Parallel Algorithms for Reducing the Generalized Hermitian-Definite Eigenvalue Problem

FLAME Working Note #56

Jack Poulson*
Robert A. van de Geijn†
Jeffrey Bennighof‡

February 11, 2011

Abstract

We discuss the parallel implementation of two operations, $A := L^{-1}AL^{-H}$ and $A := L^{H}AL$, that are important to the solution of dense generalized Hermitian-definite eigenproblems. Here A is Hermitian and L is lower triangular. 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.

1 Introduction

The generalized Hermitian-definite eigenvalue problem usually occurs in one of two forms: $Ax = \lambda Bx$ and $ABx = \lambda x$, where A is Hermitian and B is Hermitian and positive-definite. The typical 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 eigenvalues can be found on the diagonal of D and the corresponding eigenvectors as the columns of X. In this paper, we focus on Step 2. For the other form of the generalized Hermitian-definite eigenvalue problem this step becomes $C := L^H AL$. Typically C overwrites matrix A. The present paper demonstrates how Elemental benefits from the FLAME methodology [15, 14, 18, 24, 2] 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 [17] gave a broad overview of the vision behind the design of the Elemental library and performance comparisons between ScaLAPACK [6] and Elemental for a representative subset of operations. Since the comparison is not the main focus of that

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

†Department of Computer Science, The University of Texas at Austin, Austin, TX 78712, rvdg@cs.utexas.edu

‡Department of Aerospace Engineering and Engineering Mechanics, The University of Texas at Austin, Austin, TX 78712, bennighof@mail.utexas.edu

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 and PLAPACK [23].

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

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 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.

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 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 to cast three rank-k updates into a single symmetric rank-$2k$ update. Now, the PME can be rewritten as

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

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

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

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

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

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 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. A quick way to realize where the algorithms spend most of their time is to consider the partitionings

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

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 Figures 2 and 3 attain high performance.

On sequential architectures, all of the highlighted operations *can* attain high performance [11, 12]. 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 [21], 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. All of the other highlighted operations can, in principle, asymptotically attain near-peak performance when correctly parallelized on an architecture with reasonable communication [22, 7, 13, 23]. 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.

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

Figure 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.

$L^{-1}AL^{-H}$	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) $A_{21} := A_{21}L_{11}$ $A_{21} := W_{21} = A_{21} + \frac{1}{2}Y_{21}$ $A_{11} := L_{11}^HA_{11}L_{11}$ $A_{11} := A_{11} + (A_{21}^HL_{21} + L_{21}^HA_{21})$ $A_{21} := A_{21} + \frac{1}{2}Y_{21}$ $A_{21} := L_{22}^HA_{21}$ (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 $A_{10} = L_{11}^HA_{10}$ $A_{10} = A_{10} + L_{21}^HA_{20}$ (GEMM) $Y_{21} = A_{22}L_{21}$ (HEMM) $A_{21} = A_{21}L_{11}$ $A_{21} = A_{21} + \frac{1}{2}Y_{21}$ $A_{11} = L_{11}^HA_{11}L_{11}$ $A_{11} = A_{11} + (A_{21}^HL_{21} + L_{21}^H * A_{21})$ $A_{21} = A_{21} + \frac{1}{2}Y_{21}$
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 $Y_{10} := A_{11}L_{10}$ $A_{10} := W_{10} = A_{10} + \frac{1}{2}Y_{10}$ $A_{00} := A_{00} + (A_{10}^HL_{10} + L_{10}^HA_{10})$ (HER2K) $A_{10} := A_{10} + \frac{1}{2}Y_{10}$ $A_{10} := L_{11}^HA_{10}$ $A_{11} := L_{11}^HA_{11}L_{11}$ $A_{20} := A_{20} + A_{21}L_{10}$ (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) $A_{10} := W_{10} = A_{10} + \frac{1}{2}Y_{10}$ $A_{00} := A_{00} + (A_{10}^HL_{10} + L_{10}^HA_{10})$ (HER2K) $A_{10} := A_{10} + \frac{1}{2}Y_{10}$ $A_{10} := L_{11}^HA_{10}$ $A_{11} := L_{11}^HA_{11}L_{11}$

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

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

Variant 2 might be a good choice when implementing an out-of-core algorithm, since the highlighted computations 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 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

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

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.

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)
$A_{21} := A_{21} + G_{21} R_{11}$
$A_{10} := L_{11}^{-1} A_{10}$
```

$$\left.\begin{array}{l} C_{11} := \text{tril}(A_{11}) \\ \text{triu}(C_{11}) := \text{tril}(C_{11})^H \\ C_{11} := L_{11}^{-1} C_{11} \\ C_{11} := C_{11} L_{11}^{-1} \\ \text{tril}(A_{11}) := \text{tril}(C_{11}) \end{array}\right\} A_{11} := L_{11}^{-1} A_{11} L_{11}^{-H}$$

```
$A_{21} := A_{21} L_{11}^{-H}$
```

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

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 of this operation 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 each of length 8, and the intra-node dimension, T, is of size 4. The optimal decomposition of these four dimensions into a two-dimensional topology was empirically found to be $(Z, T) \times (X, Y)$ in every experiment, which is to say that the Z and T dimensions are combined to form the column dimension of our two-dimensional process grid, while the X and Y dimensions form the row dimension. This decomposition results in a 32×64 process grid.

ScaLAPACK. ScaLAPACK [8, 1, 6, 4] was developed in the 1990s as a distributed 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. 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 Basic Linear Algebra Subprograms (BLAS) [16, 10, 9]. 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 $A := L^{-1} A L^{-H}$ and Variant 5 from Figure 2 when computing $A := L^H A L$. In addition, for $A := L^{-1} A L^{-H}$, 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` appears 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 [23]. It is coded in C++ in a style that resembles the FLAME/C API [3] used to implement the `libflame` library [25, 26] (a modern replacement for LAPACK, coded in C). 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 [17] for details.

Tuning. 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.

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 $A := L^{-1} A L^{-H}$ 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.

In Figure 7 performance for computing $A := L^H A L$ 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.

[1]Elemental also efficiently supports SMP+MPI parallelism while ScaLAPACK does not seem to benefit from this kind of hybrid parallelism on this architecture [17]. For the sake of an apples-to-apples comparison, performance of hybrid implementations is not given for either package.

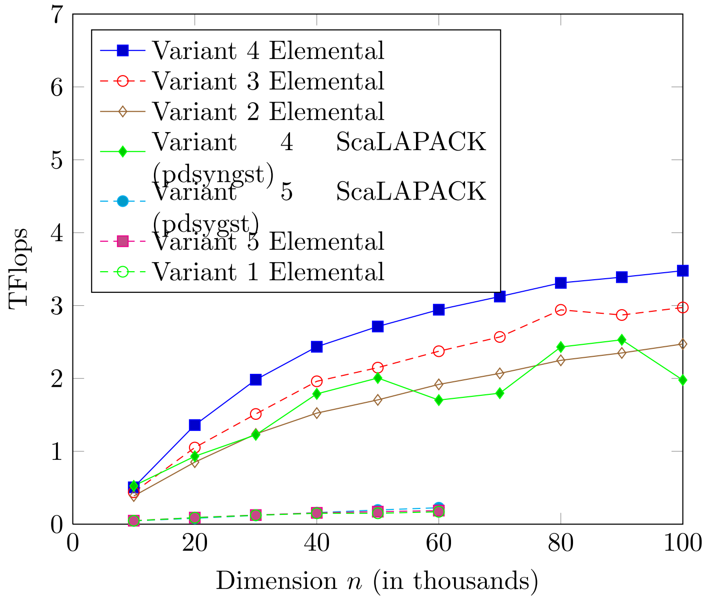

Figure 6: Performance of the various implementations for $A := L^{-1}AL^{-H}$ on 2048 cores of Blue Gene/P. The top of the graph represents the theoretical peak of this architecture. (The three curves for Variants 1 and 5, which cast substantial computation in terms of a parallel TRSM, essentially coincide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

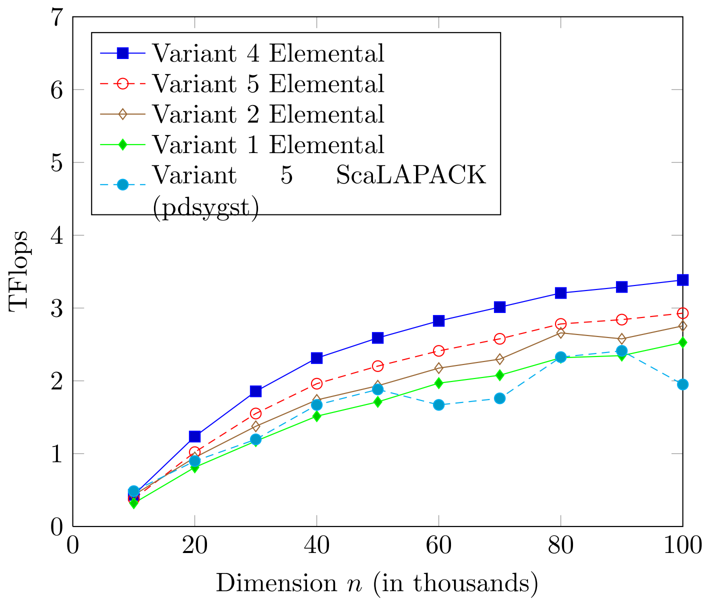

Figure 7: Performance of the various implementations for $A := L^{H}AL$ on 2048 cores of Blue Gene/P. The legend lists the implementations from fastest to slowest for the largest problem size.

5 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 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 $A := L^HAL$ 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.

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 [25] (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 [20, 5, 19]. M-script implementations for Matlab and Octave can be found at

`http://www.cs.utexas.edu/users/flame/Extras/FLAWN56/`.

Acknowledgements

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

References

[1] 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.

[2] Paolo Bientinesi, Brian Gunter, and Robert A. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Soft.*, 35(1):3:1–3:22, July 2008.

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

[4] 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.

[5] Ernie Chan and Francisco D. Igual. 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 2010.

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

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

[8] Jack Dongarra, 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.

[9] 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.

[10] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

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

[12] Kazushige Goto and Robert van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Soft.*, 35(1):1–14, 2008b.

[13] 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.

[14] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Soft.*, 27(4):422–455, 2001.

[15] 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.

[16] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[17] Jack Poulson, Bryan Marker, Jeff R. Hammond, Nichols A. Romero, and Robert van de Geijn. 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`.

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

[19] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *ACM SIGPLAN 2009 symposium on Principles and practices of parallel programming (PPoPP'09)*, pages 121–129, 2009a.

[20] 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 Trans. Math. Soft.*, 36(3):14:1–14:26, July 2009b.

[21] Mark P. Sears, Ken Stanley, and Greg Henry. Application of a high performance parallel eigensolver to electronic structure calculations. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–1, Washington, DC, USA, 1998. IEEE Computer Society.

[22] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

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

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

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

[26] Field G. Van Zee, Ernie Chan, Robert van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Introducing: The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering*, 11(6):56–62, 2009.