Parallel Algorithms for Reducing the Generalized Hermitian-Definite Eigenvalue Problem

JACK POULSON
The University of Texas at Austin
and
ROBERT A. VAN DE GEIJN
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, that are important to the solution of dense generalized Hermitian-definite eigenproblems. We use the FLAME formalisms to derive and represent a family of algorithms and implement these using Elemental, a new C++ library for distributed memory architectures that may become the successor to the widely-used ScaLAPACK and PLAPACK libraries. 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

1. INTRODUCTION

The generalized Hermitian-definite eigenvalue problem occurs in one of three forms: $Ax = \lambda Bx$, $ABx = \lambda x$, and $BAx = \lambda x$, where A is Hermitian and B is Hermitian and positive-definite. The usual solution strategy for each of these problems is to exploit the positive-definiteness of B in order to compute its Cholesky factorization and transform the problem into a standard Hermitian eigenvalue problem, solve the Hermitian eigenproblem, and then backtransform the eigenvectors if necessary.

In particular, the following steps are performed for the $Ax = \lambda Bx$ case: (1) Compute the Cholesky factorization $B = LL^H$. (2) Transform $C := L^{-1}AL^{-H}$ so that $Ax = \lambda Bx$ is transformed to $Cy = \lambda y$ with $y = L^{-H}x$. (3) Reduce C to tridiagonal form: $C := Q_C T Q_C^H$ where Q_C is unitary and T is tridiagonal. (4) Compute the Spectral Decomposition of T: $T := Q_T D Q_T^H$ where Q_T is unitary and D is diagonal. (5) Form $X := LQ_C Q_T$ so that $AX = BXD$. Then the generalized eigenvalues

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.

can be found on the diagonal of D and the corresponding generalized eigenvectors as the columns of X. In this paper, we focus on Step 2. For the other two forms of the generalized Hermitian eigenvalue problem this step becomes $C := L^H AL$ and $C := LAL^H$. Typically C overwrites matrix A. The present paper demonstrates how Elemental benefits from the FLAME methodology [Gunnels and van de Geijn 2001; Gunnels et al. 2001; Quintana-Ortí and van de Geijn 2003; van de Geijn and Quintana-Ortí 2008; Bientinesi et al. 2008] by allowing families of algorithms for dense matrix computations to be systematically derived and presented in a clear, concise fashion. This results in the most complete exposition to date of algorithms for computing $L^{-1}AL^{-H}$ and $L^H AL$.

While a family of algorithms and their parallelization for these operations is the primary focus of this paper, we also consider this paper the second in a series of papers related to the Elemental library for dense matrix computations on distributed memory architectures. The first paper [Poulson et al.] gave a broad overview of the vision behind the design of the Elemental library and performance comparisons between ScaLAPACK [Choi et al. 1992] and Elemental for a representative subset of operations. Since the comparison is not the main focus of that paper, the reader is left wondering as to how much of the improvement in performance is due to algorithm choice rather than implementation. This paper answers that question for the discussed operations. It thus adds to the body of evidence that Elemental may be a worthy successor to ScaLAPACK.

This paper is organized as follows. In Sections 2 and 3 we derive algorithms for computing $L^{-1}AL^{-H}$ and $L^H AL$, respectively. In Section 4 we present results from performance experiments on a large cluster. Related work is discussed in Section 5 and concluding remarks are given in the conclusion.

2. ALGORITHMS FOR COMPUTING $A := L^{-1}AL^{-H}$

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 understand how the algorithms that are discussed later were derived. Those not familiar with the methodology can simply take the resulting algorithms—presented in Figure 2—on face value and move on to the discussion at the end of this section.

First, we reformulate the computation $C := L^{-1}AL^{-H}$ as the constraint $A = C \wedge LCL^H = \hat{A}$ where $\wedge$ denotes the logical AND operator. This constraint expresses that A is to be overwritten by matrix C, where C satisfies the given constraint in which $\hat{A}$ represents the input matrix A. 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

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

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 represents 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 to cast three rank-k updates into a single symmetric rank-k 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 tabulated in Figure 1.

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

Discussion. All algorithms in Figure 2 incur a cost of about n^3 flops where n is the matrix size. A quick way to realize where the algorithms in Figure 2 spend most of their time is to consider the partitionings

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

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. Four loop invariants for computing $A := L^{-1} A L^{-H}$. Loop Invariant 2 is maintained by the new algorithm described in this paper while Loop Invariant 4 is maintained by the algorithm currently used by LAPACK and ScaLAPACK.

and to note that operations that involve at least one operand that is highlighted contribute to an $O(n^3)$ (highest order) cost term while the others contribute to an $O(bn^2)$ term. Thus, first and foremost, it is important that the highlighted operations in Figure 2 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. In particular, 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 1/3 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 (as was, for example, already noted in [Sears et al. 1998]). 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, 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 implementing an out-of-core algorithm, since the highlighted computations

```
Algorithm: $A := L^{-1}AL^{-H}$

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 $0 \times 0$. Initially, $Y = 0$.
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 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 2
    $Y_{10} := \frac{1}{2}L_{10}A_{00}$   (HEMM)
    $A_{10} := W_{10} = -Y_{10} + A_{10}$
    $A_{11} := A_{11} - (A_{10}L_{10}^{H} + L_{10}A_{10}^{H})$
    $A_{11} := L_{11}^{-1}A_{11}L_{11}^{-H}$
    $A_{21} := A_{21} - A_{20}L_{10}^{H}$   (GEMM)
    $A_{21} := A_{21}L_{11}^{-H}$
    $A_{10} := L_{11}^{-1}(A_{10} - Y_{10})$

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

  Variant 5
    $A_{11} := L_{11}^{-1}A_{11}L_{11}^{-H}$
    $Y_{21} := L_{21}A_{11}$
    $A_{21} := W_{21} = A_{21}L_{11}^{-H} - \frac{1}{2}Y_{21}$
    $A_{22} := A_{22} - (L_{21}A_{21}^{H} + A_{21}L_{21}^{H})$   (HER2K)
    $A_{21} := A_{21} - \frac{1}{2}Y_{21}$
    $A_{21} := L_{22}^{-1}A_{21}$   (TRSM)

  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 algorithms for computing $A := L^{-1}AL^{-H}$.

Loop Invariant 1

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

Loop Invariant 2

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

Loop Invariant 3

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

Fig. 3. Four loop invariants for computing $A := L^H AL$.

for it require the bulk of data (A_{00} and A_{20}) to be read but not written.

3. ALGORITHMS FOR COMPUTING $A := L^H AL$

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 (discussed later) 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 partition the matrices so that

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

where A_{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 the PME

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

where $W_{TL} = \hat{A}_{BL} L_{TL} + \frac{1}{2} \hat{A}_{BR} L_{BL}$. Letting $Y_{BL} = \hat{A}_{BR} L_{BL}$ yields three loop invariants for this operation that exploit and maintain symmetry without adding cost. These loop invariants are listed in Figure 3 while the corresponding blocked algorithms are shown in Figure 4.

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 implementation of this operation that is part of `netlib` ScaLAPACK version 1.8.

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

  Variant 1
    $Y_{10} := A_{11} L_{10}$
    $A_{10} := A_{10} L_{00}$   (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)
    $A_{10} := L_{11}^H (A_{10} + \frac{1}{2} Y_{10})$
    $A_{11} := L_{11}^H A_{11} L_{11}$

  Variant 2
    $Y_{10} := A_{11} L_{10}$
    $A_{10} := W_{10} = A_{10} + \frac{1}{2} Y_{10}$
    $A_{00} := A_{00} + (A_{10}^H L_{10} + L_{10}^H A_{10})$   (HER2K)
    $A_{10} := L_{11}^H (A_{10} + \frac{1}{2} Y_{10})$
    $A_{11} := L_{11}^H A_{11} L_{11}$
    $A_{20} := A_{20} + A_{21} L_{10}$   (GEMM)
    $A_{21} := A_{21} L_{11}$

  Variant 3
    $Y_{21} := A_{22} L_{21}$   (HEMM)
    $A_{21} := W_{21} = A_{21} L_{11} + \frac{1}{2} Y_{21}$
    $A_{11} := L_{11}^H A_{11} L_{11}$
    $A_{11} := A_{11} + A_{21}^H L_{21} + L_{21}^H A_{21}$
    $A_{21} := A_{21} + \frac{1}{2} Y_{21}$
    $A_{21} := L_{22}^H A_{21}$   (TRMM)

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

Fig. 4. Blocked algorithms for computing $A := L^H AL$.

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 dimensions of the torus are each 8 and the optimal decomposition into a two-dimensional topology was found to be $Z \times (X, Y)$ in every case, with the fourth intra-node dimension, T, combined with the Z dimension to produce a 32×64 process grid.

ScaLAPACK. ScaLAPACK [Dongarra et al. 1992; Anderson et al. 1992; Choi et al. 1992; Blackford et al. 1997] was developed in the 1990s as a distributed

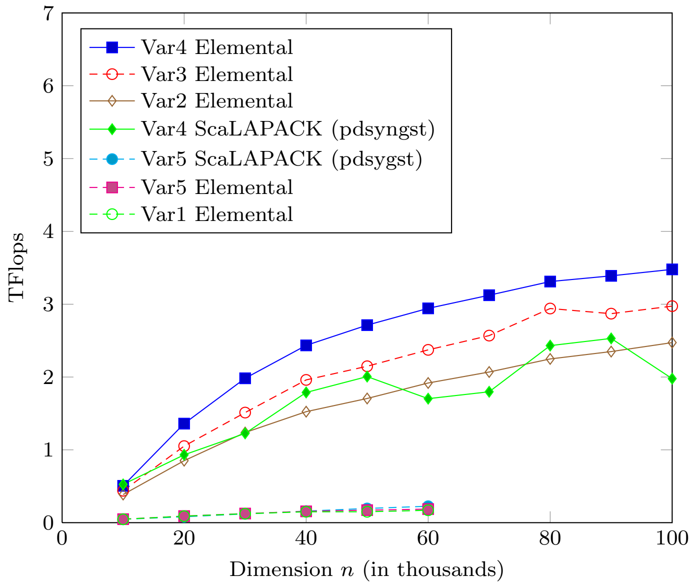

Fig. 5. Performance of the various implementations for $A := L^{-1}AL^{-H}$ on 2048 cores of Blue Gene/P. The top of the graph represents the theoretical peak of this architecture. (The three curves for Variants 1 and 5, which cast substantial computation in terms of a parallel TRSM, essentially coincide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

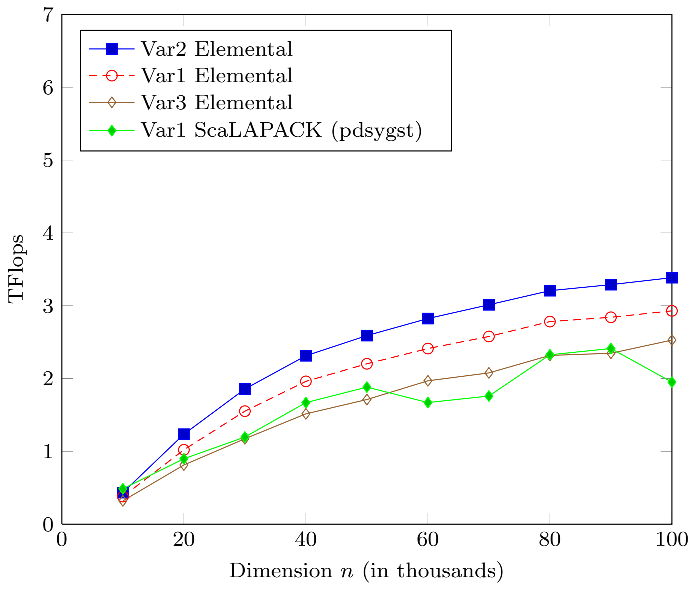

Fig. 6. Performance of the various implementations for $A := L^{H}AL$ on 2048 cores of Blue Gene/P. The top of the graph represents the theoretical peak of this architecture.

memory dense matrix library coded in Fortran-77 in the style of LAPACK. 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. Almost always is 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 Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979; Dongarra et al. 1988; Dongarra et al. 1990]. 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 1 from Figure 4 when computing $A := L^H AL$. In addition, 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.

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 [Bientinesi et al. 2005] used to implement the `libflame` library [Van Zee 2009] (a modern replacement for LAPACK, coded in C). For the uninitiated, this means that the routines in Elemental that implement the algorithm in Figure 2 resemble those algorithms. Elemental uses a two-dimensional elemental distribution that can be most easily described as the same distribution used by ScaLAPACK except that $b_{\text{distr}} = 1$. 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 regarding Elemental.

Tuning. As mentioned above, all variants of Elemental and ScaLAPACK were tested on one midplane of a Blue Gene/P using a 32×64 process grid. Both packages were run with one MPI process per core using IBM's non-threaded ESSL library for sequential BLAS calls.[1] 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.

[1]Elemental also efficiently supports SMP+MPI parallelism while ScaLAPACK does not seem to benefit from this kind of hybrid parallelism on this architecture [Poulson et al.]. For the sake of an apples-to-apples comparison, performance of hybrid implementations is not given for either package.

Results. In Figure 5 and 6 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 5 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.

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). The point is that, in general, 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.

In Figure 6 performance for computing $A := L^HAL$ is given. As can be expected given the above discussion, Variant 2, 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. RELATED WORK

In writing this paper, we first derived all of the discussed algorithms via the FLAME methodology. Upon finishing, we felt that algorithms like Variants 3 and 4 in Figure 2 would be nearly impossible to discover without the systematic method described in Sections 2 and 3. When we subsequently searched the literature, we were surprised to find what is essentially Variant 4 in Figure 2 in a paper by Sears, Stanley, and Henry [Sears et al. 1998] and also in the ScaLAPACK routines `p[sd]syngst` and `p[cz]hengst`.

The algorithm in the paper by Sears et al. computes $L^{-H}AL^{-1}$ after having computed a somewhat unusual version of the Cholesky factorization: $B = L^HL$. The paper gives few clues about what motivated the discovery of the algorithm.

Variant 4 (Elemental)	Variant 4 (ScaLAPACK)	
$A_{10} := L_{11}^{-1} A_{10}$	$G_{21} := L_{21}$	
$A_{20} := A_{20} - L_{21} A_{10}$ (gemm)	$R_{21} := A_{21}$	
$A_{11} := L_{11}^{-1} A_{11} L_{11}^{-H}$	$S_{10} := A_{10}$	
$Y_{21} := L_{21} A_{11}$	$R_{11} := \mathrm{tril}(A_{11})$	
$A_{21} := W_{21} = A_{21} L_{11}^{-H} - \frac{1}{2} Y_{21}$	$G_{21} := -G_{21} L_{11}^{-1}$	
$A_{22} := A_{22} - (L_{21} A_{21}^H + A_{21} L_{21}^H)$ (her2k)	$R_{21} := R_{21} + \frac{1}{2} G_{21} A_{11}$	
$A_{21} := A_{21} - \frac{1}{2} Y_{21}$	$A_{22} := A_{22} + G_{21} R_{21}^H + R_{21} G_{21}^H$ (her2k)	
	$A_{20} := A_{20} + G_{21} S_{10}$ (gemm)	
	$A_{21} := A_{21} + G_{21} R_{11}$	
	$A_{10} := L_{11}^{-1} A_{10}$	
	$C_{11} := \mathrm{tril}(A_{11})$	
	$\mathrm{triu}(C_{11}) := \mathrm{tril}(C_{11})^H$	
	$C_{11} := L_{11}^{-1} C_{11}$	$A_{11} := L_{11}^{-1} A_{11} L_{11}^{-H}$
	$C_{11} := C_{11} L_{11}^{-1}$	
	$\mathrm{tril}(A_{11}) := \mathrm{tril}(C_{11})$	
	$A_{21} := A_{21} L_{11}^{-H}$	

Fig. 7. Operations performance by Variant 4 in Elemental (left) and ScaLAPACK (right).

The algorithm itself is presented as a double-nested loop around vector operations so that even level-2 BLAS operations are not obvious. No blocked algorithm is given in the paper. It is merely mentioned that the unblocked algorithm that is given motivated a blocked version. The one hint at how the algorithm came to be is the mentioning of "Gauss transforms". We conclude that the authors used the following insight: When factoring a square matrix A into its LU factorization, $A = LU$, one can view the process as

$$\underbrace{\hat{L}_{n-1} \cdots \hat{L}_0}_{L^{-1}} A = U, \text{ where } \hat{L}_k = \left(\begin{array}{c|c|c} I_{k \times k} & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & -l_{21}^{(k)} & I \end{array} \right)$$

is a Gauss transform. (Here $I_{k \times k}$ denotes a $k \times k$ identity matrix.) There is a simple relation between L and these Gauss transforms in that the $l_{21}^{(k)}$ equal the corresponding subdiagonal elements of matrix L. Thus, the inverse of a *unit* lower triangular matrix L can be viewed as a sequence of Gauss transforms that are easily identified from the matrix L. Thus, if one wishes to compute $L^{-1} A L^{-H}$ one can instead compute $\left(\hat{L}_{n-1} \cdots \left(\hat{L}_0 A \hat{L}_0^H \right) \cdots \hat{L}_{n-1}^H \right)$. The paper contains a few details beyond this insight, like how to modify this algorithm for the L from the Cholesky factorization, which is not unit lower triangular, and how to generalize this to a blocked algorithm. However, these topics are only barely discussed in [Sears et al. 1998], and we choose to omit discussion here since we have an alternative way of deriving the algorithm. Hopefully, this gives the reader some idea of how the authors of [Sears et al. 1998] may have derived their algorithm as an alternative to our derivation based on the FLAME methodology.

It appears that the routines `p[sd]syngst` and `p[cz]hengst` in ScaLAPACK were derived from the work described above. There are a few subtle differences between the algorithm used by those routines and Variant 4 in Figure 2, as illustrated in Figure 7. The two algorithms are essentially the same: 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.

6. CONCLUSION

We have systematically derived and presented a multitude of algorithms for the transformation of the generalized Hermitian-definite eigenvalue problem to the standard Hermitian eigenvalue problem. While the concept of avoiding the unscalability in the traditional algorithm for $A := L^{-1}AL^H$ was preceded by the work of Sears et al., we give a clear derivation of this prior work 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 that 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 approaches. The resulting performance improvements can be viewed as a consequence of this approach.

Availability

The Elemental library can be found at
`http://code.google.com/p/elemental`.
This library includes all discussed variants for single, double, complex, and double complex datatypes, and for updating either the upper or lower triangular parts of A. The algorithms are also implemented as part of the `libflame` library [Van Zee 2009] (a modern replacement library for LAPACK) including algorithm-by-blocks that can be scheduled for parallel execution on multi-threaded and/or multi-GPU accelerated architectures via the SuperMatrix runtime system [Quintana-Ortí et al. 2009; Chan and Igual 2010; Quintana-Ortí et al. 2009].

Acknowledgements

This research was partially sponsored by a grant from Microsoft. We gratefully acknowledge the use of Argonne National Laboratory's BlueGene/P system.

REFERENCES

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

Bientinesi, P., Gunter, B., and van de Geijn, R. A. 2008. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Transactions on Mathematical Software 35,* 1 (July), 3:1–3:22.

Bientinesi, P., Quintana-Ortí, E. S., and van de Geijn, R. A. 2005. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft. 31,* 1 (March), 27–59.

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

Chan, E. and Igual, F. D. 2010. Runtime data flow graph scheduling of matrix computations with multiple hardware accelerators. FLAME Working Note #50 TR-10-36, The University of Texas at Austin, Department of Computer Sciences. October.

Choi, J., Dongarra, J. J., Pozo, R., and Walker, D. W. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Comput. Soc. Press, 120–127.

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 (Sept.), 837–857.

Dongarra, J., van de Geijn, R., and Walker, D. April 27-29, 1992. A look at scalable dense linear algebra libraries. In *Proceedings of Scalable High Performance Concurrent Computing '92*.

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 (March), 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 (March), 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 (May).

Goto, K. and van de Geijn, R. 2008b. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw. 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., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software 27,* 4 (December), 422–455.

Gunnels, J. A. and van de Geijn, R. A. 2001. Formal methods for high-performance linear algebra libraries. In *The Architecture of Scientific Software*, R. F. Boisvert and P. T. P. Tang, Eds. Kluwer Academic Press, 193–210.

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 (Sept.), 308–323.

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 Transactions on Mathematical Software*. In revision. Available from `http://www.cs.utexas.edu/users/flame/pubs/Elemental1.pdf`.

Quintana-Ortí, E. S. and van de Geijn, R. A. 2003. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Trans. Math. Soft. 29,* 2 (June), 218–243.

Quintana-Ortí, G., Igual, F. D., Quintana-Ortí, E. S., and van de Geijn, R. 2009. Solving dense linear systems on platforms with multiple hardware accelerators. In *ACM SIGPLAN 2009 symposium on Principles and practices of parallel programming (PPoPP'09)*. 121–129.

Quintana-Ortí, G., Quintana-Ortí, E. S., van de Geijn, R. A., Van Zee, F. G., and Chan, E. 2009. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software 36,* 3 (July), 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. and Watts, J. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience 9,* 4 (April), 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`.

Received Month Year; revised Month Year; accepted Month Year