The Science of Deriving Dense Linear Algebra Algorithms

Dedicated to the Memory of Edsger Wybe Dijkstra

PAOLO BIENTINESI
The University of Texas at Austin
and
JOHN A. GUNNELS
IBM T.J. Watson Research Center
and
MARGARET E. MYERS
The University of Texas at Austin
and
ENRIQUE S. QUINTANA-ORTÍ
Universidad Jaume I
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

In this paper we present a systematic approach to the derivation of families of high-performance algorithms for a large set of frequently encountered dense linear algebra operations. As part of the derivation a constructive proof of the correctness of the algorithm is generated. The paper is structured so that it can be used as a tutorial for novices. However, the method has been shown to yield new, high-performance algorithms for well-studied linear algebra operations and should also be of interest to those who wish to produce best-in-class high-performance codes.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: —*Algorithm Design and Analysis*; *Efficiency*; *User interfaces*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain specific architectures*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms: Algorithms;Design;Theory;Performance

Authors' addresses: Paolo Bientinesi, Margaret E. Myers, Robert A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, {pauldj,myers,rvdg}@cs.utexas.edu. John A. Gunnels, IBM T.J. Watson Research Center, P.O. Box 218 Yorktown Heights, N.Y. 10598, gunnels@us.ibm.com. Enrique S. Quintana-Ortí, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain, quintana@icc.uji.es.

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

1. INTRODUCTION

In this paper we show that for a broad class of linear algebra operations families of algorithms can be systematically derived. We further demonstrate that, given the predicates which describe the input and output, the process of deriving such a family of algorithms is completely prescribed and that the methodology employed is such that the algorithms so produced are guaranteed to be correct.

1.1 Verification and derivation

The title of this paper was taken from the title of Gries' undergraduate text *The Science of Programming* [1981]. That text introduces students to the concept of verifying the correctness of programs. The approach is based on the early work of Floyd [1967], Dijkstra [1968; 1976], and Hoare [1969], among others.

Ideally, algorithmic derivation is constructive: predicates that describe the desired states of the variables at various points in the program are derived first. The statements in the program are then chosen so as to change the state from that described by one predicate to that entailed by the next predicate. Since the statements are chosen to make the predicates *true*, the program is guaranteed to be correct.

A key obstacle to the formal derivation of algorithms for all but the simplest examples is the determination of these predicates. For iterative algorithms (those that depend on a loop) the predicate (called the loop-invariant) that describes the state of the variables just before and after the evaluation of the condition that guards the loop is not easily ascertained *a priori* [Misra 1976; Ernst et al. 2000; Ernst 2000]. Thus, in practice, the program is often written first, at which time predicates that can be used to prove the program correct are determined. This kind *a posteriori* verification of correctness can often be mechanically performed by automatic theorem provers (Formal Methods) [Kaufmann et al. 2000; Kaufman and Moore 1997].

1.2 Derivation of linear algebra algorithms

A fundamental contribution of our work is the observation that, for a broad class of dense linear algebra operations which are typically implemented using a loop, the determination of a loop-invariant *is* systematic. Moreover, *multiple* loop-invariants can be systematically determined, leading to different members of a family of algorithms for the same operation. Thus, our approach is to use formal derivation constructively, which is distinctly different from Formal Methods: the resulting algorithm is guaranteed to be correct and needs not to be verified *a posteriori*.

An additional enabling observation and contribution is that notation is inherently simplified by raising the level at which data (matrices) are described so that indexing details are hidden.

1.3 Relation to other papers on our project

This paper is the third in what we hope will be a series that illustrates the benefits of the formal derivation of algorithms to the high-performance linear algebra library community.

—The first paper [Gunnels et al. 2001] gave a broad outline of the approach, introducing the concept of formal derivation and its application to dense linear algebra algorithms. In that paper we also showed that by introducing an Application Programming Interface (API) for coding the provably correct algorithms, claims about the correctness of the algorithms allow claims about the correctness of the implementation to be made. Finally, we showed that excellent performance can be attained by employing these methods. The primary vehicle for illustrating the techniques in that paper was the LU factorization.
The treatment of the systematic approach was relatively vague in that paper in part because we had not yet had the insight that a "worksheet", introduced later in this paper, provides a convenient framework for the derivation of the algorithms.

—We showed that the method applies to more complex operations in the second paper [Quintana-Ortí and van de Geijn 2003]. In that paper we showed how a large number of new high-performance algorithms for the solution of the triangular Sylvester equation can be derived using the methodology.
As originally submitted, that paper did not include the worksheet. However, this (third) paper was written and submitted before the final submission of the second paper, which was subsequently rewritten by explaining the derivation of those algorithms with the aid of the worksheet.

In a number of workshop papers we have also given a more cursory treatment of the techniques [Gunnels and van de Geijn 2001b; Bientinesi et al. 2002]. This third paper focuses specifically on the system for deriving algorithms: it gives a step-by-step "recipe" that novice and veteran alike can use to rapidly derive correct algorithms. The second paper discussed above should be viewed as an application of the current paper to a much more complex operation.

1.4 Scope

The techniques in this paper apply to linear algebra operations for which there are algorithms that consist of a simple initialization followed by a loop. While this may appear to be extremely restrictive, the linear algebra libraries community has made tremendous strides towards modularity. As a consequence, almost any operation can be decomposed into operations (linear algebra building blocks) that, on the one hand, are themselves meaningful linear algebra operations and, on the other hand, exhibit this simple structure. Over the last few years, we have shown that this category includes all Basic Linear Algebra Subprograms (BLAS) (levels 1, 2, and 3) [Bientinesi et al. 2002; Bientinesi and van de Geijn 2002; Lawson et al. 1979; Dongarra et al. 1988; Dongarra et al. 1990; Gunnels and van de Geijn 2001a], all major factorization algorithms (LU, Cholesky, and QR) [Gunnels et al. 2001], matrix inversion (of general, symmetric, and triangular matrices) [Quintana et al. 2001], and a large number of operations that arise in control theory [Quintana-Ortí and van de Geijn 2003].

1.5 Organization of the paper

While the research described in this paper represents, in our opinion, an original contribution, the format of the paper is that of a tutorial and includes exercises for the reader. The reason for this is two-fold: First, it emphasizes the systematic nature of the approach. Second, it is our experience that it is only when the reader applies the methodology him/herself that the potential of the approach becomes completely clear.

We assume only that the reader has a basic understanding of linear algebra. In particular, it is important for the reader to recall how to multiply partitioned matrices. For those not fluent in the art of high-performance implementation of linear algebra algorithms we suggest first reading [Gunnels et al. 2001]. That paper also contains a more complete discussion regarding how our approach relates to the state-of-the-art in high-performance linear algebra library development.

In Section 2 we review those basic results regarding the verification of the correctness of programs that will be exploited in later sections to derive algorithms. In Section 3 we relate these results to operations and algorithms encountered in linear algebra, using notation more familiar to researchers in that area. This notation is further refined by raising the level of abstraction used to describe data (matrices), hiding indexing details. We also show how, by annotating the algorithm with predicates that describe the state of the variables, the correctness of linear algebra algorithms can be verified *a posteriori*.

The real contribution of this paper is found in Section 4. There we describe our systematic approach by showing how predicates that describe the state of variables at various points in the algorithm can be systematically derived. These predicates then dictate the various components of the algorithm so that the derivation is constructive and is guaranteed to yield a correct algorithm. While the methodology inherently derives *loops* for carrying out a given operation, we briefly discuss how recursive algorithms fit into this framework in Section 4.10.

Ironically, the new methodology generates as many questions as it solves, as we point out in the conclusion. There, we also suggest future directions that can be pursued in order to fully exploit the methodology.

Although it is the derivation of the algorithms that is the central focus of this paper, we do address the practical issues of stability, implementation, and performance. So as not to distract from the central message, these topics are discussed in the appendix.

2. CORRECTNESS OF ALGORITHMS

In this section, we review the relevant notation and results related to the formal verification of programs. Notice that if one assumes that a program is already given, then predicates can be added to the program, after which the program can be verified *a posteriori*. More ideally, the predicates are derived first, and the results given in this section are used to guide the derivation of various components of the program.

2.1 Notation

As part of our reasoning about the correctness of algorithms we will use predicates to indicate assertions about the state of the variables encountered in an algorithm. For example, after the command

$$\alpha := 1$$

which assigns the value 1 to the scalar variable α, we can assert that the predicate "$\alpha = 1$" is *true*. We can then indicate the state of variable α after the assignment by the predicate $\{\alpha = 1\}$.

Similarly, we can use predicates to assert how a statement changes the state. If Q and R are predicates and S is a sequence of commands then $\{Q\}S\{R\}$ has the following interpretation ([Gries 1981], page 100):

> If execution of S is begun in a state satisfying Q, then it is guaranteed to terminate in a finite amount of time in a state satisfying R.

Here $\{Q\}S\{R\}$ is called the *Hoare triplet* and Q and R are referred to as the *precondition* and *postcondition* for the triplet, respectively.

Example The predicate

$$\begin{array}{l} \{\alpha = \beta\} \\ \alpha := \alpha + 1 \\ \{\alpha = (\beta + 1)\} \end{array}$$

is *true*. Here $\alpha = \beta$ is the precondition while $\alpha = (\beta + 1)$ is the postcondition.

2.2 The correctness of loops

In a standard text by Gries and Schneider, used to teach program verification to undergraduates in computer science, we find the following ([1992], pages 236–237)[1]:

> We prefer to write a while loop using the syntax
> **do** $G \rightarrow S$ **od**
> where Boolean expression G is called the [loop-]*guard* and statement S is called the *repetend*.
> [The l]oop is executed as follows: If G is *false*, then execution of the loop terminates; otherwise S is executed and the process is repeated.
> Each execution of repetend S is called an *iteration*. Thus, if G is initially *false*, then 0 iterations occur.

The text goes on to state:

> We now state and prove the fundamental invariance theorem for loops. This theorem refers to an assertion P that holds before and after each iteration (provided it holds before the first). Such a predicate is called a *loop-invariant*.
> (12.43) **Fundamental Invariance Theorem.** Suppose

[1]Small changes from the original text are delimited by [...]. In addition, in that text B is used to denote the (loop-)guard, while we use G. The primary reason for this is that B is commonly used to denote one of the matrix operands.

(1) $\{P \wedge G\} S \{P\}$ holds – i.e. execution of S begun in a state in which P and G are *true* terminates with P *true* – and
(2) $\{P\}$ **do** $G \rightarrow S$ **od** $\{true\}$ – i.e. execution of the loop begun in a state in which P is *true* terminates.

Then $\{P\}$ **do** $G \rightarrow S$ **od** $\{P \wedge \neg G\}$ holds. [In other words, if the loop is entered in a state where P is *true*, it will complete in a state where P is *true* and guard G is *false*.]

The text proceeds to prove this theorem using the axiom of mathematical induction.

3. VERIFICATION OF LINEAR ALGEBRA ALGORITHMS

In this section we introduce the relatively simple operation that computes the solution of a triangular system with multiple right-hand sides. We use this example to relate the notation that is more commonly used to verify programs (given in Section 2) to the notation that is frequently encountered in linear algebra related papers. In particular, we relate the loop as presented in the Fundamental Invariance Theorem to loops as they are more commonly encountered in algorithms in linear algebra. Next, we show that by describing algorithms for linear algebra at a level where indexing details are hidden, a framework for presenting algorithms emerges wherein annotations (predicates) that describe the state of the variables (matrices) can be easily added. This then allows us to illustrate how *a posteriori* verification of the correctness of linear algebra algorithms can be presented in form of a "worksheet".

3.1 A typical linear algebra operation

Given a nonsingular $m \times m$ lower triangular matrix L and an $m \times n$ general matrix B, let X equal the solution of the equation

$$LX = B. \tag{1}$$

Partitioning matrices X and B in (1) by columns yields

$$L \left(\, x_1 \, \middle| \, x_2 \, \middle| \cdots \middle| \, x_n \, \right) = \left(\, b_1 \, \middle| \, b_2 \, \middle| \cdots \middle| \, b_n \, \right)$$

or

$$\left(\, Lx_1 \, \middle| \, Lx_2 \, \middle| \cdots \middle| \, Lx_n \, \right) = \left(\, b_1 \, \middle| \, b_2 \, \middle| \cdots \middle| \, b_n \, \right).$$

From this we conclude that each column of the solution, x_j, must satisfy $Lx_j = b_j$. In other words, the solution of (1) requires the solution of a triangular system for each column of B. Since the coefficient matrix, L, is the same for all columns, the overall computation is referred to as a triangular solve with multiple right-hand sides (TRSM). A simple algorithm for overwriting B with the solution X,

$$B := X = L^{-1}B, \tag{2}$$

is now given by:
We emphasize that rather than computing L^{-1}, the solution of $Lx_j = b_j$ is computed, overwriting b_j. Computing the solution of a triangular system of equations this way is often referred to as *forward substitution*.

for $j = 1, \ldots, n$
 $b_j := x_j = L^{-1} b_j$
endfor

Fig. 1. Simple algorithm for computing $B := X = L^{-1}B$.

Step	Annotated Algorithm: $B := L^{-1}B$
1a	$\left\{ P_{\rm pre} : (B = \hat{B}) \wedge \ldots \right\}$
4	**Partition** $B \rightarrow \left(B_L \Vert B_R \right)$ and $\hat{B} \rightarrow \left(\hat{B}_L \Vert \hat{B}_R \right)$ **where** $\mathrm{n}(B_L) = \mathrm{n}(\hat{B}_L) = 0$
2	$\left\{ P_{\rm inv} : \left(B_L \Vert B_R \right) = \left(L^{-1}\hat{B}_L \Vert \hat{B}_R \right) \wedge \ldots \right\}$
3	**while** $G : (\mathrm{n}(B_L) \neq \mathrm{n}(B))$ **do**
2,3	$\left\{ \left(P_{\rm inv} : \left(B_L \Vert B_R \right) = \left(L^{-1}\hat{B}_L \Vert \hat{B}_R \right) \wedge \ldots \right) \wedge (G : (\mathrm{n}(B_L) \neq \mathrm{n}(B))) \right\}$
5a	**Repartition** $\left(B_L \Vert B_R \right) \rightarrow \left(B_0 \Vert b_1 \vert B_2 \right)$ and $\left(\hat{B}_L \Vert \hat{B}_R \right) \rightarrow \left(\hat{B}_0 \Vert \hat{b}_1 \vert \hat{B}_2 \right)$ **where** $\mathrm{n}(b_1) = 1$
6	$\left\{ Q_{\rm before} : \left(B_0 \Vert b_1 \vert B_2 \right) = \left(L^{-1}\hat{B}_0 \Vert \hat{b}_1 \vert \hat{B}_2 \right) \wedge \ldots \right\}$
8	$b_1 := L^{-1} b_1$
5b	**Continue with** $\left(B_L \Vert B_R \right) \leftarrow \left(B_0 \vert b_1 \Vert B_2 \right)$ and $\left(\hat{B}_L \Vert \hat{B}_R \right) \leftarrow \left(\hat{B}_0 \vert \hat{b}_1 \Vert \hat{B}_2 \right)$
7	$\left\{ Q_{\rm after} : \left(B_0 \Vert b_1 \vert B_2 \right) = \left(L^{-1}\hat{B}_0 \vert L^{-1}\hat{b}_1 \Vert \hat{B}_2 \right) \wedge \ldots \right\}$
2	$\left\{ P_{\rm inv} : \left(B_L \Vert B_R \right) = \left(L^{-1}\hat{B}_L \Vert \hat{B}_R \right) \wedge \ldots \right\}$
	enddo
2,3	$\left\{ \left(P_{\rm inv} : \left(B_L \Vert B_R \right) = \left(L^{-1}\hat{B}_L \Vert \hat{B}_R \right) \wedge \ldots \right) \wedge \neg (G : (\mathrm{n}(B_L) \neq \mathrm{n}(B))) \right\}$
1b	$\left\{ P_{\rm post} : B = L^{-1}\hat{B} \right\}$

Fig. 2. Annotated algorithm for the computation of $B := X = L^{-1}B$ by columns.

3.2 An algorithm, annotated for verification

In order to relate the above material to the discussion in the previous section regarding the verification of the correctness of a loop, we turn our attention to Fig. 2. Let $\hat{B}$ denote the original contents of B, let $\mathrm{m}(A)$ and $\mathrm{n}(A)$ return the row and column dimensions of matrix A, respectively, and let $\mathrm{LowTr}(A)$ be *true* if and only if A is a lower triangular matrix. The *precondition* (Step 1a in Fig. 2) is given by

$$P_{\rm pre} : (B = \hat{B}) \wedge (\mathrm{m}(L) = \mathrm{n}(L)) \wedge \mathrm{LowTr}(L) \wedge (\mathrm{n}(L) = \mathrm{m}(B)).$$

NOTE 1. *For brevity, we will assume throughout this paper that the dimensions and structure of the matrices are correct and will simply give the precondition as* $P_{\rm pre} : (B = \hat{B}) \wedge \ldots$.

Since upon completion the loop is to have computed (2), the postcondition is given by $P_{\rm post} : B = L^{-1}\hat{B}$ (Step 1b).

If one asks what has been computed at the top of the loop in Fig. 1, one discovers that the first $j-1$ columns have been overwritten by the desired solution. In our approach, we partition B and $\hat{B}$ as

$$B \rightarrow \left(B_L \Vert B_R \right) \quad \text{and} \quad \hat{B} \rightarrow \left(\hat{B}_L \Vert \hat{B}_R \right) \tag{3}$$

Partition $B \rightarrow \left(B_L \,\|\, B_R \right)$ and $\hat{B} \rightarrow \left(\hat{B}_L \,\|\, \hat{B}_R \right)$
  where $\mathrm{n}(B_L) = \mathrm{n}(\hat{B}_L) = 0$
while $G : (\mathrm{n}(B_L) \neq \mathrm{n}(B))$ **do**
  Repartition

$$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right) \text{ and } \left(\hat{B}_L \,\|\, \hat{B}_R \right) \rightarrow \left(\hat{B}_0 \,\|\, \hat{b}_1 \,|\, \hat{B}_2 \right)$$

    where $\mathrm{n}(b_1) = 1$
  $b_1 := L^{-1} b_1$
  Continue with

$$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, b_1 \,\|\, B_2 \right) \text{ and } \left(\hat{B}_L \,\|\, \hat{B}_R \right) \leftarrow \left(\hat{B}_0 \,|\, \hat{b}_1 \,\|\, \hat{B}_2 \right)$$

enddo

Fig. 3. Final algorithm for the computation of $B := X = L^{-1}B$ by columns.

where (relating this to Fig. 1) B_L and $\hat{B}_L$ represent the first $j-1$ columns of B and $\hat{B}$, respectively. (Notice that subscripts L and R stand for Left and Right, respectively.) Thus, at the top of the loop the desired current contents of B are given by $P_{\text{inv}} : \left(B_L \,\|\, B_R \right) = \left(L^{-1}\hat{B}_L \,\|\, \hat{B}_R \right) \wedge \ldots$, the loop-invariant (Step 2). Since the loop in Fig. 1 is executed as long as not all columns have been updated, the loop-guard is given by $\mathrm{n}(B_L) \neq \mathrm{n}(B)$ (Step 3).

Now, the loop-invariant must be true before the loop commences, which is achieved by "boot-strapping" the partitioning in (3) by letting B_L have no columns (Step 4).

Finally, we are ready to discuss the body of the loop in Fig. 2. In Fig. 1, the left-most column of the set of columns yet to be updated is updated, moving it to the set of columns that have been updated. In our notation, we accomplish this by repartitioning as in Step 5a, which means that the current contents of B, in terms of the repartitioned matrices, is given by

$$Q_{\text{before}} : \left(B_0 \,\|\, b_1 \,|\, B_2 \right) = \left(L^{-1}\hat{B}_0 \,\|\, \hat{b}_1 \,|\, \hat{B}_2 \right) \wedge \ldots$$

(Step 6). Next, the exposed column is updated (Step 8), which updates the contents of B to

$$Q_{\text{after}} : \left(B_0 \,\|\, b_1 \,|\, B_2 \right) = \left(L^{-1}\hat{B}_0 \,|\, L^{-1}\hat{b}_1 \,\|\, \hat{B}_2 \right) \wedge \ldots$$

(Step 7). After this, the updated column is moved from B_R to B_L (Step 5b).

The Fundamental Invariance Theorem can now be used to show that all assertions in Fig. 2 are *true*, proving that the algorithm is correct. Finally, we notice that $\hat{B}$ was only introduced for the benefit of the assertions in Fig. 2. Since the update in the body of the loop never referenced $\hat{B}$ or its submatrices, a final algorithm is given in Fig. 3.

Exercise 3.1. *Consider the alternative algorithm for computing the columns of B in reverse order:*

$$\begin{array}{l} \text{for } j = n, \ldots, 1 \\ \quad b_j := x_j = L^{-1} b_j \\ \text{endfor} \end{array}$$

Create an annotated algorithm like that given in Fig. 2 for this algorithm.

Step	Annotated Algorithm: $[D, E, F, \ldots] = \text{op}(A, B, C, D, \ldots)$
1a	$\{P_{\text{pre}}\}$
4	**Partition** **where**
2	$\{P_{\text{inv}}\}$
3	**while** G **do**
2,3	$\{(P_{\text{inv}}) \wedge (G)\}$
5a	**Repartition** **where**
6	$\{Q_{bu}\}$
8	S_U
5b	**Continue with**
7	$\{Q_{au}\}$
2	$\{P_{\text{inv}}\}$
	enddo
2,3	$\{(P_{\text{inv}}) \wedge \neg (G)\}$
1b	$\{P_{\text{post}}\}$

Fig. 4. Worksheet for developing linear algebra algorithms.

4. A SYSTEM FOR DERIVING LINEAR ALGEBRA ALGORITHMS

The preceding two sections introduced the basic concepts behind the formal verification of programs and how it relates to linear algebra algorithms. The immediately previous section also introduced notation that hides intricate indexing, which simplified the annotations required to verify the correctness of the given algorithm for computing the solution of a triangular system with multiple right-hand sides. In this section, we show that the combination of the new notation that hides indexing details and the worksheet that organizes the annotations, given in Fig. 4, provide us with the means for systematically deriving correct algorithms.

We shall see that given the precondition and postcondition, a set of possible loop-invariants are easily derived. Once a loop-invariant is selected from the set of loop-invariants, all steps for filling out the worksheet are completely prescribed. Thus, we unveil a system for constructively deriving families of correct algorithms for a given operation. We illustrate the system by applying it to the triangular solve with multiple right-hand sides.

Notice that the steps indicated in the section headers refer to the steps in the worksheet.

4.1 Step 1: precondition and postcondition

The most general form that a linear algebra operation takes is given by

$$[D, E, \ldots] := \text{op}(A, B, C, D, \ldots), \tag{4}$$

where the variables on the left of the assignment $:=$ are the output variables. Notice that, as for the TRSM operation in the previous section, some of the input variables can appear as output variables.

> Example (trsm) In the previous section we saw that the triangular solve with multiple right-hand sides, trsm, can be expressed as $B := L^{-1}B = \text{TRSM}(L, B)$, where L is a nonsingular $m \times m$ lower triangular matrix and B is an $m \times n$ general matrix. For the matrix multiplication on the right to be well-defined, the column dimension of L must match the row dimension of B. We will want to overwrite B with the result without requiring a work array.

The description of the input and output variables dictates the precondition P_{pre}. For variables that are to be overwritten, it is important to introduce variables that indicate the original contents. If Z is both an input and an output variable, we will typically use $\hat{Z}$ to denote the original contents of Z.

> Example (continued) The variables for trsm can be described by the precondition
>
> $$P_{\text{pre}} : (B = \hat{B}) \wedge (\text{m}(L) = \text{n}(L)) \wedge \text{LowTr}(L) \wedge (\text{n}(L) = \text{m}(B))$$
>
> where, as before, $\hat{B}$ indicates the original contents of B. For brevity, typically we will only explicitly state the most important part of this predicate: $P_{\text{pre}} : (B = \hat{B}) \wedge \ldots$.

The operation to be performed and the substitutions required to indicate the original contents of variables dictate the postcondition P_{post}.

> Example (continued) The operation to be performed, $B := L^{-1}B$, translates to the postcondition $P_{\text{post}} : B = L^{-1}\hat{B}$.

4.2 Step 2: deriving loop-invariants

To determine a set of possible loop-invariants, we pick one of the variables and partition it into two submatrices, either horizontally or vertically, or into quadrants. The general rule is that if a matrix has special structure, e.g., triangular or symmetric, it is typically partitioned into quadrants that are consistent with the structure. If the matrix has no special structure, it can be partitioned vertically or horizontally, or into quadrants.

> Example (continued) Let us pick variable L. Since it is triangular, we partition it as
>
> $$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right).$$
>
> Here L_{TL} is square so that both submatrices on the diagonal are themselves lower triangular. (The subscripts TL, BL, and BR stand for Top-Left, Bottom-Left, and Bottom-Right, respectively.)

Next, we substitute this partitioned variable into the postcondition, which is then used to determine the partitioning of the other variables.

Example (continued) Substituting the partitioning of L into the postcondition yields

$$\text{(some partitioning of } B) = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)^{-1} \text{(some partitioning of } \hat{B})$$

This suggests that B and $\hat{B}$ should be partitioned horizontally into two submatrices, or into quadrants. Let us consider the case where B and $\hat{B}$ are partitioned horizontally into two submatrices. Then

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)^{-1} \left(\begin{array}{c} \hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right)$$

In order to be able to multiply the matrices on the right out and to be able to then set the submatrices on the left equal to the result on the right we find that the following must hold:

$$(\mathrm{n}(L_{TL}) = \mathrm{m}(\hat{B}_T)) \wedge (\mathrm{m}(L_{TL}) = \mathrm{m}(B_T)) \qquad (5)$$

which in turn implies that $\mathrm{m}(B_T) = \mathrm{m}(\hat{B}_T)$ since L_{TL} is a square matrix. This is convenient, since B and $\hat{B}$ will reference the same matrix (B is being overwritten).

We now perform the operation using the partitioned matrices. This gives us the desired final contents of the output parameter(s) in terms of the submatrices.

Example (continued)

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)^{-1} \left(\begin{array}{c} \hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right) = \left(\begin{array}{c||c} L_{TL}^{-1} & 0 \\ \hline\hline -L_{BR}^{-1} L_{BL} L_{TL}^{-1} & L_{BR}^{-1} \end{array}\right) \left(\begin{array}{c} \hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right)$$

and hence

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) = \left(\begin{array}{c} L_{TL}^{-1} \hat{B}_T \\ \hline\hline L_{BR}^{-1}(\hat{B}_B - L_{BL} L_{TL}^{-1} \hat{B}_T) \end{array}\right) \qquad (6)$$

Different possible loop-invariants can now be derived by considering individual operations that contribute to the final result. Each such operation may or may not have been performed at an intermediate stage. Careful attention has to be paid to the inherent order in which the operations should be resolved. Any of the resulting conditions on the current contents of the output variable together with the constraints on the structure and dimensions of the submatrices is now considered a possible loop-invariant.

Example (continued) A careful look at (6) shows that inherently $L_{TL}^{-1}\hat{B}_T$ should be computed first, followed by $\hat{B}_B - L_{BL}(L_{TL}^{-1}\hat{B}_T)$, and, finally, $L_{BR}^{-1}(\hat{B}_B - L_{BL}(L_{TL}^{-1}\hat{B}_T))$. This leads to a subset of possible loop-invariants given in Table I.

For each such possible loop-invariant the subsequent steps performed will either show it to be infeasible or will yield an algorithm for computing the operation.

Table I. Possible loop-invariants for the TRSM example when the process is started by partitioning matrix L into quadrants. The reason listed for rejecting the loop-invariant given in the column labeled "Comment" may not be the only reason for doing so.

#	Loop-invariant	Comment
1	$\left(\frac{B_T}{B_B}\right) = \left(\frac{\hat{B}_T}{\hat{B}_B}\right)$	Infeasible (Reason 1).
2	$\left(\frac{B_T}{B_B}\right) = \left(\frac{L_{TL}^{-1}\hat{B}_T}{\hat{B}_B}\right)$	Loop-invariant 1.
3	$\left(\frac{B_T}{B_B}\right) = \left(\frac{L_{TL}^{-1}\hat{B}_T}{\hat{B}_B - L_{BL}L_{TL}^{-1}\hat{B}_T}\right)$	Loop-invariant 2.
4	$\left(\frac{B_T}{B_B}\right) = \left(\frac{L_{TL}^{-1}\hat{B}_T}{L_{BR}^{-1}(\hat{B}_B - L_{BL}L_{TL}^{-1}\hat{B}_T)}\right)$	Infeasible (Reason 2).

Reasons for declaring a loop-invariant infeasible include

Reason 1: No loop-guard exists such that $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$.

Reason 2: No initialization step S_I exists that involves only the partitioning of the variables such that $\{P_{\text{pre}}\}S_I\{P_{\text{inv}}\}$ is *true*.

EXAMPLE (CONTINUED) Unless noted otherwise, we will subsequently use the loop-invariant

$$\left(\frac{B_T}{B_B}\right) = \left(\frac{L_{TL}^{-1}\hat{B}_T}{\hat{B}_B}\right) \tag{7}$$

as our example, showing it to be feasible by deriving an algorithmic variant corresponding to it. Notice that, strictly speaking, the conditions indicated in (5) should be part of the loop-invariant.

4.3 Step 3: derive the loop-guard

The loop-invariant P_{inv} and postcondition P_{post} dictate the loop-guard G since it must have the property that $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$.

EXAMPLE (CONTINUED) Comparing the loop-invariant in (7) with the postcondition $B = L^{-1}\hat{B}$ we see that *if* $B = B_T$, $\hat{B} = \hat{B}_T$, *and* $L = L_{TL}$ then the loop-invariant implies the postcondition, i.e., that the desired result has been computed. Thus, we must choose a loop-guard G so that its negation, $\neg G$, implies that the dimensions of these matrices match appropriately and therefore that $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$. The loop-guard $G : (\text{m}(L_{TL}) \neq \text{m}(L))$ meets this condition.

NOTE 2. *If no loop-guard can be found so that* $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$, *then the loop-invariant is declared infeasible by Reason 1 in Step 2.*

EXAMPLE (CONTINUED) Possible Loop-invariant 1 in Table I is rejected for Reason 1: Since the contents of B never change, (for general triangular matrix L) there is no loop-guard so that exiting the loop implies that the postcondition holds.

4.4 Step 4: derive the initialization

The loop-invariant $P_{\rm inv}$ and precondition $P_{\rm pre}$ dictate the initialization step, S_I. More precisely, S_I should partition the variables so that $\{P_{\rm pre}\}S_I\{P_{\rm inv}\}$ is *true*.

EXAMPLE (CONTINUED) Consider the initialization statement S_I:

$$\textbf{Partition}\quad B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right),\ \hat{B} \rightarrow \left(\begin{array}{c} \hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right),\ \text{and}\ L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$$
$$\textbf{where}\quad B_T \text{ and } \hat{B}_T \text{ have 0 rows and } L_{TL} \text{ is } 0 \times 0$$

in Step 4 in Fig. 5. Since then B_T and $\hat{B}_T$ have no rows, and $B_B = B$ and $\hat{B}_B = \hat{B}$, it is not hard to see that $\{P_{\rm pre}\}S_I\,\{P_{\rm inv}\}$ is *true*.

NOTE 3. *If no initialization S_I can be found so that $\{P_{\rm pre}\}S_I\{P_{\rm inv}\}$ is* true *then the loop-invariant is declared infeasible by Reason 2 in Step 2.*

EXAMPLE (CONTINUED) Possible Loop-invariant 4 in Table I is rejected for Reason 2: no matter how the matrices are initially partitioned, B must be put in a state where it contains the final result (satisfies the postcondition) in order to satisfy the loop-invariant.

4.5 Step 5: determine how to make progress

The loop-guard G and the initialization S_I dictate in what direction the variables need to be repartitioned to make progress towards making G *false*.

EXAMPLE (CONTINUED) Loop-guard G indicates that eventually L_{TL} should equal all of L, at which point G becomes *false* and the loop is exited. After the initialization, L_{TL} is 0×0. The partitioning of L is also such that L_{TL} should always be square. Thus, the repartitioning should be such that as the computation proceeds the dimensions of L_{BR} should decrease as the dimensions of L_{TL} increase. This is accomplished by the shifting of the double-lines as indicated in Steps 5a and 5b in Fig. 5.
Notice that we are exposing **blocks** of rows and/or columns as part of the movement of the double lines. The reason for this is related to performance and will become more clearly apparent in Appendix A.2.

4.6 Step 6: derive the state in terms of the repartitioned variables

The repartitioning of the variables and the loop-invariant $P_{\rm inv}$ in Step 5a dictates $Q_{\rm before}$, the state of the variables before the update S_U. In particular, the double lines in the repartitioning have semantic meaning in that they show what submatrices of the repartitioned matrix correspond to the original submatrices. Substituting the submatrices of the repartitioned matrix into the appropriate place in the loop-invariant yields $Q_{\rm before}$. This is (often referred to as) *textual substitution* into the expression that defines the loop-invariant.

Example (continued) The repartitionings in Step 5a in Fig. 5 identify that

$$\begin{array}{c||c} L_{TL}=L_{00} & \\ \hline\hline L_{BL}=\left(\begin{array}{c}L_{10}\\ \hline L_{20}\end{array}\right) & L_{BR}=\left(\begin{array}{c|c}L_{11}&0\\ \hline L_{21}&L_{22}\end{array}\right)\end{array}\;, \quad \begin{array}{c} B_T=B_0 \\ \hline\hline B_B=\left(\begin{array}{c}B_1\\ \hline B_2\end{array}\right)\end{array}, \quad \text{and} \quad \begin{array}{c} \hat{B}_T=\hat{B}_0 \\ \hline\hline \hat{B}_B=\left(\begin{array}{c}\hat{B}_1\\ \hline \hat{B}_2\end{array}\right)\end{array}.$$

Textual substitution into the loop-invariant yields the state

$$Q_{\text{before}}: \left(\begin{array}{c} B_0 \\ \hline\hline \left(\begin{array}{c}B_1\\ \hline B_2\end{array}\right)\end{array}\right) = \left(\begin{array}{c} L_{00}^{-1}\hat{B}_0 \\ \hline\hline \left(\begin{array}{c}\hat{B}_1\\ \hline \hat{B}_2\end{array}\right)\end{array}\right) \wedge \ldots \tag{8}$$

4.7 Step 7: derive the state in terms of the repartitioned variables at the bottom of the loop body

The redefinition via partitioning of the variables in Step 5b and the loop-invariant P_{inv} dictate the desired state of the variables after the update S_U and after the shifting of the double-lines, Q_{after}. This can again be viewed as textual substitution of the various submatrices into the loop-invariant.

Example (continued) The redefinition in Step 5b in Fig. 5 identifies the following equivalent submatrices:

$$\begin{array}{c||c} L_{TL}=\left(\begin{array}{c|c}L_{00}&0\\ \hline L_{10}&L_{11}\end{array}\right) & \\ \hline\hline L_{BL}=\left(\begin{array}{c|c}L_{20}&L_{21}\end{array}\right) & L_{BR}=L_{22}\end{array}\;, \quad \begin{array}{c} B_T=\left(\begin{array}{c}B_0\\ \hline B_1\end{array}\right) \\ \hline\hline B_B=B_2\end{array}\;, \quad \text{and} \quad \begin{array}{c} \hat{B}_T=\left(\begin{array}{c}\hat{B}_0\\ \hline \hat{B}_1\end{array}\right) \\ \hline\hline \hat{B}_B=\hat{B}_2\end{array}.$$

Textual substitution into the loop-invariant implies that the following state must be true before the redefinition in Step 5b. In other words, the update in Step 8 must leave the variables in the state

$$Q_{\text{after}}: \left(\begin{array}{c} \left(\begin{array}{c}B_0\\ \hline B_1\end{array}\right) \\ \hline\hline B_2\end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c|c}L_{00}&0\\ \hline L_{10}&L_{11}\end{array}\right)^{-1}\left(\begin{array}{c}\hat{B}_0\\ \hline \hat{B}_1\end{array}\right) \\ \hline\hline \hat{B}_2\end{array}\right)$$

which, inverting the triangular matrix and multiplying out the right-hand side, is equivalent to

$$Q_{\text{after}}: \left(\begin{array}{c} \left(\begin{array}{c}B_0\\ \hline B_1\end{array}\right) \\ \hline\hline B_2\end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c}L_{00}^{-1}\hat{B}_0\\ \hline L_{11}^{-1}(\hat{B}_1 - L_{10}L_{00}^{-1}\hat{B}_0)\end{array}\right) \\ \hline\hline \hat{B}_2\end{array}\right) \tag{9}$$

4.8 Step 8: derive how submatrices must be updated

The difference in the states Q_{before} and Q_{after} dictates the update S_U.

Example (continued) Comparing (8) and (9) we find that the updates

$$\begin{aligned} B_1 &:= B_1 - L_{10}B_0 \\ B_1 &:= L_{11}^{-1}B_1 \end{aligned}$$

are required to change the state from Q_{before} to Q_{after}.

Note 4. *If no update can be found that does not use the original contents of a* *matrix to be overwritten, then either the loop-invariant is infeasible (for Reason 1 in Step 2) or a temporary variable is inherently required.*

> EXAMPLE (CONTINUED) In our example, if the update inherently has to use submatrices of $\hat{B}$ (referencing the original contents of B), the loop-invariant would be infeasible since the operation is expected to overwrite the original matrix without requiring a temporary variable.

4.9 Step 9: The final algorithm

Often variables that indicate the original contents of a variable are only introduced to facilitate the predicates denoting the states at different stages of the algorithm. Whenever possible, such variables should be eliminated from the final algorithm.

> EXAMPLE (CONTINUED) By recognizing that $\hat{B}$ is never referenced we can eliminate all parts of the algorithm that refer to this matrix, yielding the final algorithm given in Fig. 6.

EXERCISE 4.1. *(Partition L Variant 2) Repeat Steps 3–8 for the feasible loop-invariant*

$$P_{\rm inv} : \left(\left(\frac{B_T}{\overline{B_B}} \right) = \left(\frac{L_{TL}^{-1}\hat{B}_T}{\overline{\hat{B}_B - L_{BL}L_{TL}^{-1}\hat{B}_T}} \right) \right) \wedge \ldots$$

State the final algorithm by removing references to $\hat{B}$, similar to the algorithm given in Fig. 6

EXERCISE 4.2. *Repeat Step 2 by choosing to partition B vertically:*

$$B \rightarrow \left(B_L \,\|\, B_R \right).$$

Show that this leads to a vertical partitioning of $\hat{B}$: $\hat{B} \rightarrow \left(\hat{B}_L \,\|\, \hat{B}_R \right)$ while L is not partitioned at all. Finally, show that this leads to two possible loop-invariants:

$$\left(B_L \,\|\, B_R \right) = \left(L^{-1}\hat{B}_L \,\|\, \hat{B}_R \right) \wedge \ldots \tag{10}$$

and

$$\left(B_L \,\|\, B_R \right) = \left(\hat{B}_L \,\|\, L^{-1}\hat{B}_R \right) \wedge \ldots \tag{11}$$

EXERCISE 4.3. *(Partition B Variant 1) In Exercise 4.2 consider loop-invariant (10). Show that by applying Steps 3-9 one can systematically derive the algorithms in Figs. 2 and 3.*

If one repartitions

$$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right), \ldots$$

one recovers exactly those algorithms, while the repartitioning

$$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right), \ldots$$

yields the corresponding blocked algorithm.

EXERCISE 4.4. *(Partition B Variant 2) Repeat Exercise 4.3 with loop-invariant (11) and relate the result to Exercise 3.1.*

<table>
<tr><th>Step</th><th>Annotated Algorithm: $B := L^{-1}B$</th></tr>
<tr><td>1a</td><td>$\left\{ B = \hat{B} \wedge \ldots \right\}$</td></tr>
<tr><td>4</td><td>Partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$, $\hat{B} \rightarrow \left(\begin{array}{c} \hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right)$, and $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$
where B_T and $\hat{B}_T$ have 0 rows and L_{TL} is 0×0</td></tr>
<tr><td>2</td><td>$\left\{ \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) = \left(\begin{array}{c} L_{TL}^{-1}\hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right) \right\}$</td></tr>
<tr><td>3</td><td>while $\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)$ do</td></tr>
<tr><td>2,3</td><td>$\left\{ \left(\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) = \left(\begin{array}{c} L_{TL}^{-1}\hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right) \right) \wedge (\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)) \right\}$</td></tr>
<tr><td>5a</td><td>Determine block size b
Repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$,
$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$, $\left(\begin{array}{c} \hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right) \rightarrow \left(\begin{array}{c} \hat{B}_0 \\ \hline\hline \hat{B}_1 \\ \hline \hat{B}_2 \end{array}\right)$
where $m(B_1) = m(\hat{B}_1) = b$ and $m(L_{11}) = b$</td></tr>
<tr><td>6</td><td>$\left\{ \left(\begin{array}{c} B_0 \\ \hline\hline \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} L_{00}^{-1}\hat{B}_0 \\ \hline\hline \left(\begin{array}{c} \hat{B}_1 \\ \hline \hat{B}_2 \end{array}\right) \end{array}\right) \right\}$</td></tr>
<tr><td>8</td><td>$B_1 := B_1 - L_{10}B_0$
$B_1 := L_{11}^{-1}B_1$</td></tr>
<tr><td>5b</td><td>Continue with
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right)$,
$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$, $\left(\begin{array}{c} \hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right) \leftarrow \left(\begin{array}{c} \hat{B}_0 \\ \hline \hat{B}_1 \\ \hline\hline \hat{B}_2 \end{array}\right)$</td></tr>
<tr><td>7</td><td>$\left\{ \left(\begin{array}{c} \left(\begin{array}{c} B_0 \\ \hline B_1 \end{array}\right) \\ \hline\hline B_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} L_{00}^{-1}\hat{B}_0 \\ \hline L_{11}^{-1}(\hat{B}_1 - L_{10}L_{00}^{-1}\hat{B}_0) \end{array}\right) \\ \hline\hline \hat{B}_2 \end{array}\right) \right\}$</td></tr>
<tr><td>2</td><td>$\left\{ \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) = \left(\begin{array}{c} L_{TL}^{-1}\hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right) \right\}$</td></tr>
<tr><td></td><td>enddo</td></tr>
<tr><td>2,3</td><td>$\left\{ \left(\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) = \left(\begin{array}{c} L_{TL}^{-1}\hat{B}_T \\ \hline\hline \hat{B}_B \end{array}\right) \right) \wedge \neg\,(\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)) \right\}$</td></tr>
<tr><td>1b</td><td>$\left\{ P_{\mathrm{post}} : B = L^{-1}\hat{B} \right\}$</td></tr>
</table>

Fig. 5. Annotated algorithm for TRSM example.

Partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$ and $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$
 where B_T has 0 rows and L_{TL} is 0×0
while $\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)$ **do**
 Determine block size b
 Repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

 where $\mathrm{m}(B_1) = b$ and $\mathrm{n}(L_{11}) = b$
 $B_1 := B_1 - L_{10}B_0$
 $B_1 := L_{11}^{-1}B_1$
 Continue with

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

enddo

Fig. 6. Algorithm for the TRSM example.

4.10 Recursion

For the unblocked algorithms, where the boundaries move one row and/or column at a time, the operations that update the contents of some of the matrices tend to be relatively simple. Algorithms for those operations can also be systematically derived, hand-in-hand with the proof of their correctness. Ultimately, these algorithms are built upon addition, subtraction, multiplication, and division as well as operations such as taking the square root of a scalar. Thus, correct algorithms for these operations can be derived using our techniques.

For the blocked algorithms, the operation for which we are deriving the algorithms tends to show up as an operation in the body of the loop (the repetend). Clearly the correctness of the blocked algorithm can be ensured by employing some correct algorithm for this operation in the repetend. In the simplest case, a correct unblocked algorithm can be derived and utilized. However, the implementation of the blocked algorithm itself can be called recursively, or a different blocked algorithmic variant can be used. It is not difficult to see that, as long as only a finite number of levels of such calls are allowed and a correct implementation is called at every level, the correctness of the overall algorithm is ensured.

5. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper we have presented a systematic approach to the derivation of provably correct linear algebra algorithms. The methodology represents what we believe to be a significant refinement of our earlier approach, presented in [Gunnels et al. 2001]. The result is a method which, in our opinion, puts the derivation of families of correct algorithms for a class of dense linear algebra operations on solid theoretical footing. We would like to think that it has scientific, pedagogical, and practical implications.

The fact that we can now systematically derive correct algorithms leads to a number of additional questions:

—Once a correct algorithm has been derived, there is still the problem of translating this algorithm to code without introducing programming bugs. We hint at a solution to this problem in Appendix A.1.

—If it were possible to fully *automate* the derivation and implementation of provably correct algorithms for linear algebra operations, then one could claim that this area of research is well-understood.
  A prototype system, implemented by Sergey Kolos at UT-Austin as part of a semester project, automatically derives all algorithms for some linear algebra operations using Mathematica [Wolfram 1996] as a tool. Similarly, a semi-automatic tool has been developed by one of the authors that uses Mathematica to perform many of the indicated steps for a very broad class of operations. These prototype tools demonstrate that automation may be achievable.

—In practice, implementations of different algorithms will have different performance characteristics as a function of such parameters as operand dimensions and architecture specifications (see also Section A.3). Thus, given that a family of algorithms has been derived, one must choose from among the algorithms. Systematic (or automatic) derivation of parameterized cost analysis hand-in-hand with the algorithms and implementations would be highly desirable. An alternative to this would be the identification of general techniques for a heuristic for selection.
  Some preliminary work on the automatic derivation of cost analyses for parallel architectures shows that this may be possible [Gunnels 2001].

—Not all algorithmic variants will necessarily have the same stability properties. The most attractive solution to this problem would be to make systematic (or automate) the derivation of the stability analysis, hand-in-hand with the derivation of the algorithm. We are hopeful this also will achievable in the future.

—We have mentioned that the presented techniques have been shown to apply to a wide range of linear algebra operations. It would be highly desirable to more precisely characterize the class of problems to which the technique applies.

In conclusion, it is our belief that the application of formal derivation methods to dense linear algebra operations provides a new tool for examining a number of challenging open questions.

Additional Information

Additional information regarding formal derivation of algorithms for linear algebra operations can be found at `http://www.cs.utexas.edu/users/flame/`. As part of that website we have started to maintain a list of operations to which the methodology has been applied. It is our hope to eventually provide for each such operation not only the derivations of the algorithms, but also the implementations using APIs for various languages.

Acknowledgments

This research was partially sponsored by NSF grant ACI-0203685.

We would like to thank Fred G. Gustavson and G.W. (Pete) Stewart for extensive feedback on this research.

REFERENCES

ALPATOV, P., BAKER, G., EDWARDS, C., GUNNELS, J., MORROW, G., OVERFELT, J., VAN DE GEIJN, R., AND WU, Y.-J. J. 1997. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*.

BIENTINESI, P., GUNNELS, J. A., GUSTAVSON, F. G., HENRY, G. M., MYERS, M. E., QUINTANA-ORTI, E. S., AND VAN DE GEIJN, R. A. 2002. The science of programming high-performance linear algebra libraries. In *Proceedings of Performance Optimization for High-Level Languages and Libraries (POHLL-02)*. To appear.

BIENTINESI, P. AND VAN DE GEIJN, R. A. 2002. Developing linear algebra algorithms: Class projects Spring 2002. Tech. Rep. CS-TR-02, Department of Computer Sciences, The University of Texas at Austin. June. In preparation. `http://www.cs.utexas.edu/users/flame/pubs/`.

DIJKSTRA, E. W. 1968. A constructive approach to the problem of program correctness. *BIT 8*, 174–186.

DIJKSTRA, E. W. 1976. *A discipline of programming*. Prentice-Hall.

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

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

ERNST, M. D. 2000. Dynamically discovering likely program invariants. Ph.D. thesis, University of Washington Department of Computer Science and Engineering.

ERNST, M. D., CZEISLER, A., GRISWOLD, W. G., , AND NOTKIN, D. 2000. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*. 449–458.

FLOYD, R. W. 1967. Assigning meanings to programs. In *Symposium on Applied Mathematics*, J. T. Schwartz, Ed. Vol. 19. American Mathematical Society, 19–32.

GRIES, D. 1981. *The Science of Programming*. Springer-Verlag.

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

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

GUNNELS, J. A., GUSTAVSON, F. G., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft. 27*, 4 (December), 422–455.

GUNNELS, J. A., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001. 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. AND VAN DE GEIJN, R. A. 2001a. Developing linear algebra algorithms: A collection of class projects. Tech. Rep. CS-TR-01-19, Department of Computer Sciences, The University of Texas at Austin. May. `http://www.cs.utexas.edu/users/flame/`.

GUNNELS, J. A. AND VAN DE GEIJN, R. A. 2001b. Formal methods for high-performance linear algebra libraries. In *The Architecture of Scientific Software*, R. F. Boisvert and P. T. P. Tang, Eds. Kluwer Academic Press, 193–210.

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

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12*, 12 (October), 576–580.

KAUFMAN, M. AND MOORE, J. S. 1997. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering 23*, 4 (April), 203–213.

KAUFMANN, M., MANOLIOS, P., AND MOORE, J. S., Eds. 2000. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers.

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

Misra, J. 1976. Some aspects of the verification of loop computations. *IEEE Transactions on Software Engineering SE-4,* 6 (Nov.).

Moler, C., Little, J., and Bangert, S. 1987. *Pro-Matlab, User's Guide*. The Mathworks, Inc.

Quintana, E. S., Quintana, G., Sun, X., and van de Geijn, R. 2001. A note on parallel matrix inversion. *SIAM J. Sci. Comput. 22,* 5, 1762–1771.

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

van de Geijn, R. A. Representing linear algebra algorithms in code: The FLAME API. *ACM Trans. Math. Soft.*. submitted.

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*.

Wolfram, S. 1996. *The Mathematica Book: 3rd Edition*. Cambridge University Press.

A. PRACTICAL CONSIDERATIONS

This paper is intended to highlight the formal derivation method that allows algorithms for linear algebra operations to be developed. However, we cannot ignore the fact that in order for these methods to be accepted by the linear algebra libraries community, it must be shown that the insights impact the practical aspects of the development of libraries. In an effort to address these issues without detracting from the central message of the paper, we give a few details in this appendix.

A.1 Implementation

The systematic derivation of provably correct algorithms solves only part of the problem, namely that of establishing that there are no logic errors in the algorithm. So-called programming bugs are generally introduced in the translation of the algorithm into code. While the implementation of the algorithms is not the topic of this paper, we show in Fig. 7 how an appropriately defined API, our FLAME/C library [Gunnels and van de Geijn 2001b; Gunnels et al. 2001; Gunnels and van de Geijn 2001a; van de Geijn], can be used to program algorithms so that the code closely resembles the algorithms. A similar API has been defined for MATLAB (FLAME@lab) [Moler et al. 1987; van de Geijn]. Finally, the Parallel Linear Algebra Package (PLAPACK) [van de Geijn 1997; Alpatov et al. 1997] API has been extended to allow parallel implementations to also more closely reflect the presented algorithms.

Notice that the correctness of the implementations depends on the correctness of the operations used to implement the derived algorithms. The operations that partition matrices, creating references into the original matrices, are extremely simple. Thus their correctness can be established through normal (exhaustive) debugging methods or, preferably, they can themselves be formally proven correct. As mentioned in Section 4.10, algorithms and implementations for operations required in the body of the algorithm can themselves be derived using our techniques.

The code in Fig. 7 illustrates how the FLAME/C API can be used to implement the algorithms for TRSM that start by partitioning L. This example also illustrates how recursion and iteration can be easily mixed in the implementation, as mentioned in Section 4.10.

```
void Trsm_lower_rec( int variant, FLA_Obj L, FLA_Obj B, int nblks, int *nb_alg )
{
  FLA_Obj     LTL, LTR,     L00, L01, L02,    BT,           B0,
              LBL, LBR,     L10, L11, L12,    BB,           B1,
                            L20, L21, L22,                  B2;
  int         b;

  FLA_Part_2x2( L,  &LTL, /**/ &LTR,
                /* ************** */
                  &LBL, /**/ &LBR,   0, 0,     /* submatrix */ FLA_TL );
  FLA_Part_2x1( B,  &BT,
                 /***/
                  &BB,               0, /* length submatrix */ FLA_TOP );

  while ( FLA_Obj_length( LTL ) != FLA_Obj_length( L ) ){
    b = min( FLA_Obj_length( LBR ), nb_alg[ 0 ] );

    FLA_Repart_2x2_to_3x3( LTL, /**/ LTR,         &L00, /**/ &L01, &L02,
                        /* ************* */    /* ******************** */
                                /**/              &L10, /**/ &L11, &L12,
                           LBL, /**/ LBR,         &L20, /**/ &L21, &L22,
                           b, b, /* L11 from */ FLA_BR );
    FLA_Repart_2x1_to_3x1( BT,                   &B0,
                          /**/                   /**/
                                                 &B1,
                           BB,                   &B2,
                           b, /* length B1 from */ FLA_BOTTOM );
    /* ****************************************************************** */
    if ( variant == 1 )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, L10, B0, ONE, B1 );

    if ( nblks > 1 ) Trsm_lower_rec( variant, L11, B1, nblks-1, &nb_alg[ 1 ] );
    else             Trsm_lower_unb( variant, L11, B1 );

    if ( variant == 2 )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, L21, B1, ONE, B2 );
    /* ****************************************************************** */
    FLA_Cont_with_3x3_to_2x2( &LTL, /**/ &LTR,         L00, L01, /**/ L02,
           /**/               L10, L11, /**/ L12,
                            /* ************** */   /* ****************** */
                             &LBL, /**/ &LBR,         L20, L21, /**/ L22,
      /* L11 added to */ FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       B1,
                             /***/                    /**/
                             &BB,                      B2,
      /* B1  added to */ FLA_TOP );
  }
}
```

Fig. 7. FLAME implementation of recursive blocked TRSM algorithm in Fig. 6 (`variant == 1`) and the algorithm in Exercise 4.1 (`variant == 2`).

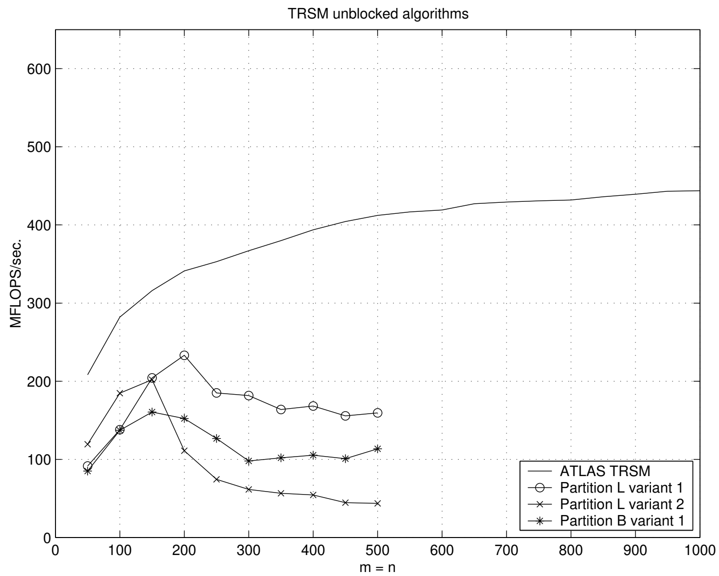

Fig. 8. Performance of unblocked TRSM implementations.

A.2 Experimental Results

In this section we illustrate how the derivation method, combined with the FLAME/C API, leads to high-performance algorithms and implementations for the TRSM operation. Performance was measured on a 650 MHz Intel (R) Pentium (R) III processor-based laptop with a 256K L2 cache running the Linux (Red Hat 7.1) operating system. All computations were performed in 64-bit (double precision) arithmetic. For our implementations, the FLAME/C API linked to BLAS provided by the ATLAS Version R3.2 BLAS library for the Pentium III processor [Whaley and Dongarra 1998]. In other words, whenever a call like `FLA_Ger` is made, it results in a call to the corresponding BLAS routine, in this case the rank-1 update `dger`. The only exception occurs when `FLA_Gemm` is called: For some of the experiments, the ATLAS implementation of the DGEMM routine is called by this routine. For other experiments, our ITXGEMM [Gunnels et al. 2001] implementation of DGEMM is called instead.

In our graphs we report the rate of computation, in millions of floating point operations per second (MFLOPS), using the accepted operation count of n^3 floating point operations, where B is $n \times n$. Notice that the theoretical peak of this particular architecture is 650 MFLOPS. However, due to memory bandwidth limitations, in practice the peak performance achieved by `dgemm` is around 525 MFLOPS. [Gunnels et al. 2001].

In Fig. 8 we report the performance of various unblocked algorithms. These implementations perform the bulk of their computation in the level-2 BLAS operations `dger`, `dgemv`, and/or `dtrsv` [Dongarra et al. 1988]. It is well-known that these operations cannot attain high-performance since they perform $O(n^2)$ operations on

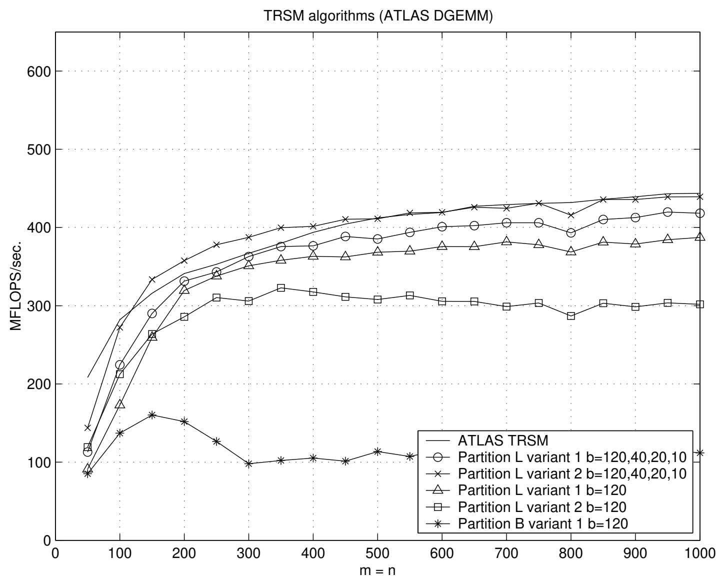

Fig. 9. Performance of blocked and recursive TRSM implementations. Here FLAME/C is interfaced with the DGEMM provided by ATLAS.

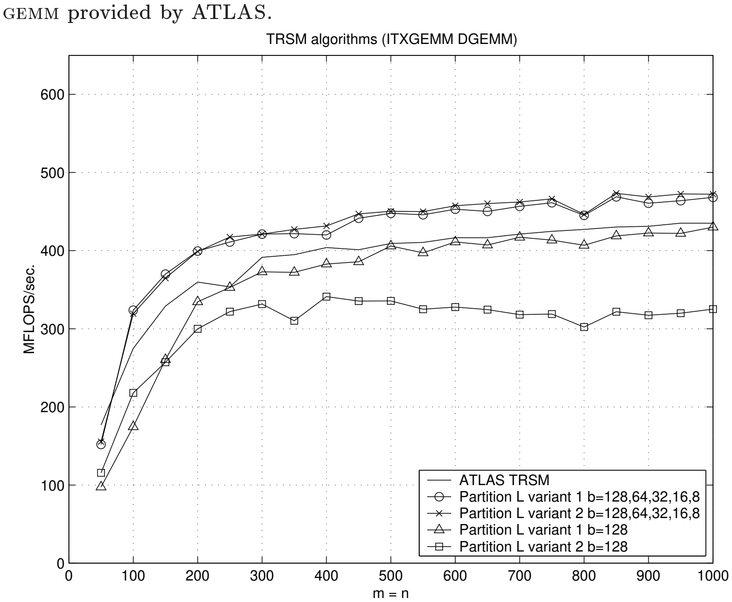

Fig. 10. Performance of blocked and recursive TRSM implementations. Here FLAME/C is interfaced with the DGEMM provided by ITXGEMM.

$O(n^2)$ data, which makes the limited memory bandwidth a bottleneck. Note that `Partition L variant 1` and `Partition L variant 2` perform most of their computation in `dgemv` and `dger`, respectively. This explains the relative performance of these implementations since high-performance implementations of `dgemv` incur about half the memory traffic of `dger`. `Partition B variant 1` performs the bulk of its computation in `dtrsv`. In theory, this implementation should actually be able to attain higher performance than either of the other two implementations for small matrices as matrix L can be kept in the L1 cache. However, its performance suffers considerably from the fact that the FLAME approach to tracking submatrices is particularly expensive for this implementation.

In Fig. 9 we report the performance of blocked versions of the algorithms when the algorithmic blocksize, b, equals 120 and an unblocked implementation of the indicated variant is used for the smaller subproblem. We also show the performance of recursive implementations where the blocks were chosen to equal $b = 120, 40, 20, 10$, after which an unblocked algorithm was used once matrix L was smaller than 10×10. The matrix-matrix multiply called by `FLA_Gemm` in this case is provided by ATLAS. These block sizes were chosen in an attempt to optimize the implementation that uses ATLAS.

In Fig. 10 we report the same experiments as reported in Fig. 9 except that our ITXGEMM matrix multiplication kernel is used rather than the ATLAS counterpart. The block sizes were adjusted to accommodate different design decisions made when implementing this matrix multiplication kernel, as indicated in the legend of the graph.

A.3 How to choose

An interesting question becomes how to choose from among the different algorithms. A lot of factors affect the answer to this question.

Quality of `dgemm`. Consider the matrix-matrix multiplication $C := C - AB$, where C, A, and B are $m \times n$, $m \times k$, and $k \times n$, respectively. The blocked algorithms derived in this paper inherently attempt to cast the bulk of their computation in terms of this matrix-matrix multiplication.

Often, implementations of `dgemm` achieve better performance when k is small relative to m and n than when m is small relative to n and k. When considering the update $B_1 := B_1 - L_{10} B_0$ in Variant 1, the block size, b, is typically chosen so that m is small relative to n and k in the call to the matrix-matrix multiplication kernel. By contrast, those who worked out the details for Exercise 4.1 will see that the bulk of computation is in the update $B_2 := B_2 - L_{21} B_1$ in Variant 2. This explains why `Partition L variant 2` generally outperforms `Partition L variant 1` in Fig. 9. When the implementation of `dgemm` is such that it performs well in both cases, the performance of the two variants is nearly identical, as is shown in Fig. 10.

The conclusion is that the choice of the variant depends on the implementation of the underlying operations.

Size of the matrix and architecture specifications. Consider the situation where we wish to solve $LX = B$ in the case where together L and B do not fit in the main memory of a processor and are thus stored on disk (out-of-core). Assume also that

L fills most of the memory of that processor. Then an out-of-core implementation of TRSM can be achieved by loading most of the memory with L. Next, an algorithm that partitions B by columns, as in Exercise 4.3, can be used to bring blocks of columns of B, (B_1 in the algorithm), into memory, after which some other variant can be used to compute $B_1 := L^{-1}B_1$. Once updated, B_1 is returned to disk.

These same techniques can be applied as subproblems that partially fit in various levels of the memory hierarchy (e.g., L2 and L1 caches) are encountered.

A.4 Stability

As mentioned in the conclusion, the stability of the derived algorithms is a real concern which we hope to address as part of future research.

For the particular operation that is used to illustrate our methodology in this paper, the stability of the different algorithms turns out to be relatively well-understood: All of the derived algorithms, unblocked and blocked, have roughly the same stability properties. For details regarding the stability of the triangular solve with multiple right-hand sides see, for example, [Higham 2002].

Received Month Year; revised Month Year; accepted Month Year