Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix

PAOLO BIENTINESI
Duke University
and
BRIAN GUNTER
Delft University of Technology
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

We study the high-performance implementation of the inversion of a Symmetric Positive Definite (SPD) matrix on architectures ranging from sequential processors to Symmetric MultiProcessors to distributed memory parallel computers. This inversion is traditionally accomplished in three "sweeps": a Cholesky factorization of the SPD matrix, the inversion of the resulting triangular matrix, and finally the multiplication of the inverted triangular matrix by its own transpose. We state different algorithms for each of these sweeps as well as algorithms that compute the result in a single sweep. One algorithm outperforms the current ScaLAPACK implementation by 20-30 percent due to improved load-balance on a distributed memory architecture.

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

General Terms: Algorithms;Performance

Additional Key Words and Phrases: linear algebra, libraries, symmetric positive definite, inversion

1. INTRODUCTION

We discuss the need for the inclusion of families of algorithms and implementations in dense linear algebra libraries in order to more effectively address situation specific requirements. Special situations may be due to architectural features of a target platform or to application requirements. While this observation is not new, the general consensus in the community has been that traditional libraries are already too complex to develop when only one or two algorithms are supported, making it impractical to consider including all algorithms for all operations [Demmel and Don-

Authors' addresses: Paolo Bientinesi, Department of Computer Science, Duke University, Durham, NC 27708, pauldj@cs.duke.edu. Brian Gunter, Delft University of Technology, Department of Earth Observation and Space Systems, 2629 HS, Delft, The Netherlands, b.c.gunter@tudelft.nl. Robert van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, rvdg@cs.utexas.edu.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0098-3500/20YY/1200-0001 $5.00

garra 2005]. Obstacles include the effort required to identify candidate algorithms, backward compatibility with traditional development techniques, establishing the formal correctness[1] through extensive testing, numerical stability analyses[2], and the time required for empirical tuning. Recent developments towards the systematic and mechanical development of libraries, as part of the Formal Linear Algebra Methods Environment (FLAME) project, suggest that many of these obstacles can be overcome if new software engineering approaches and tools are embraced. A departure from traditional development methods has the potential for greatly reducing the effort and expense of upgrading libraries as new architectures become available and new situations arise.

The FLAME project encompasses a large number of theoretical and practical tools. At the core is a new notation for expressing dense linear algebra algorithms [Quintana et al. 2001; Bientinesi and van de Geijn 2006]. This notation has a number of attractive features: (1) it avoids the intricate indexing into the arrays that store the matrices that often obscures the algorithm; (2) it raises the level of abstraction at which the algorithm is represented; (3) it allows different algorithms for the same operation and similar algorithms for different operations to be easily compared and contrasted; and (4) it allows the state of the matrix at the beginning and end of each loop (the loop-invariant)[3] to be concisely expressed. The notation supports a step-by-step process for deriving formally correct families of loop-based algorithms requiring as input only a mathematical specification of the operation [Bientinesi et al. 005a]. As part of the project, Application Program Interfaces (APIs) for representing algorithms in code have been defined for a number of programming languages [Bientinesi et al. 005b]. These APIs allow the code to closely resemble the formally correct algorithms so that (1) the implementation requires little effort and (2) the formal correctness of the algorithms implies the formal correctness of the implementations. The methodology is sufficiently systematic that it has been made mechanical using Mathematica [Bientinesi 2006]. The project is also working towards making numerical stability analysis [Bientinesi 2006] and performance analysis similarly systematic and mechanical [Gunnels 2001].

The breadth of the methodology has been shown to include all of the Basic Linear Algebra Subprograms (BLAS) [Dongarra et al. 1988; Dongarra et al. 1990], many of the higher level dense linear algebra operations supported by the Linear Algebra Package (LAPACK) [Anderson et al. 1999] and all the operations in the RECSY library [Jonsson and Kågström 2002a; 2002b]. The primary contribution of this paper is to highlight that multiple algorithms for a single operation must be supported by a complete library. This way the user, or, better yet, an expert system can select the best performing or the most appropriate algorithm for a given situation.

This paper uses the inversion of a Symmetric Positive Definite (SPD) matrix

[1]Formal correctness in computer science refers to correctness in the absence of round-off errors.
[2]In the presence of finite precision arithmetic a numerically stable algorithm will yield an answer that equals the exact answer to a nearby problem.
[3]A formal definition of "loop-invariant" can be found in the book *A logical Approach to Discrete Math* [Gries and Schneider 1992]. We caution that this term is often used by the compiler community with a different (almost opposite) meaning.

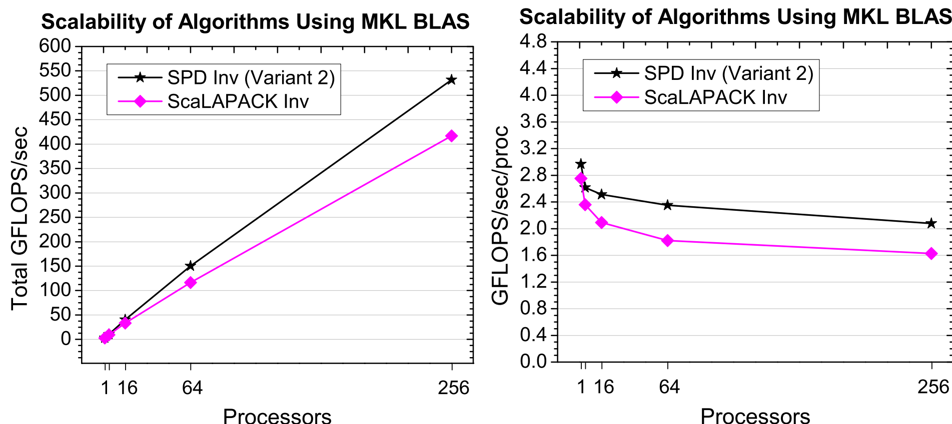

Fig. 1. Comparison of the performance of the ScaLAPACK routine for inverting a SPD matrix and our best one-sweep algorithm implemented with PLAPACK, as a function of the number of processing nodes, p. The problem size is chosen to equal $5000\sqrt{p}$ so that the memory use per processing node is constant. Total performance and performance per node are reported in the left and right graph, respectively. Details on the cluster on which the experiment was performed are given in Section 5.

operation as a case study. Such an operation is used to compute the covariance matrix. While the algorithms presented here can be applied to a number of problems, their development was motivated by specific applications within the Earth, aerospace and medical sciences. For example, the determination of the Earth's gravity field from satellite and terrestrial data is a computationally intensive process that involves the dense linear least squares solution of hundreds of thousands of model parameters from millions of observations [Gunter 2004; Tapley et al. 2004; Sanso and Rummel 1989]. The statistics of these solutions are often desired to aid in determining the accuracy and behavior of the resulting models, so the covariance matrix is typically computed. Another application involves the analysis of nuclear imaging in medicine, where the investigation of noise propagation in Positron Emission Tomography (PET), as well as Single Photon Emission Computed Tomography (SPECT), can involve the inversion of large dense covariance matrices [Gullberg et al. 2003; Huesman et al. 1999].

The inverse of a SPD matrix A is typically obtained by first computing the upper triangular Cholesky factor R of A, $A = R^T R$, after which $A^{-1} = (R^T R)^{-1} = R^{-1}R^{-T}$ can be computed by first inverting the matrix R ($U = R^{-1}$) and then multiplying the result by its transpose ($A^{-1} = UU^T$). We will show that there are multiple loop-based algorithms for each of these three operations, all of which can be orchestrated so that the result overwrites the input without requiring temporary space. Also presented will be two algorithms that overwrite A by its inverse without the explicit computation of these intermediate results, requiring only a single sweep through the matrix, as was already briefly mentioned in [Quintana et al. 2001]. The performance benefit of the single-sweep algorithm for a distributed memory archi-

tecture is illustrated in Fig. 1 in which the best algorithm for inverting a SPD matrix proposed in this paper and implemented with PLAPACK [van de Geijn 1997] is compared with the current three-sweep algorithm supported by ScaLAPACK [Choi et al. 1992]. A secondary contribution of this paper lies with the thorough treatment of loop-based algorithms, implementations, and performance for these operations. Many references to classic inversion methods can be found in [Householder 1964] and [Higham 2002]. An in-place procedure for inverting positive definite matrices by the Gauss-Jordan method is given by Bauer and Reinsch [Bauer and Reinsch 1970]. Recent advances in data formats are applied to matrix inversion in [Andersen et al. 2002] and [Georgieva et al. 2000].

The organization of the paper is as follows: Section 2 introduces multiple algorithms for each of the three sweeps: the computation of the Cholesky factorization, the inversion of the resulting triangular matrix, and the multiplication of a triangular matrix by its transpose. Section 3 discusses one-sweep algorithms for computing the inversion of a SPD matrix. Different scenarios when different algorithms should be used are discussed in Section 4 followed by performance results in Section 5. Conclusions are given in Section 6.

2. ALGORITHMS FOR THE INDIVIDUAL SWEEPS

In this section we present algorithms for the three separate operations that, when executed in order, will compute the inverse of a SPD matrix. Each algorithm is annotated with the names, in parenthesis, of the basic operations being performed, specifying the shape of the operands. A detailed list of basic operations is provided in Fig. 2. In Section 4 we discuss how performance depends not only on the operation performed, but also on the shape of the operands involved in the computation.

2.1 Cholesky Factorization: $A := \textsc{Chol}(A)$

Given a SPD matrix A, its Cholesky factor is defined as the unique upper triangular matrix R such that R has positive diagonal elements and $A = R^T R$ (the Cholesky factorization of A). We will denote the function that computes the Cholesky factor of A by $\textsc{Chol}(A)$. We assume that only the upper triangular part of A is stored and $A := \textsc{Chol}(A)$ overwrites this upper triangular part with the Cholesky factor R. A recursive definition of $A := \textsc{Chol}(A)$ is

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) := \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline \star & R_{BR} \end{array}\right), \text{ where } \begin{cases} R_{TL} = \textsc{Chol}\left(\hat{A}_{TL}\right) \\ R_{TR} = R_{TL}^{-T} \hat{A}_{TR} \\ R_{BR} = \textsc{Chol}\left(\hat{A}_{BR} - R_{TR}^T R_{TR}\right), \end{cases}$$

where the base case for a 1×1 matrix $A = \alpha$ is $\textsc{Chol}(A) = \sqrt{\alpha}$. In this definition, $\hat{A}$ represents the original contents of A,[4] the quadrants A_{TL}, R_{TL}, and $\hat{A}_{TL}$ are all square and of equal dimensions, and $\star$ indicates the symmetric part of the matrix that is not stored. It is this recursive definition, the Partitioned Matrix Expression

[4]Throughout this paper, $\hat{A}$, $\hat{R}$, and $\hat{U}$ will denote the *original* contents of the matrices A, R, and U, respectively.

Name	Operation	Shape	Used in			
			Chol	R^{-1}	UU^T	A^{-1}
DOT	Dot product		1,2		2,3	1
SCAL	Scaling of vector		2,3	1,2,3	1,2	1,2
GEMV	General matrix-vector multiply		2		2	
SYMV	Symmetric matrix-vector multiply					1
GER	General rank-1 update			3		2
SYR	Symmetric rank-1 update		3		1	1,2
TRMV	Triangular matrix-vector multiply			1	3	
TRSV	Triangular solve		1	2		
GEMM						
GEPP	Panel-panel (rank-k) update			3		2
GEMP	Matrix-panel multiply				2	
GEPM	Panel-matrix multiply					
GEBP	Block-panel multiply					
GEPB	Panel-block multiply					
GEPDOT	Panel dot product					1
SYMM						
SYMP	Matrix-panel multiply					1
SYPM	Panel-matrix multiply					
Not shown: SYBP and SYPB, similar to GEBP and GEPB.						
SYRK						
SYPP	Panel-panel update		3		1	1,2
Not shown: SYPDOT, similar to GEPDOT.					2,3	
TRMM						
TRMP	Matrix-panel multiply			1		
TRPM	Panel-matrix multiply				3	
Not shown: TRBP and TRPB, similar to GEBP and GEPB.					1,2	2
TRSM						
TRSMP	Solve with matrix and panel		1			
TRSPM	Solve with panel and matrix			2		
Not shown: TRSBP and TRSPB, similar to GEBP and GEPB.			2,3	1,2,3		1,2

Fig. 2. Basic operations used to implement the different algorithms.

Variant	State maintained (loop-invariant)	where
1	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} R_{TL} & \hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right)$	$R_{TL} = \text{Chol}(\hat{A}_{TL})$ $R_{TR} = R_{TL}^{-T}\hat{A}_{TR}$
2	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} R_{TL} & R_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right)$	
3	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} R_{TL} & R_{TR} \\ \hline \star & \hat{A}_{BR} - R_{TR}^T R_{TR} \end{array}\right)$	

Fig. 3. Loop-invariants (states of matrix A maintained at the beginning and the end of each iteration) corresponding to the algorithms given in Fig. 4 below.

Algorithm: $A := \text{Chol_unb}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**

Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline \star & \alpha_{11} & a_{12}^T \\ \hline \star & \star & A_{22} \end{array}\right)$$
where α_{11} is 1×1

Variant 1:
- $a_{01} := A_{00}^{-T} a_{01}$ (Trsv)
- $\alpha_{11} := \alpha_{11} - a_{01}^T a_{01}$ (Dot)
- $\alpha_{11} := \sqrt{\alpha_{11}}$

Variant 2:
- $\alpha_{11} := \alpha_{11} - a_{01}^T a_{01}$ (Dot)
- $\alpha_{11} := \sqrt{\alpha_{11}}$
- $a_{12}^T := a_{12}^T - a_{01}^T A_{02}$ (Gemv)
- $a_{12}^T := a_{12}^T / \alpha_{11}$ (Scal)

Variant 3:
- $\alpha_{11} := \sqrt{\alpha_{11}}$
- $a_{12}^T := a_{12}^T / \alpha_{11}$ (Scal)
- $A_{22} := A_{22} - a_{12} a_{12}^T$ (Syr)

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

Algorithm: $A := \text{Chol_blk}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$$
where A_{11} **is** $b \times b$

Variant 1:
- $A_{01} := A_{00}^{-T} A_{01}$ (Trsmp)
- $A_{11} := A_{11} - A_{01}^T A_{01}$ (Sypdot)
- $A_{11} := \text{Chol}(A_{11})$

Variant 2:
- $A_{11} := A_{11} - A_{01}^T A_{01}$ (Sypdot)
- $A_{11} := \text{Chol}(A_{11})$
- $A_{12} := A_{12} - A_{01}^T A_{02}$ (Gepm)
- $A_{12} := A_{11}^{-T} A_{12}$ (Trsbp)

Variant 3:
- $A_{11} := \text{Chol}(A_{11})$
- $A_{12} := A_{11}^{-T} A_{12}$ (Trsbp)
- $A_{22} := A_{22} - A_{12}^T A_{12}$ (Sypp)

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

Fig. 4. Unblocked and blocked algorithms for computing the Cholesky factorization. We indicate within parenthesis the name of the operation being performed. A complete list of basic operations, with emphasis on the shape of the operands is given in Fig. 2.

(PME) in FLAME terminology, that is the input to the FLAME methodology for generating loop-based algorithms.

Three pairs of algorithmic variants for computing the Cholesky factorization are presented in Fig. 4. The function $m(X)$ returns the number or rows of matrix X. For details on the notation used, see [Bientinesi et al. 005a; Bientinesi et al. 005b; Bientinesi and van de Geijn 2006]. The algorithms on the left are *unblocked* algorithms, meaning that each iteration moves the computation along by one row and column. Casting the algorithm in terms of matrix-matrix operations (level-3 BLAS [Dongarra et al. 1990]) allows high performance to be attained [Dongarra et al. 1991]. This is achieved by the *blocked* algorithms on the right, which move through the matrix by blocks of b rows and columns.

In Fig. 3 we present the contents (state) of the matrix before and after each iteration of the loop. In computer science, the predicate that defines this state is known as the *loop-invariant*. This state should be a partial result toward computing the PME since until the loop terminates not all of the result is yet computed. Once a loop-invariant is determined, the algorithm is prescribed: the update in the body of the loop must be such that this state is maintained from one iteration to the next. See [Bientinesi et al. 2006] for details on the FLAME methodology as applied to this operation.

The experienced reader will recognize Variant 1 as the "bordered" algorithm, Variant 2 as the "left-looking" algorithm, and Variant 3 as the "right-looking" algorithm.[5] All algorithms are known to be numerically stable.

2.2 Inversion of an Upper Triangular Matrix: $R := R^{-1}$

In this section we discuss the "in-place" inversion of a triangular matrix, overwriting the original matrix with the result. By in-place it is meant that no work space is required. The PME for this operation is

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline \star & R_{BR} \end{array}\right) := \left(\begin{array}{c|c} \hat{R}_{TL}^{-1} & -\hat{R}_{TL}^{-1}\hat{R}_{TR}\hat{R}_{BR}^{-1} \\ \hline \star & \hat{R}_{BR}^{-1} \end{array}\right).$$

Derivations of the algorithms can be found in [Bientinesi et al. 2006].

Three blocked algorithms are given in Fig. 6(left). They, respectively, maintain the loop-invariants in Fig. 5(left). (Note again how the loop-invariants relate to the PME.) For each blocked algorithm there is a corresponding unblocked algorithm, which is not presented. Also, three more pairs of unblocked and blocked algorithms exist that sweep through the matrix from the bottom-right to the top-left. Finally, two more blocked and unblocked pairs of algorithms that are correct in the absense of round-off error but numerical unstable can be derived. We will only consider the three numerically stable algorithms in Fig. 6(left) [Bientinesi et al. 2006; Higham 2002].

[5] This terminology comes from the case where $L = R^T$ is computed instead.

Variant	State maintained	
	$R := R^{-1}$	$U := UU^T$
1	$\left(\begin{array}{c\|c} \hat{R}_{TL}^{-1} & \hat{R}_{TR} \\ \hline 0 & \hat{R}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{U}_{TL}\hat{U}_{TL}^T & \hat{U}_{TR} \\ \hline 0 & \hat{U}_{BR} \end{array}\right)$
2	$\left(\begin{array}{c\|c} \hat{R}_{TL}^{-1} & -\hat{R}_{TL}^{-1}\hat{R}_{TR}\hat{R}_{BR}^{-1} \\ \hline 0 & \hat{R}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{U}_{TL}\hat{U}_{TL}^T + \hat{U}_{TR}\hat{U}_{TR}^T & \hat{U}_{TR} \\ \hline 0 & \hat{U}_{BR} \end{array}\right)$
3	$\left(\begin{array}{c\|c} \hat{R}_{TL}^{-1} & -\hat{R}_{TL}^{-1}\hat{R}_{TR} \\ \hline 0 & \hat{R}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{U}_{TL}\hat{U}_{TL}^T + \hat{U}_{TR}\hat{U}_{TR}^T & \hat{U}_{TR}\hat{U}_{BR}^T \\ \hline 0 & \hat{U}_{BR} \end{array}\right)$

Fig. 5. States maintained in matrix R and U, respectively, by the algorithms given in Fig. 6 below.

Algorithm: $R := R^{-1}$

Partition $R \rightarrow \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array}\right)$
  where R_{TL} is 0×0
while $m(R_{TL}) \neq m(R)$ **do**
  Determine block size b
  Repartition

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} R_{00} & R_{01} & R_{02} \\ \hline 0 & R_{11} & R_{12} \\ \hline 0 & 0 & R_{22} \end{array}\right)$$

  where R_{11} **is** $b \times b$

Variant 1

$R_{01} := -R_{00}R_{01}$ (TRMP)
$R_{01} := R_{01}R_{11}^{-1}$ (TRSPB)
$R_{11} := R_{11}^{-1}$

Variant 2

$R_{12} := -R_{12}R_{22}^{-1}$ (TRSPM)
$R_{12} := R_{11}^{-1}R_{12}$ (TRSBP)
$R_{11} := R_{11}^{-1}$

Variant 3

$R_{12} := -R_{11}^{-1}R_{12}$ (TRSBP)
$R_{02} := R_{02} + R_{01}R_{12}$ (GEPP)
$R_{01} := R_{01}R_{11}^{-1}$ (TRSPB)
$R_{11} := R_{11}^{-1}$

  Continue with

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} R_{00} & R_{01} & R_{02} \\ \hline 0 & R_{11} & R_{12} \\ \hline 0 & 0 & R_{22} \end{array}\right)$$

endwhile

Algorithm: $U := UU^T$

Partition $U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline \star & U_{BR} \end{array}\right)$
  where U_{TL} is 0×0
while $m(U_{TL}) \neq m(U)$ **do**
  Determine block size b
  Repartition

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline \star & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline \star & U_{11} & U_{12} \\ \hline \star & \star & U_{22} \end{array}\right)$$

  where U_{11} **is** $b \times b$

Variant 1:

$U_{00} := U_{00} + U_{01}U_{01}^T$ (SYPP)
$U_{01} := U_{01}U_{11}^T$ (TRPB)
$U_{11} := U_{11}U_{11}^T$

Variant 2:

$U_{01} := U_{01}U_{11}^T$ (TRPB)
$U_{01} := U_{01} + U_{02}U_{12}^T$ (GEMP)
$U_{11} := U_{11}U_{11}^T$
$U_{11} := U_{11} + U_{12}U_{12}^T$ (SYPDOT)

Variant 3:

$U_{11} := U_{11}U_{11}^T$
$U_{11} := U_{11} + U_{12}U_{12}^T$ (SYPDOT)
$U_{12} := U_{12}U_{22}^T$ (TRPM)

  Continue with

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

endwhile

Fig. 6. Blocked algorithms for inverting a triangular matrix and for multiplying a triangular matrix by its transpose.

2.3 Triangular matrix multiplication by its transpose: $C = UU^T$

The PME for this operation is

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline \star & U_{BR} \end{array}\right) := \left(\begin{array}{c|c} \hat{U}_{TL}\hat{U}_{TL}^T + \hat{U}_{TR}\hat{U}_{TR}^T & \hat{U}_{TR}\hat{U}_{BR}^T \\ \hline \star & \hat{U}_{BR}\hat{U}_{BR}^T \end{array}\right).$$

Three loop-invariants are given in Fig. 5(right) that correspond to the algorithms in Fig. 6(right). As for the computation of R^{-1} there are three more algorithms that sweep in the opposite direction. We do not present the corresponding unblocked algorithms.

All algorithms are known to be numerically stable, since they are special cases of matrix-matrix multiplication.

2.4 Three-sweep algorithms

Application of the three operations in the order in which they were presented yields the inversion of a SPD matrix: $UU^T = R^{-1}R^{-T} = (R^TR)^{-1} = A^{-1}$. We refer to any algorithm that executes three algorithms, one for each operation, a three-sweep algorithm. The current implementations of LAPACK and ScaLAPACK use a three-sweep algorithm, consisting of Variant 2 for the Cholesky factorization, Variant 1 for $R := R^{-1}$, and Variant 2 for $U := UU^T$.

3. ONE-SWEEP ALGORITHMS

We now present two algorithms that compute the inverse of a SPD matrix by sweeping through the matrix once rather than three times. We show how one of these algorithms can also be obtained by merging carefully chosen algorithms from Sections 2.1–2.3 into a one-sweep algorithm. The numerical stability of the three-sweep algorithm is known [Higham 2002; Bientinesi et al. 2006] and therefore the merged one-sweep algorithm inherits the same stability properties[6].

The PME for computing $A := A^{-1}$ can be stated as

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) := \left(\begin{array}{c|c} \hat{A}_{TL}^{-1} + \hat{A}_{TL}^{-1}\hat{A}_{TR}B_{BR}\hat{A}_{TR}^T\hat{A}_{TL}^{-1} & -\hat{A}_{TL}^{-1}\hat{A}_{TR}B_{BR} \\ \hline \star & B_{BR} \end{array}\right),$$

where we introduce $B_{BR} = \left(\hat{A}_{BR} - \hat{A}_{TR}^T\hat{A}_{TL}^{-1}\hat{A}_{TR}\right)^{-1}$, the inverse of the Schur complement. From this PME two loop-invariants can be identified, given in Fig. 7, and the application of the FLAME derivation techniques with these loop-invariants yields the algorithms in Fig. 8.

It is possible to identify more loop-invariants other than the two shown in Fig. 7, but the corresponding algorithms perform redundant computations and/or are numerically instable. More loop-invariants yet can be devised by considering the alternative PME

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} B_{TL} & -B_{TL}\hat{A}_{TL}\hat{A}_{BR}^{-1} \\ \hline \star & \hat{A}_{BR}^{-1} + \hat{A}_{BR}^{-1}\hat{A}_{TR}^TB_{TL}\hat{A}_{TR}\hat{A}_{BR}^{-1} \end{array}\right)$$

[6]The order in which the merged one-sweep algorithm updates each entry is the same as the three-sweep algorithm.

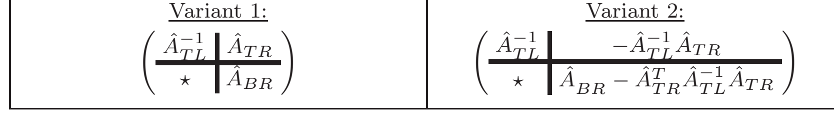

Fig. 7. States maintained in matrix A corresponding to the algorithms given in Fig. 8.

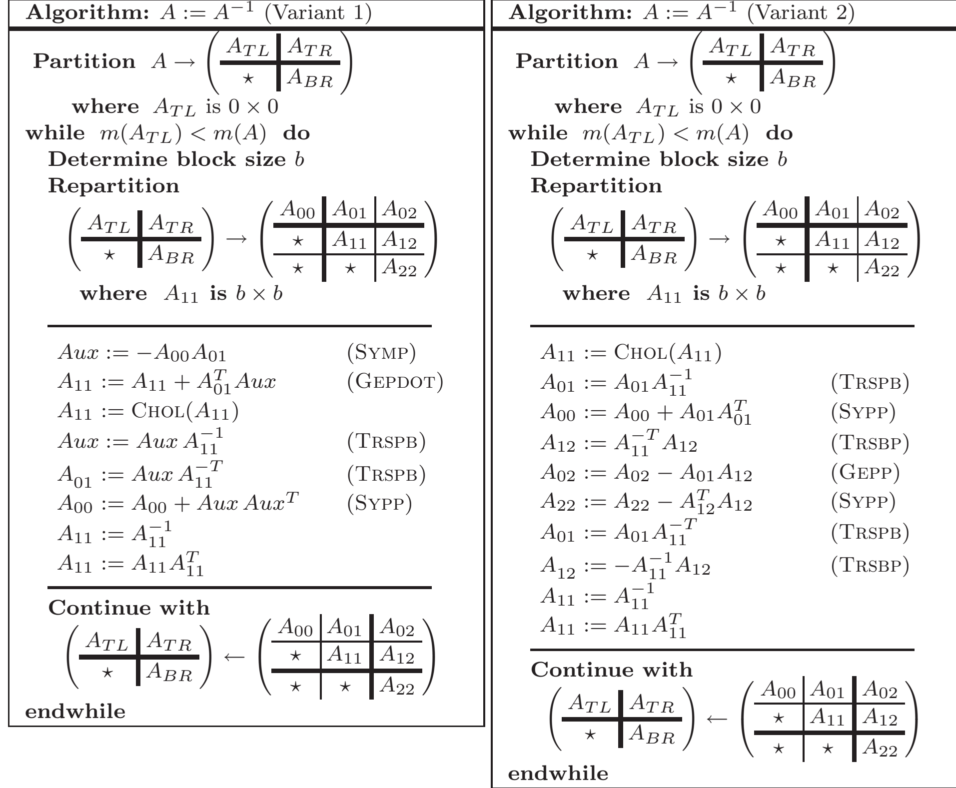

Fig. 8. One-sweep algorithms for inverting a SPD matrix.

where $B_{TL} = \left(\hat{A}_{TL} - \hat{A}_{TR}\hat{A}_{BR}^{-1}\hat{A}_{TR}^T\right)^{-1}$. The corresponding algorithms compute the solution by sweeping the matrix from the bottom right corner as opposed to the two algorithms that we present that sweep the matrix from the top left corner.

A one-sweep algorithm can also be obtained by merging carefully chosen algorithmic variants for each of the three sweeps discussed in Sections 2.1–2.3. The result, in Fig. 9, is identical to Fig. 8 (right), which was obtained by applying the FLAME approach. The conditions under which algorithms can be merged is a topic of current research and goes beyond the scope of this paper.

The real benefit of the one-sweep algorithm in Fig. 9 (left) comes from the following observation: The order of the updates in that variant can be changed as in Fig. 9 (right), so that the most time consuming computations ($A_{22} - A_{12}^T A_{12}$,

Algorithm: $A := A^{-1}$ (Variant 2)

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition

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

where A_{11} **is** $b \times b$

$A_{11} := \text{CHOL}(A_{11})$
$A_{12} := A_{11}^{-T} A_{12}$
$A_{22} := A_{22} - A_{12}^T A_{12}$
} Chol Var. 3

$A_{12} := -A_{11}^{-1} A_{12}$
$A_{02} := A_{02} + A_{01} A_{12}$
$A_{01} := A_{01} A_{11}^{-1}$
$A_{11} := A_{11}^{-1}$
} $R := R^{-1}$ Var. 3

$A_{00} := A_{00} + A_{01} A_{01}^T$
$A_{01} := A_{01} A_{11}^T$
$A_{11} := A_{11} A_{11}^T$
} $U := UU^T$ Var. 1

Continue with

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

endwhile

Algorithm: $A := A^{-1}$ (Variant 2, reordered)

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition

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

where A_{11} **is** $b \times b$

$A_{11} := \text{CHOL}(A_{11})$
$A_{01} := A_{01} A_{11}^{-1}$ (TRSPB)
$A_{12} := A_{11}^{-T} A_{12}$ (TRSBP)

$A_{00} := A_{00} + A_{01} A_{01}^T$ (SYPP)
$A_{02} := A_{02} - A_{01} A_{12}$ (GEPP)
$A_{22} := A_{22} - A_{12}^T A_{12}$ (SYPP)

$A_{01} := A_{01} A_{11}^{-T}$ (TRSPB)
$A_{12} := -A_{11}^{-1} A_{12}$ (TRSBP)
$A_{11} := A_{11}^{-1}$ (Triang. inv.)
$A_{11} := A_{11} A_{11}^T$

Continue with

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

endwhile

Fig. 9. One-sweep algorithm for inverting a SPD matrix as a merging of three sweeps. As a side effect of the reordering of the updates, the sign of the operands in the right column might be the opposite with respect to the corresponding update in the left column.

$A_{00} + A_{01}A_{01}^T$, and $A_{02} + A_{01}A_{12}$) can be scheduled to be computed simultaneously:

$$\left(\begin{array}{c|c|c} A_{00} + A_{01}A_{01}^T & & A_{02} + A_{01}A_{12} \\ \hline \star & & \\ \hline \star & \star & A_{22} - A_{12}^T A_{12} \end{array}\right).$$

On a distributed memory architecture, where the matrix is physically distributed among memories, there is the opportunity to: 1) consolidate the communication among processors by first performing the collective communications for the three updates followed by the actual computations, and 2) improve load-balance since during every iteration of the merged algorithm, *on each element* of the quadrants A_{00}, A_{02} and A_{22} the same amount of computation is performed.

4. DIFFERENT ALGORITHMS FOR DIFFERENT SITUATIONS

There are a number of reasons why different algorithms are more appropriate under different circumstances. In this section we highlight a few.

4.1 Performance

The most prominent reason for picking one algorithm over another is related to performance. Here we mention some high-level issues. Some experimental results related to this are presented in Section 5. Please refer to Fig. 2 for the definition of the BLAS and BLAS-like operations Gemv, Syr, Gepp, Sypp, etc.

Unblocked algorithms are typically used when the problem size is small and the data fits in the L1 or L2 cache of the processor (for example, for the smaller subproblems that occur as part of the blocked algorithms). Here it is the loading and storing of data that critically impacts performance. The symmetric rank-1 update (Syr) requires the matrix to be read and written while the matrix-vector multiply (Gemv) requires the matrix to only be read. This means that algorithms that cast most computation in terms of Gemv incur half the memory operations relative to those that use Syr.

Blocked algorithms cast most computation in terms of one of the matrix-matrix multiplies (Gepp, Gemp, Gepm, Sypp, etc.) [Dongarra et al. 1990; Kågström et al. 1998; Anderson et al. 1999]. There are architectural reasons why the rank-k updates (Gepp and Sypp) on current sequential architectures inherently attain somewhat better performance than the other cases [Goto and van de Geijn 2002; Gunnels et al. 001a]. As a result, it is typically best to pick the algorithmic variant that casts most computation in terms of those cases of matrix-matrix multiplication. (Note that this means a different algorithmic variant is preferred than was for the corresponding unblocked algorithm.)

What property of an algorithmic variant yields high performance on an SMP architecture is a topic of current study. Not enough experience and theory has been developed to give a definitive answer. On distributed memory architectures it appears that casting computation in terms of rank-k updates is again a good choice.

For out-of-core computation (where the problem resides on disk) the issues are again much like they were for the unblocked algorithms: The I/O is much less for algorithms that are rich in the Gemp and/or Gepm cases of matrix-matrix multiplication since the largest matrix involved in these operations is only read.

We have thus reasoned how performance depends not just on what operation is performed, but even on the shape of the operands that are involved in the operation. A taxonomy of operations that exposes the shape of the operands is given in Fig. 2. The algorithms that were presented earlier in this paper were annotated to expose the operations being performed and the legends of the graphs in Section 5 indicate the operation in which most computation is cast, using this taxonomy.

4.2 Algorithmic fault-tolerance

There is a real concern that some future architectures will require algorithmic fault-tolerance to be a part of codes that execute on them. There are many reasons quoted, including the need for low power consumption, feature size, and the need

to use off-the-shelf processors in space where they are subjected to cosmic radiation [Gunnels et al. 001b]. Check-pointing for easy restart partially into the computation is most easily added to an algorithm that maintains a loop-invariant where each quadrant is either completely updated or not updated at all. For such algorithms it is easy to keep track of how far into the matrix the computation has progressed.

4.3 Related operations

An operation closely related to the computation of the Cholesky factorization is determining whether a symmetric matrix is numerically SPD. The cheapest way for this is to execute the Cholesky factorization until a square root of a negative number occurs. Variant 1 will execute the fewest operations if the matrix is not SPD and is therefore a good choice if a matrix is suspected not to be SPD.

4.4 Impact on related computer science research and development

Dense linear algebra libraries are a staple domain for research and development in many areas of computer science. For example, frequently-used linear algebra routines are often employed to assess future architectures (through simulators) and new compiler techniques. As a result, it is important that libraries used for such assessments include all algorithms so that a poor choice of algorithm can be ruled out as a source of an undesirable artifact that is observed in the proposed architecture or compiler.

5. PERFORMANCE EXPERIMENTS

To evaluate the performance of the algorithms derived in the previous sections, both serial and parallel implementations were tested on a variety of problem sizes and on different architectures. Although the best algorithms for each operation attain very good performance, this study is primarily about the qualitative differences between the performance of different algorithms on different architectures.

5.1 Implementations

Implementing all the algorithms discussed in this paper on sequential, SMP, and distributed memory architectures would represent a considerable coding effort if traditional library development techniques were used. The APIs developed as part of the FLAME project have the benefit that the code closely resembles the algorithms as they are presented in this paper. Most importantly, they hide the indexing that makes coding in a traditional style error-prone and time-consuming.

The FLAME/C (C) and PLAPACK (C interfaced with MPI) APIs [Bientinesi et al. 005b; van de Geijn 1997; Chtchelkanova et al. 1997; Gropp et al. 1994; Snir et al. 1996] were used for all the implementations, making the coding effort manageable.

5.2 Platforms

The two architectures chosen for this study were picked to highlight performance variations when using substantially different architectures and/or programming models.

Shared Memory. IBM Power 4 SMP System. This architecture consists of an SMP node containing sixteen 1.3 GHz Power4 processors and 32 GBytes of shared memory. The processors operate at four floating point operations per cycle for a peak theoretical performance of 5.2 GFLOPS/proc (1 GFLOP = 1 billion of floating point operations per second), with a sequential DGEMM (double precision matrix-matrix multiply) benchmarked by the authors at 3.7 GFLOPS/proc.

On this architecture, we compared performance when parallelism was attained in two different ways: 1) Implementing the algorithms with PLAPACK, which employs message passing via calls to IBM's MPI library; and 2) invoking the sequential FLAME/C implementations with calls to the multithreaded BLAS that are part of IBM's ESSL library as well as the GotoBLAS [Goto and van de Geijn].

Distributed Memory. Cray-Dell Linux Cluster. This system consists of an array of Intel PowerEdge 1750 Xeon processors operating at 3.06 GHz. Each compute node contains two processors and has 2 GB of total shared memory (1 Gb/proc). The theoretical peak for each processor is 6.12 GFLOPS (2 floating point operations per clock cycle), with the sequential DGEMM, as part of Intel's MKL 7.2.1 library, benchmarked by the authors at roughly 4.8 GFLOPS.

On this system we measured the performance of PLAPACK-based implementations, linked to the MPICH MPI implementation [Gropp and Lusk 1994] and Intel's MKL library as well as to the GotoBLAS. The system was also used to do the performance comparison with ScaLAPACK reported in Fig. 1 and Section 5.9.

5.3 Data Distribution

ScaLAPACK uses the two-dimensional block cyclic data distribution [Blackford et al. 1997]. PLAPACK uses the Physically Based Matrix Distribution, which is a variation of the block cyclic distribution. The primary difference is that ScaLAPACK ties the algorithmic block size to the distribution block size, whereas PLAPACK does not. Because of this, PLAPACK may use a smaller distribution block size to improve load balance.

5.4 Reading the graphs

The performance attained by the different implementations is given in Figs. 10–14. The top line of most of the graphs represents the asymptotic performance attained on the architecture by matrix-matrix multiplication (DGEMM). Since all the algorithms cast most computation in terms of this operation, its performance is the limiting factor. In the case where different BLAS implementations were employed, the theoretical peak of the machine was used as the top line of the graph. The following operation counts were used for each of the algorithms: $\frac{1}{3}n^3$ for each of $\textsc{Chol}(A)$, R^{-1}, and UU^T, and n^3 for the inversion of a SPD matrix. In the legends, the variant numbers correspond to those used earlier in the paper and the operations within parentheses indicate the matrix-matrix operation in which the bulk of the computation for that variant is cast (See Fig. 2 for details).

5.5 Sequential performance

In Fig. 10 we show performance on a single CPU of the IBM Power 4 system. In these experiments, a block size of 96 was used for all algorithms. From the graphs,

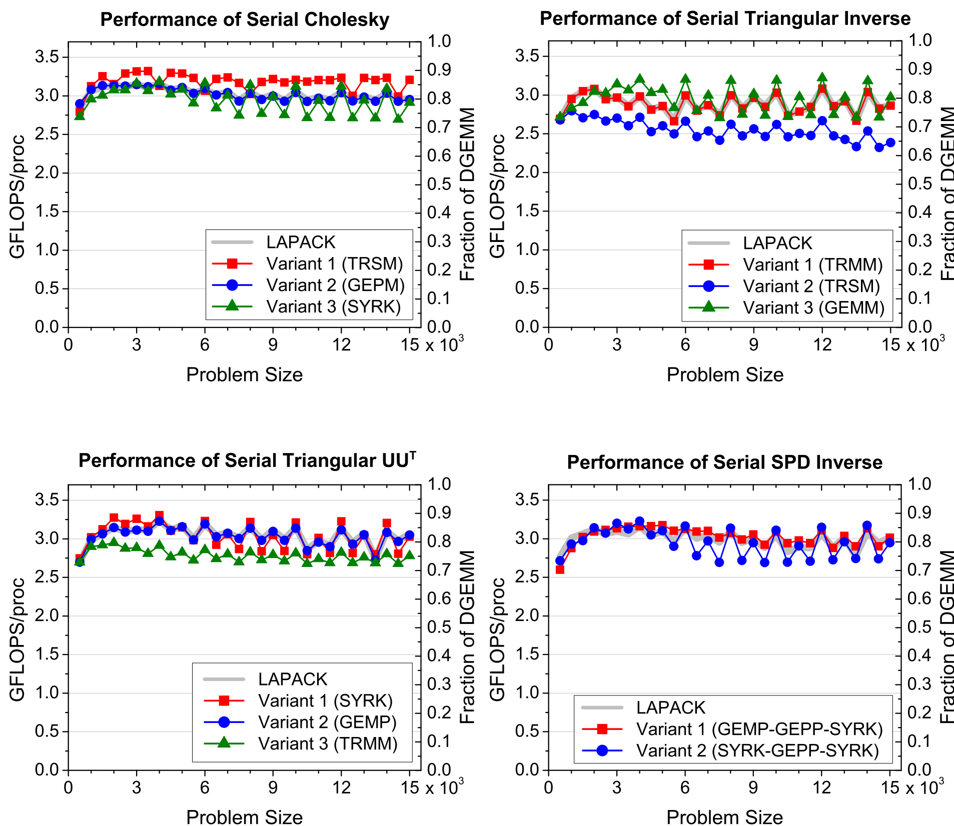

Fig. 10. Sequential performance on the IBM Power4 system.

it is obvious which algorithmic variant was incorporated in LAPACK.

5.6 Parallel performance

In Figs. 11 and 12 we report performance results from experiments on a sixteen CPU IBM Power 4 SMP system and on 16 processors (eight nodes with two processors each) of the Cray-Dell cluster. Since the two systems attain different peak rates of computation, the fraction of DGEMM performance that is attained by the implementations is reported.

On the IBM system, we used an algorithmic block size of 96 for the FLAME/C experiments, while we used a distribution block size of 32 and an algorithmic block size of 96 for the PLAPACK experiments. The results on the IBM system show that linking to multithreaded BLAS yields better performance than the PLAPACK implementations. One reason is that exploiting the SMP features of the system avoids much of the overhead of communication and load-balancing.

For the Cholesky factorization the PLAPACK Variant 1 performs substantially worse than the other variants. This is due to the fact that this variant is rich

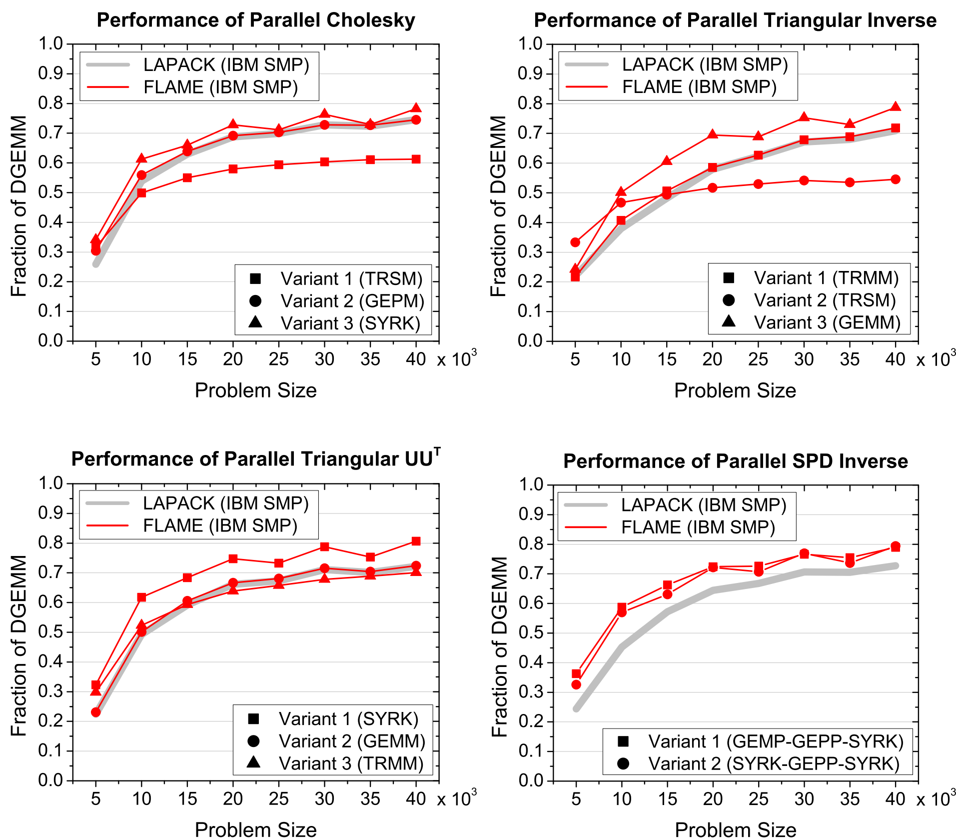

Fig. 11. Parallel Performance on the 16 CPU IBM Power 4 SMP system.

in triangular solves with a limited number of right-hand sides. This operation inherently does not parallelize well on distributed memory architectures due to dependencies. Interestingly, Variant 1 for the Cholesky factorization attains the best performance in the sequential experiment on the same machine.

The PLAPACK implementations of Variants 1 and 2 for computing R^{-1} do not perform well. Variant 1 is rich in triangular matrix times panel-of-columns multiply where the matrix being multiplied has a limited number of columns. It is not inherent that this operation does not parallelize well. Rather, it is the PLAPACK implementation for that BLAS operation that is not completely optimized. Similar comments apply to PLAPACK Variant 3 for computing UU^T and PLAPACK Variant 1 for computing the inversion of a SPD matrix. This shows that a deficiency in the performance of a specific routine in a parallel BLAS library (provided by PLAPACK in this case) can be overcome by selecting an algorithmic variant that casts most computation in terms of a BLAS operation that does attain high per-

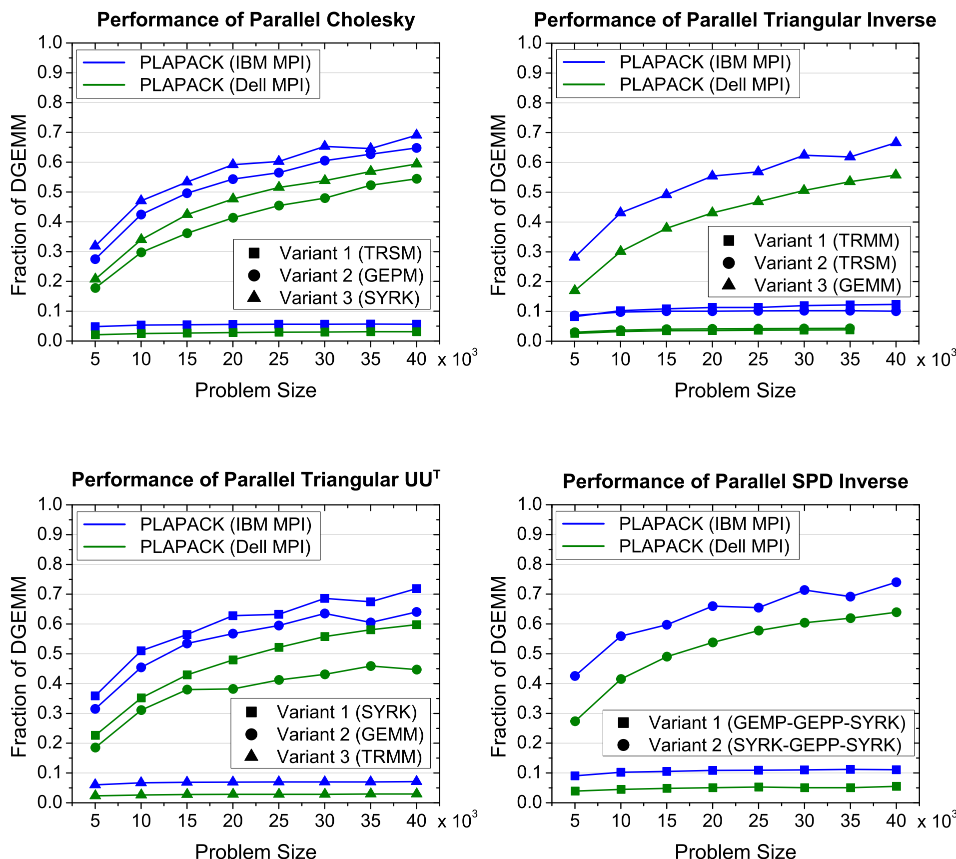

Fig. 12. Parallel Performance for PLAPACK-based implementations (C interfaced with MPI).

formance[7]. Note the cross-over between the curves for the SMP Variants 2 and 3 for the parallel triangular inverse operation. This shows that different algorithmic variants may be appropriate for different problem sizes.

It is again obvious from the graphs which algorithmic variant is used for each of the three sweeps as part of LAPACK. The LAPACK curve does not match either of the FLAME variants in the SPD inversion graph since LAPACK uses a three-sweep algorithm.

5.7 Scalability

In Fig. 13 we report the scalability of the best algorithmic variants for each of the four operations when executed on the Cray-Dell cluster. It is well-known that for these types of distributed memory algorithms it is necessary to scale the problem

[7]The techniques described in this paper still need to be applied to yield parallel BLAS libraries that attain high performance under all circumstances.

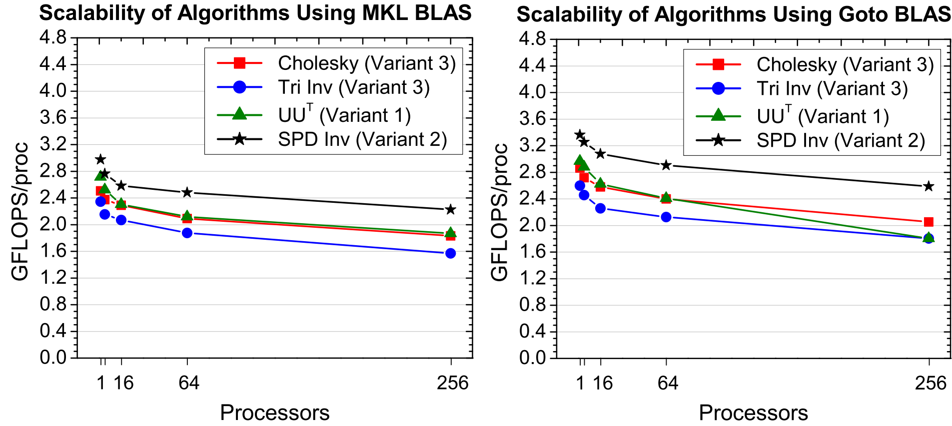

Fig. 13. Scalability on the Dell Cluster. Here the matrix size is scaled to equal $5000 \times \sqrt{p}$ so that memory use per processor is held constant. Left: when linked to MKL 7.X. Right: when linked to GotoBLAS 0.97.

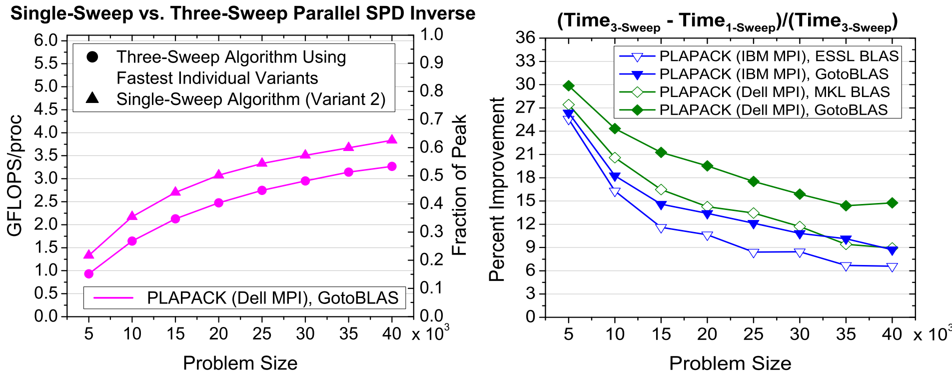

Fig. 14. Comparison of the Three-Sweep and Single-Sweep SPD inverse algorithms. The left panel shows the performance difference for the case run on the Cray-Dell system linked to the GotoBLAS. The right panel shows the wall-clock savings for all PLAPACK cases.

size with the square-root of the number of processors, so that memory-use per processor is kept constant [Hendrickson and Womble 1994; Stewart 1990]. Notice that as the number of processors is increased, the performance per node that is attained eventually decreases very slowly, indicating that the implementations are essentially scalable.

5.8 Comparison of the Three-Sweep and Single-Sweep Algorithms

We examined the benefits of consolidating the collective communications and improving the load balancing in the single-sweep algorithm. In Fig. 14 (left) we show

improvements in raw performance on the Cray-Dell system. The improvement over three-sweep algorithm is quite substantial, in the 15-30% range. Fig. 14(right) shows the time savings gained for the PLAPACK implementations of the SPD inverse algorithms.

On serial and SMP architectures, essentially no performance improvements were observed by using the single-sweep algorithms over the best three-sweep algorithm. This is to be expected, since for these architectures the communications and load balancing are not an issue.

5.9 Comparison with ScaLAPACK

In Fig. 1 we had already shown a performance comparison with ScaLAPACK on the Dell-Cray cluster. It verifies that our implementations rival and even surpass those of a library that is generally considered to be of high quality and scalable. ScaLAPACK requires the nodes to be logically viewed as an $r \times c$ mesh and uses a block-cyclic distribution of matrices to the nodes. For the ScaLAPACK experiments we determined that $r = c$ attained the best performance and a block size of 32 or 64 was used depending on which achieved better performance. (A block size of 128 achieved inferior performance since it affected load balance.) For the PLAPACK experiments $r = c$, a distribution block size of 32 and an algorithmic block size of 96 was used.

6. CONCLUSION

In this paper, we have shown the benefit of including a multitude of different algorithmic variants for dense linear algebra operations in libraries, such as LAPACK/ScaLAPACK and FLAME/PLAPACK, that attempt to span a broad range of architectures. The best algorithm can then be chosen, as a function of the architecture, the problem size, and the optimized libraries to which the implementations are linked. The FLAME approach to deriving algorithms, discussed in a number of other papers, enables a systematic generation of such families of algorithms.

Another contribution of the paper lies with the link it establishes between the three-sweep and one-sweep approach to computing the inverse of a SPD matrix. The observation that the traditional three-sweep algorithm can be fused together so that only a single pass through the matrix is required has a number of advantages. The single-sweep method provides for greater flexibility because the sequence of operations can be arranged differently than they would be if done as three separate sweeps. This allows the operations of the SPD inverse to be organized to optimize load balance and communication. The resulting single-sweep algorithm outperforms the three-sweep method on distributed memory architectures.

The paper raises many new questions. In particular, the availability of many algorithms and implementations means that a decision must be made as to when to use what algorithm. One approach is to use empirical data from performance experiments to tune the decision process. This is an approach that has been applied in the simpler arena of matrix-matrix multiplication (DGEMM) by the PHiPAC and ATLAS projects [Bilmes et al. 1997; Whaley and Dongarra 1998]. An alternative approach would be to carefully design every layer of a library so that its performance can be accurately modeled [Dackland and Kågström 1996]. We intend to pursue a combination of these two approaches.

7. ACKNOWLEDGEMENTS

The authors would like to acknowledge the Texas Advanced Computing Center (TACC) for providing access to the IBM Power4 and Cray-Dell PC Linux cluster machines, along with other computing resources, used in the development of this study. As always, we are grateful for the support provided by the other members of the FLAME team.

This research was partially sponsored by NSF grants CCF-0342369 and ACI-0305163. 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.

More Information

For more information on FLAME and PLAPACK visit

`http://www.cs.utexas.edu/users/flame`
`http://www.cs.utexas.edu/users/plapack`

REFERENCES

Andersen, B. S., Gunnels, J. A., Gustavson, F. G., and Wasniewski, J. 2002. A recursive formulation of the inversion of symmetric positive definite matrices in packed storage data format. In *Applied Parallel Computing New Paradigms for HPC in Industry and Academia, Seventh International Workshop PARA 2002 Proceedings*. Lecture Notes in Computer Science, No. 2367. Springer, 287–296.

Anderson, E., Bai, Z., Bischof, C. H., Blackford, S., Demmel, J. W., Dongarra, J. J., Croz, J. J. D., Greenbaum, A., Hammarling, S. J., McKenney, A., and Sorensen, D. C. 1999. *LAPACK Users' Guide*, Third ed.

Bauer, F. L. and Reinsch, C. 1970. Inversion of positive definite matrices by the Gauss-Jordan methods. In *Handbook for Automatic Computation Vol. 2: Linear Algebra*, J. H. Wilkinson and C. Reinsch, Eds. Springer, New York, NY, USA, 45–49.

Bientinesi, P. 2006. Mechanical derivation and systematic analysis of correct linear algebra algorithms. Ph.D. thesis, Department of Computer Sciences, The University of Texas.

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 (March), 1–26.

Bientinesi, P., Gunter, B., and van de Geijn, R. 2006. Families of algorithms related to the inversion of a symmetric positive definite matrix. FLAME Working Note #19 CS-TR-06-20, Department of Computer Sciences, The University of Texas at Austin.

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 (March).

Bientinesi, P. and van de Geijn, R. 2006. Representing dense linear algebra algorithms: A farewell to indices. FLAME Working Note #17 CS-TR-06-10, Department of Computer Sciences, The University of Texas at Austin.

Bilmes, J., Asanović, K., Whye Chin, C., and Demmel, J. 1997. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*. Vienna, Austria.

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.

Choi, J., Dongarra, J., Pozo, R., and Walker, D. 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. 1997. Parallel Implementation of BLAS: General Techniques for Level 3 BLAS. *Concurrency: Practice and Experience 9,* 9 (Sept.), 837–857.

Dackland, K. and Kågström, B. 1996. A hierarchical approach for performance analysis of scalapack-based routines using the distributed linear algebra machine. In *PARA '96: Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization*. Lecture Notes in Computer Science, No. 1184. Springer-Verlag, 186–195.

Demmel, J. and Dongarra, J. 2005. LAPACK 2005 prospectus: Reliable and scalable software for linear algebra computations on high end computers. LAPACK Working Note 164 UT-CS-05-546, University of Tennessee. February.

Dongarra, 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., Du Croz, J., Hammarling, S., and Hanson, R. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft. 14,* 1 (March), 1–17.

Dongarra, J., Duff, I., Sorensen, D., and van der Vorst, H. A. 1991. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA.

Georgieva, G., Gustavson, F. G., and Yalamov, P. Y. 2000. Inversion of symmetric matrices in a new block packed storage. In *Numerical Analysis and Its Applications: Second International Conference, NAA 2000*. Springer, 333–340.

Goto, K. and van de Geijn, R. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.*. Accepted for publication.

Goto, K. and van de Geijn, R. A. 2002. On reducing TLB misses in matrix multiplication. Tech. Rep. CS-TR-02-55, Department of Computer Sciences, The University of Texas at Austin.

Gries, D. and Schneider, F. B. 1992. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag.

Gropp, W. and Lusk, E. 1994. User's guide for `mpich`, a portable implementation of MPI. Technical Report ANL-06/6, Argonne National Laboratory.

Gropp, W., Lusk, E., and Skjellum, A. 1994. *Using MPI*. The MIT Press.

Gullberg, G., Huesman, R., Roy, D. G., Qi, J., and Reutter, B. 2003. Estimation of the parameter covariance matrix for a one-compartment cardiac perfusion model estimated from a dynamic sequence reconstructed using map iterative reconstruction algorithms. In *Nuclear Science Symposium Conference Record*. Vol. 5. IEEE, 3019–3023.

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. 2001a. A family of high-performance matrix multiplication algorithms. In *Computational Science - ICCS 2001, Part I*, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, Eds. Lecture Notes in Computer Science 2073. Springer-Verlag, 51–60.

Gunnels, J. A., Katz, D. S., Quintana-Ortí, E. S., and van de Geijn, R. A. 2001b. Fault–tolerant high–performance matrix multiplication: Theory and practice. In *Proceedings of the International Conference for Dependable Systems and Networks (DSN-2001)*. 47–56.

Gunter, B. 2004. Computational methods and processing strategies for estimating earth's gravity field. Ph.D. thesis, Department of Aerospace Engineering and Engineering Mechanics, The University of Texas at Austin.

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

Higham, N. J. 2002. *Accuracy and Stability of Numerical Algorithms*, Second ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Householder, A. 1964. *The Theory of Matrices in Numerical Analysis*. Dover, New York.

HUESMAN, R., KADRMAS, D., DIBELLA, E., AND GULLBERG, G. 1999. Analytical propagation of errors in dynamic SPECT: estimators, degrading factors, bias and noise. *Phys. Med. Biol. 44*, 1999–2014.

JONSSON, I. AND KÅGSTRÖM, B. 2002a. Recursive blocked algorithms for solving triangular systems—part I: One-sided and coupled Sylvester-type matrix equations. *ACM Transactions on Mathematical Software 28,* 4, 392–415.

JONSSON, I. AND KÅGSTRÖM, B. 2002b. Recursive blocked algorithms for solving triangular systems—part II: Two-sided and generalized Sylvester and Lyapunov matrix equations. *ACM Transactions on Mathematical Software 28,* 4, 416–435.

KÅGSTRÖM, B., LING, P., AND LOAN, C. V. 1998. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft. 24,* 3, 268–302.

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.

SANSO, R. AND RUMMEL, R., Eds. 1989. *Theory of Satellite Geodesy and Gravity Field Determination.* Lecture Notes in Earth Sciences, vol. 25. Springer-Verlag, Berlin.

SNIR, M., OTTO, S. W., HUSS-LEDERMAN, S., WALKER, D. W., AND DONGARRA, J. 1996. *MPI: The Complete Reference.* The MIT Press.

STEWART, G. 1990. Communication and matrix computations on large message passing systems. *Parallel Computing 16*, 27–40.

TAPLEY, B., SCHUTZ, B., AND BORN, G. 2004. *Statistical Orbit Determination.* Elsevier Academic Press.

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

WHALEY, R. C. AND DONGARRA, J. J. 1998. Automatically tuned linear algebra software. In *Proceedings of SC'98.*

Received Month Year; revised Month Year; accepted Month Year