{10} = L_{11}^{-1}A_{10}$ $Y_{20} = Y_{20} + L_{21}A_{10}$ (GEMM) $Y_{21} = L_{21}A_{11}$ $Y_{21} = Y_{21} + L_{20}A_{10}^H$ (GEMM)	Variant 3 This variant performs $O(n^3)$ additional computations and is therefore not included.
Variant 4 $A_{10} := L_{11}^{-1}A_{10}$ $A_{20} := A_{20} - L_{21}A_{10}$ (GEMM) $A_{11} := L_{11}^{-1}A_{11}L_{11}^{-H}$ $Y_{21} := L_{21}A_{11}$ $A_{21} := A_{21}L_{11}^{-H}$ $A_{21} := W_{21} = A_{21} - \frac{1}{2}Y_{21}$ $A_{22} := A_{22} - (L_{21}A_{21}^H + A_{21}L_{21}^H)$ (HER2K) $A_{21} := A_{21} - \frac{1}{2}Y_{21}$	Variant 4 $Y_{10} := A_{11}L_{10}$ $A_{10} := W_{10} = A_{10} + \frac{1}{2}Y_{10}$ $A_{00} := A_{00} + (A_{10}^H L_{10} + L_{10}^H A_{10})$ (HER2K) $A_{10} := A_{10} + \frac{1}{2}Y_{10}$ $A_{10} := L_{11}^H A_{10}$ $A_{11} := L_{11}^H A_{11} L_{11}$ $A_{20} := A_{20} + A_{21}L_{10}$ (GEMM) $A_{21} := A_{21}L_{11}$
Variant 5 $A_{11} := L_{11}^{-1}A_{11}L_{11}^{-H}$ $Y_{21} := L_{21}A_{11}$ $A_{21} := A_{21}L_{11}^{-H}$ $A_{21} := W_{21} = A_{21} - \frac{1}{2}Y_{21}$ $A_{22} := A_{22} - (L_{21}A_{21}^H + A_{21}L_{21}^H)$ (HER2K) $A_{21} := A_{21} - \frac{1}{2}Y_{21}$ $A_{21} := L_{22}^{-1}A_{21}$ (TRSM)	Variant 5 $Y_{10} := A_{11}L_{10}$ $A_{10} := A_{10}L_{00}$ (TRMM) $A_{10} := W_{10} = A_{10} + \frac{1}{2}Y_{10}$ $A_{00} := A_{00} + (A_{10}^H L_{10} + L_{10}^H A_{10})$ (HER2K) $A_{10} := A_{10} + \frac{1}{2}Y_{10}$ $A_{10} := L_{11}^H A_{10}$ $A_{11} := L_{11}^H A_{11} L_{11}$

Fig. 3. All algorithms corresponding to the Invariants in Figures 1 and 4 for both $A := L^{-1}AL^{-H}$ and $A := L^HAL$.

Loop Invariant 1 $\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL}^H \hat{A}_{TL} L_{TL} + (W_{BL}^H L_{BL} + L_{BL}^H W_{BL}) & \star \\ \hline L_{BR}^H(\hat{A}_{BL} L_{TL} + \hat{A}_{BR} L_{BL}) & \hat{A}_{BR} \end{array}\right)$
Loop Invariant 2 $\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL}^H \hat{A}_{TL} L_{TL} + (W_{BL}^H L_{BL} + L_{BL}^H W_{BL}) & \star \\ \hline \hat{A}_{BL} L_{TL} + \hat{A}_{BR} L_{BL} & \hat{A}_{BR} \end{array}\right)$
Loop Invariant 3 $\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL}^H \hat{A}_{TL} L_{TL} & \star \\ \hline \hat{A}_{BL} L_{TL} & \hat{A}_{BR} \end{array}\right) \wedge \left(\begin{array}{c\|c} Y_{TL} & \\ \hline Y_{BL} & Y_{BR} \end{array}\right) = \left(\begin{array}{c\|c} & \\ \hline \hat{A}_{BR} L_{BL} & \end{array}\right)$
Loop Invariant 4 $\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL}^H \hat{A}_{TL} L_{TL} & \star \\ \hline \hat{A}_{BL} L_{TL} & \hat{A}_{BR} \end{array}\right)$
Loop Invariant 5 $\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL}^H \hat{A}_{TL} L_{TL} & \star \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$

Fig. 4. Five loop invariants for computing $A := L^H A L$.

these partitioned matrices into the postcondition yields the PME

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L_{TL}^H \hat{A}_{TL} L_{TL} + (W_{BL}^H L_{BL} + L_{BL}^H W_{BL}) & \star \\ \hline L_{BR}^H(\hat{A}_{BL} L_{TL} + \hat{A}_{BR} L_{BL}) & L_{BR}^H \hat{A}_{BR} L_{BR} \end{array}\right),$$

where $W_{BL} = \hat{A}_{BL} L_{TL} + \frac{1}{2}\hat{A}_{BR} L_{BL}$. Letting $Y_{BL} = \hat{A}_{BR} L_{BL}$ yields five loop invariants for this operation that exploit and maintain symmetry. These loop invariants are listed in Figure 4 while the corresponding blocked algorithms were already given in Figures 2 and 3. One of the loop invariants yields an algorithm that incurs $O(n^3)$ additional computation and we do not give the related algorithm.

Discussion. For this operation, in principle, all of the highlighted suboperations can be implemented to be scalable on parallel architectures.

4. PERFORMANCE EXPERIMENTS

We now show the performance attained by the different variants on a large distributed memory parallel architecture. We compare implementations that are part of the Elemental library to the implementations that are part of `netlib` ScaLAPACK version 1.8.

Target Architectures. 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 (13.6×10^9 floating-point operations per second) per node using double-precision arithmetic. 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 one midplane (512 compute nodes, or 2048 cores), which has an aggregate theoretical peak of just under 7 TFlops (7×10^{12} floating-point operations per second). For this configuration, the X, Y, and Z inter-node dimensions of the torus are each of length 8, and the intra-node dimension, T, is of size 4.

```
Variant 4 (Elemental)
$A_{10} := L_{11}^{-1} A_{10}$
$A_{20} := A_{20} - L_{21} A_{10}$   (GEMM)
$A_{11} := L_{11}^{-1} A_{11} L_{11}^{-H}$
$Y_{21} := L_{21} A_{11}$
$A_{21} := A_{21} L_{11}^{-H}$
$A_{21} := W_{21} = A_{21} - \frac{1}{2} Y_{21}$
$A_{22} := A_{22} - (L_{21} A_{21}^H + A_{21} L_{21}^H)$   (HER2K)
$A_{21} := A_{21} - \frac{1}{2} Y_{21}$
```

```
Variant 4 (ScaLAPACK)
$G_{21} := L_{21}$
$R_{21} := A_{21}$
$S_{10} := A_{10}$
$R_{11} := \text{tril}(A_{11})$
$G_{21} := -G_{21} L_{11}^{-1}$
$R_{21} := R_{21} + \frac{1}{2} G_{21} A_{11}$
$A_{22} := A_{22} + G_{21} R_{21}^H + R_{21} G_{21}^H$   (HER2K)
$A_{20} := A_{20} + G_{21} S_{10}$   (GEMM)
$A_{21} := A_{21} + G_{21} R_{11}$
$A_{10} := L_{11}^{-1} A_{10}$
$C_{11} := \text{tril}(A_{11})$                      ⎫
$\text{triu}(C_{11}) := \text{tril}(C_{11})^H$        ⎪
$C_{11} := L_{11}^{-1} C_{11}$                       ⎬ $A_{11} := L_{11}^{-1} A_{11} L_{11}^{-H}$
$C_{11} := C_{11} L_{11}^{-1}$                       ⎪
$\text{tril}(A_{11}) := \text{tril}(C_{11})$          ⎭
$A_{21} := A_{21} L_{11}^{-H}$
```

Fig. 5. Operations performance by Variant 4 for $A := L^{-1}AL^{-H}$ in Elemental (left) and ScaLAPACK (right).

Since collective communication over so-called ‘irregular’ communicators (those that do not span entire dimensions of the torus) cannot fully exploit Blue Gene/P’s hardware, the performance of both Elemental and ScaLAPACK was tested over all process grid configurations which resulted in regular row and column communicators that were sufficiently close in size (i.e., all process grid configurations whose rows and columns each spanned two dimensions of the torus). In every experiment, both Elemental and ScaLAPACK performed best with the $(Z, T) \times (X, Y)$ decomposition, which is to say that the Z and T dimensions of the torus form the columns of the two-dimensional process grid, while the X and Y dimensions make up the rows.

ScaLAPACK. ScaLAPACK was developed in the 1990s as a distributed memory dense matrix library meant to mirror the style of LAPACK, and thus the majority of the library was written in Fortran-77, but a significant portion was written in C. It uses a two-dimensional block cyclic data distribution, meaning that p MPI processes are viewed as a logical $r \times c$ mesh and the matrices are partitioned into $b_r \times b_c$ blocks (submatrices) that are then cyclically wrapped onto the mesh. It is almost always the case that $b_r = b_c = b_{\text{distr}}$, where b_{distr} is the distribution block size. The vast majority of the library is layered so that the algorithms are coded in terms of parallel implementations of the BLAS. An important restriction for ScaLAPACK is that the algorithmic block size b in Figure 2 is tied to the distribution block size.

The ScaLAPACK routines `p[sd]sygst` and `p[cz]hegst` implement Variant 5 from Figure 2 when used to compute $A := L^{-1}AL^{-H}$ and Variant 5 from Figure 2 when computing $A := L^H AL$. In addition, for $A := L^{-1}AL^{-H}$, a vastly more efficient algorithm (Variant 4 from Figure 2) is implemented as the routines `p[sd]syngst` and `p[cz]hengst`. These faster routines currently only support the case where the lower triangular part of A is stored. Routines `p[sd]syngst` and `p[cz]hengst` appear to have been derived from the work by Sears et al. There are a few subtle differences between the algorithm used by those routines and Variant 4 in Figure 2, as illustrated in Figure 5. Expansion of each of the dark gray updates in terms of the states of A and L at the beginning of the iteration reveals that they are identical. Likewise, the light gray update on the right is merely an expanded version of the update $A_{11} := L_{11}^{-1} A_{11} L_{11}^{-H}$ that could have been performed more simply via a call to LAPACK’s `[sd]sygs2` or `[cz]hegs2` routines.

Elemental. We think of Elemental as a modern replacement for ScaLAPACK and PLAPACK [van de Geijn 1997]. It is coded in C++ in a style that resembles the FLAME/C API used to implement the `libflame` library. Elemental uses a two-dimensional elemental distribution that can be most easily described as the same distribution used by ScaLAPACK except that (a) $b_{\text{distr}} = 1$, and (b) the algorithmic block size is not restricted by the distribution block size. Elemental uses a more flexible layering so that calls to global BLAS-like operations can be easily fused, which means that communication overhead can be somewhat reduced by combining communications from within separate calls to BLAS-like operations. See [Poulson et al.] for details.

Tuning. Both packages were run with one MPI process per core using IBM's non-threaded ESSL library for sequential BLAS calls. For the sake of an apples-to-apples comparison, performance of hybrid implementations is not given for either package. Both packages were tested over a wide range of typical block sizes; ScaLAPACK was tested with block sizes $\{16, 24, 32, 48, 64\}$, while the block sizes $\{64, 80, 96, 112, 128\}$ were investigated for Elemental. Only the results for the best-performing block size for each problem size are reported in the graphs. In the case of ScaLAPACK, the algorithmic and distribution block sizes are equal, since this is a restriction of the library. In the case of Elemental, the distribution is elemental (block size of one) and the block size refers to the algorithmic block size.

Results. In Figure 6 and 7 we report performance of the different variants for the studied computations. We do so for the case where only the lower triangular part of A is stored, since this case is the most commonly used and it exercises ScaLAPACK's fastest algorithms (the more efficient routines `p[sd]syngst` and `p[cz]hengst` are only implemented for the lower triangular storage case). In order to lower the required amount of compute time, all experiments were performed with real double-precision (64-bit) data.

In Figure 6 performance for computing $A := L^{-1}AL^{-H}$ is given. As expected, the variants that cast a significant part of the computation in terms of a triangular solve with multiple right-hand sides (TRSM) attain significantly worse performance. Variant 4 performs best, since it casts most computation in terms of a symmetric (Hermitian) rank-$2k$ update ($A_{22} - (L_{21}A_{21}^H + A_{21}L_{21}^H)$) and general rank-$k$ update, ($A_{20} - L_{21}A_{10}$), which parallelize more naturally. Variants 2 and 3 underperform since symmetric (Hermitian) or matrix-panel multiplies (matrix multiply where the result matrix is narrow), like $L_{10}A_{00}$, $A_{21} - A_{20}L_{10}^H$, and $Y_{21} + L_{20}A_{10}^H$, require local contributions to be summed (reduced) across processes, a collective communication that often requires significantly more time than the simpler duplications needed for rank-k updates. Also, the local matrix-panel multiply that underlies these parallel operations is often less optimized than the local rank-k update that underlies the parallel implementations of the symmetric (Hermitian) rank-$2k$ and general rank-k updates. For Variant 4, $b_{\text{distr}} = b_{\text{alg}} = 32$ was typically optimal for ScaLAPACK, while $b_{\text{alg}} = 112$ was the almost always the best blocksize for Elemental.

We believe that ScaLAPACK's Variant 4 is slower than Elemental's Variant 4 for two reasons: (1) ScaLAPACK's implementation is layered on top of the PBLAS and therefore redundantly communicates data, and (2) ScaLAPACK has a hard-coded block size for the local updates of their parallel symmetric (Hermitian) rank-$2k$ update that is therefore not a parameter that is easily tuned in that package (and we did not tune it for that reason). Thus, part of the increased performance attained by parallel implementations stems from the proper choice of algorithm, part is the result of implementation details, and part comes from how easily the implementation can be tuned.

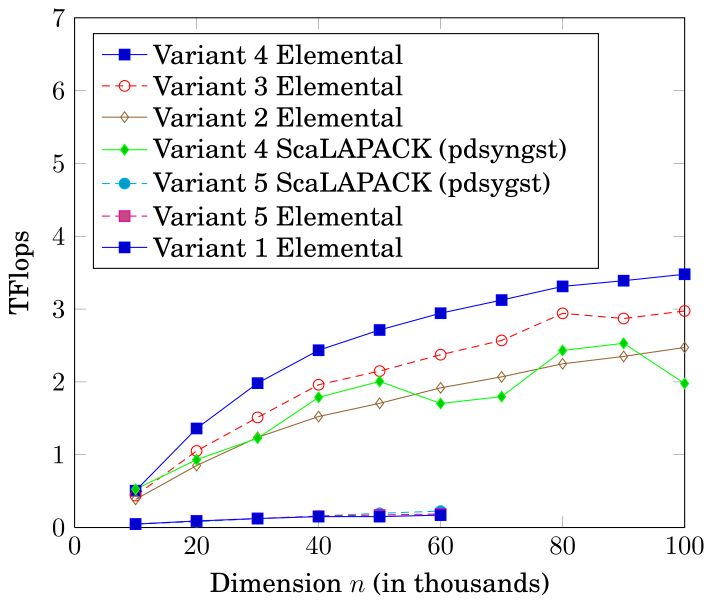

Fig. 6. Performance of the various implementations for $A := L^{-1}AL^{-H}$ on 2048 cores of Blue Gene/P. The top of the graph represents the theoretical peak of this architecture. (The three curves for Variants 1 and 5, which cast substantial computation in terms of a parallel TRSM, essentially coincide near the bottom of the graph.) The legend lists the implementations from fastest to slowest for the largest problem size.

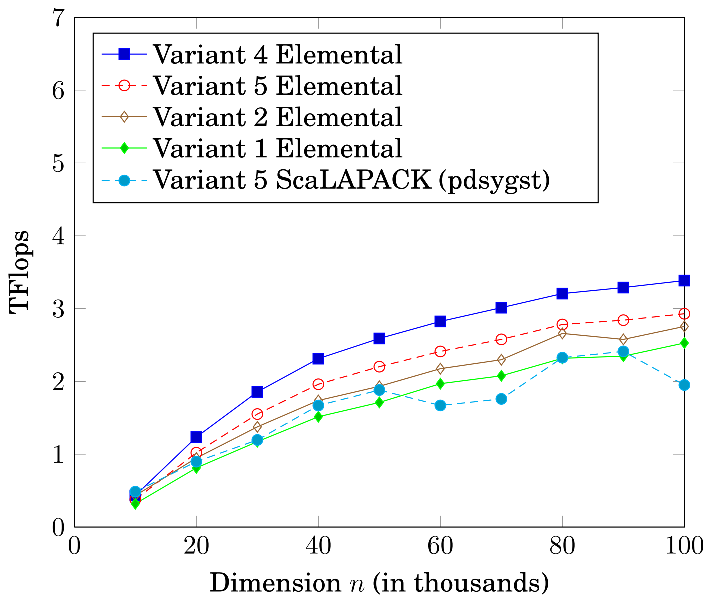

Fig. 7. Performance of the various implementations for $A := L^{H}AL$ on 2048 cores of Blue Gene/P. The legend lists the implementations from fastest to slowest for the largest problem size.

In Figure 7 performance for computing $A := L^H AL$ is given. As can be expected given the above discussion, Variant 4, which casts the bulk of computation in terms of a symmetric (Hermitian) rank-$2k$ update and general rank-k update, attains the best performance.

5. CONCLUSION

We have systematically derived and presented a multitude of algorithms for two-sided triangular solves and matrix multiplication. While the concept of avoiding the unscalability in the traditional algorithm for $A := L^{-1}AL^H$ was already discussed in the paper by Sears et al., we give a clear derivation of that algorithm as well as several other new algorithmic possibilities. For $A := L^H AL$ we similarly present several algorithms, including one that is different from that used by ScaLAPACK and achieves superior performance.

The performance improvements of Elemental over ScaLAPACK are not the central message of this paper. Instead, we argue that a systematic method for deriving algorithms combined with a highly-programmable library has allowed us to thoroughly explore the performance of a wide variety of algorithms. Still, Elemental outperforms ScaLAPACK even when the same algorithm is used and hence Elemental is clearly faster on this architecture.

Exercises

To fully appreciate the FLAME methodology for deriving algorithms, one must derive a few algorithms oneself. On the accompanying website [van de Geijn 2011], exercises related to the operations in this article have been posted.

Acknowledgements

This research was partially sponsored by National Science Foundation grant CCF-0917167 and a grant from Microsoft. Jack Poulson was partially supported by a CAM fellowship. We gratefully acknowledge the use of Argonne National Laboratory's BlueGene/P system.

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

REFERENCES

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

BALLARD, G., DEMMEL, J., DRUINSKY, A., PELED, I., SCHWARTZ, O., AND TOLEDO, S. A communication-avoiding symmetric-indefinite factorization. Unpublished manuscript.

BIENTINESI, P. 2006. Mechanical derivation and systematic analysis of correct linear algebra algorithms. Ph.D. thesis, Department of Computer Sciences, The University of Texas. Technical Report TR-06-46. September 2006.

BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005a. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft. 31,* 1, 1–26.

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

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

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

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND DUFF, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft. 16,* 1, 1–17.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND HANSON, R. J. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft. 14,* 1, 1–17.

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

GOTO, K. AND VAN DE GEIJN, R. 2008b. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Soft. 35,* 1, 1–14.

GUNNELS, J., LIN, C., MORROW, G., AND VAN DE GEIJN, R. 1998. 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)*. 110–116.

GUNNELS, J. A. 2001. A systematic approach to the design and analysis of parallel dense linear algebra algorithms. Ph.D. thesis, Department of Computer Sciences, The University of Texas.

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

IGUAL, F. D., CHAN, E., QUINTANA-ORT, E. S., QUINTANA-ORT, G., VAN DE GEIJN, R. A., AND ZEE, F. G. V. 2011. The flame approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing* 0, –.

LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft. 5,* 3, 308–323.

MOLER, C. B. 1980. MATLAB– an interactive matrix laboratory. Tech. Rep. Technical Report 369, Department of Mathematics and Statistics, University of New Mexico.

POULSON, J. 2011. Elemental, distributed-memory dense linear algebra library. `http://code.google.com/p/elemental`.

POULSON, J., MARKER, B., HAMMOND, J. R., ROMERO, N. A., AND VAN DE GEIJN, R. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Soft.*. In revision. Available from `http://www.cs.utexas.edu/users/flame/pubs/Elemental1.pdf`.

QUINTANA, E. S., QUINTANA, G., SUN, X., AND VAN DE GEIJN, R. 2001. A note on parallel matrix inversion. *SIAM J. Sci. Comput. 22,* 5, 1762–1771.

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

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

VAN DE GEIJN, R. 2011. Demos of FLAME related tools. `http://www.cs.utexas.edu/users/flame/Movies.html`.

VAN DE GEIJN, R. AND WATTS, J. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience 9,* 4, 255–274.

VAN DE GEIJN, R. A. 1997. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press.

VAN DE GEIJN, R. A. AND QUINTANA-ORTÍ, E. S. 2008. *The Science of Programming Matrix Computations*. `http://www.lulu.com/content/1911788`.

VAN ZEE, F. G. 2009. `libflame`*: The Complete Reference*. `www.lulu.com`.

VAN ZEE, F. G., CHAN, E., VAN DE GEIJN, R., QUINTANA-ORTÍ, E. S., AND QUINTANA-ORTÍ, G. 2009. Introducing: The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering 11,* 6, 56–62.

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