The Science of Deriving Dense Linear Algebra Algorithms [*]

FLAME Working Note #8

Paolo Bientinesi [†] John A. Gunnels [‡] Margaret E. Myers[†]

Enrique S. Quintana-Ortí [§] Robert A. van de Geijn[†]

September 30, 2002

Abstract

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 given. 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 the "high priests of high performance."

1 Introduction

The title of this paper was taken from the title of Gries' undergraduate text *The Science of Programming* [8]. That text introduces students to the concept of verifying the correctness of programs. The approach is based on the early work of Floyd [7], Dijkstra [3, 4], and Hoare [16], among others. Ideally, the proofs are constructive so that the derivation of the program and its proof are fundamentally intertwined. In this paper we apply this methodology to the derivation of algorithms for dense linear algebra operations.

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

- The first paper [11] 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. The primary vehicle for illustrating the techniques in that paper was the LU factorization.

- We showed that the method applies to more complex operations in the second paper [19]. 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.

In a number of workshop papers we have also given a more cursory treatment of the techniques [14, 1].

This third paper focuses primarily on the derivation method. In particular, we show how it provides a step-by-step "recipe" that novice and veteran alike can use to rapidly derive correct algorithms. A nontrivial

[*]This research was partially sponsored by NSF grant #0203685.

[†]Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, {pauldj,myers,rvdg}@cs.utexas.edu.

[‡]IBM T.J. Watson Research Center, P.O. Box 218 Yorktown Heights, N.Y. 10598, gunnels@us.ibm.com.

[§]Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain, quintana@icc.uji.es.

contribution of this paper is also the fact that we can systematically derive the *loop-invariants* that dictate the different algorithmic variants for computing a given operation.

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, whose algorithms have this simple structure. At this time, we do not have a clean characterization of the operations that fall into this category. Over the last few years, we have shown that it includes all Basic Linear Algebra Subprograms (BLAS) (levels 1, 2, and 3) [1, 2, 17, 6, 5, 13], all major factorization algorithms (LU, Cholesky, and QR) [11], matrix inversion (of general, symmetric, and triangular matrices) [18], and a large number of operations that arise in control theory [19]. A subset of these operations is given in Fig. 1.

The format of the paper is that of a tutorial and includes exercises for the reader. 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 [11]. That paper also discusses better how our approach relates to the state-of-the-art in high-performance linear algebra library development.

This paper is organized as follows: In Section 2 we introduce a few of the basics regarding the verification of the correctness of algorithms. In Section 3 we show how to use these techniques to *verify* the correctness of algorithms for linear algebra operations by concentrating on a relatively simple operation that computes the solution of a triangular system of equations with multiple right-hand sides. In Section 4 we go one step further by showing that by constructing an algorithm hand-in-hand with the proof of its correctness, a step-by-step method emerges for deriving families of correct algorithms for a given linear algebra operation. While the methodology inherently derives *loops* for computing a given operation, we briefly discuss how recursive algorithms fit into the picture in Section 5. Concluding remarks which largely concentrate on the future directions of this research can be found in the final section.

While 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 Appendix A.

2 Correctness of Algorithms

In this section we review the relevant formal derivation techniques.

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 ([8], 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.

Level-3 BLAS

Symmetric Matrix-Matrix Multiplication (SYMM)	
$C := \alpha(L + \hat{L}^T)B + \beta C$	$C := \alpha(U + \hat{U}^T)B + \beta C$
$C := \alpha B(L + \hat{L}^T) + \beta C$	$C := \alpha B(U + \hat{U}^T) + \beta C$

Symmetric Rank-K Update (SYRK)	
$\text{lo}(C) := \alpha \text{lo}(AA^T) + \beta \text{lo}(C)$	$\text{up}(C) := \alpha \text{up}(AA^T) + \beta \text{up}(C)$
$\text{lo}(C) := \alpha \text{lo}(A^T A) + \beta \text{lo}(C)$	$\text{up}(C) := \alpha \text{up}(A^T A) + \beta \text{up}(C)$

Symmetric Rank-2K Update (SYR2K)	
$\text{lo}(C) := \alpha \text{lo}(AB^T + BA^T) + \beta \text{lo}(C)$	$\text{up}(C) := \alpha \text{up}(AB^T + BA^T) + \beta \text{up}(C)$
$\text{lo}(C) := \alpha \text{lo}(A^T B + B^T A) + \beta \text{lo}(C)$	$\text{up}(C) := \alpha \text{up}(A^T B + B^T A) + \beta \text{up}(C)$

Triangular Matrix-Matrix Multiplication (TRMM)			
$B := \alpha LB$	$B := \alpha L^T B$	$B := \alpha UB$	$B := \alpha U^T B$
$B := \alpha BL$	$B := \alpha BL^T$	$B := \alpha BU$	$B := \alpha BU^T$

Triangular Solve with Multiple Right-Hand Sides (TRSM)			
$B := \alpha L^{-1} B$	$B := \alpha L^{-T} B$	$B := \alpha U^{-1} B$	$B := \alpha U^{-T} B$
$B := \alpha BL^{-1}$	$B := \alpha BL^{-T}$	$B := \alpha BU^{-1}$	$B := \alpha BU^{-T}$

Level-3 BLAS-Like Operations

TRMM-Like Operations			
$L_1 := \alpha L_1 L_2$	$L_2 := \alpha L_1 L_2$	$L := \alpha L U^T$	$L := \alpha U^T L$
$U_1 := \alpha U_1 U_2$	$U_2 := \alpha U_1 U_2$	$U := \alpha L^T U$	$U := \alpha U L^T$

TRSM-Like Operations			
$L_1 := \alpha L_1 L_2^{-1}$	$L_2 := \alpha L_1^{-1} L_2$	$L := \alpha L U^{-T}$	$L := \alpha U^{-1} L$
$U_1 := \alpha U_1 U_2^{-1}$	$U_2 := \alpha U_1^{-1} U_2$	$U := \alpha L^{-T} U$	$U := \alpha U L^{-T}$

Miscellaneous	
$\text{lo}(C) := \text{lo}(LL^T) + \text{lo}(C)$	$\text{up}(C) := \text{up}(LL^T) + \text{up}(C)$
$\text{lo}(C) := \text{lo}(L^T L) + \text{lo}(C)$	$\text{up}(C) := \text{up}(L^T L) + \text{up}(C)$
$\text{lo}(C) := \text{lo}(UU^T) + \text{lo}(C)$	$\text{up}(C) := \text{up}(UU^T) + \text{up}(C)$
$\text{lo}(C) := \text{lo}(U^T U) + \text{lo}(C)$	$\text{up}(C) := \text{up}(U^T U) + \text{up}(C)$
$\text{lo}(L) := \text{lo}(L^{-1} L^{-T})$	$\text{lo}(L) := \text{lo}(L^{-T} L^{-1})$
$\text{up}(U) := \text{up}(R^{-1} R^{-T})$	$\text{up}(U) := \text{up}(R^{-T} R^{-1})$
$C := \alpha UL + \beta C$	$C := \alpha LU + \beta C$

Factorization Operations

$A := L\backslash U = LU(A)$	$A := U\backslash L = UL(A)$
$A := L = \text{Chol}(A)$	$A := U = \text{Chol}(A)$
$A := Q\backslash R = QR(A)$	$A := Q\backslash L = QL(A)$
$A := R\backslash Q = RQ(A)$	$A := L\backslash Q = QL(A)$

Inversion Operations

$A := A^{-1}$	$\text{lo}(A) := \text{lo}(A^{-1})$ (symmetric A)
$L := L^{-1}$	$U := U^{-1}$

Operations from Control Theory

Solution of the Sylvester Equation	
$C := X$ where $L_1 X + X L_2 = C$	$C := X$ where $U_1 X + X U_2 = C$
$C := X$ where $LX + XU = C$	$C := X$ where $UX + XL = C$

Solution of the Lyapunov Equation (symmetric C)	
$C := X$ where $LX + XL^T = C$	$C := X$ where $L^T X + XL = C$
$C := X$ where $UX + XU^T = C$	$C := X$ where $U^T X + XU = C$

Figure 1: A sampling of operations to which the formal derivation technique has been applied. Note that for most of these, real as well as complex floating point implementations are required. In this figure, lo(A) and up(A) return (reference) the lower and upper triangular part of that matrix, repectively.

EXAMPLE The predicate

$\{\alpha = \beta\}$
$\alpha := \alpha + 1$
$\{\alpha = (\beta + 1)\}$

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
([9], 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. $\{P \wedge G\}S\{P\}$ 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 use the operation that computes the solution of a triangular system with multiple right-hand sides to relate formal verification methods to algorithms for linear algebra operations.

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

$$L \left(\begin{array}{c|c|c|c} x_1 & x_2 & \cdots & x_n \end{array} \right) = \left(\begin{array}{c|c|c|c} b_1 & b_2 & \cdots & b_n \end{array} \right)$$

[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.

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 in Fig. 2. 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.*

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. 3. 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. 3) is given by

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

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* $P_{\mathrm{pre}} : B = \hat{B} \wedge \ldots$.

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

If one asks what has been computed at the top of the loop in Fig. 2, 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

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

where (relating this to Fig. 2) 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 $P_{\mathrm{inv}} : \left(\; B_L \;\|\; B_R \; \right) = \left(\; L^{-1}\hat{B}_L \;\|\; \hat{B}_R \; \right) \wedge \ldots$, the loop-invariant (Step 2). Since the loop in Fig. 2 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. 3. In Fig. 2, 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

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

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

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

(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. 3 are *true* which shows that the algorithm is correct. Finally, we notice that $\hat{B}$ was only introduced for the benefit of the assertions in Fig. 3. Since the update in the body of the loop never referenced $\hat{B}$ or its submatrices, a final algorithm is given in Fig. 4.

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

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

Create an annotated algorithm like that given in Fig. 3 for this algorithm.

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

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

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

```
Partition $B \rightarrow \left( B_L \,\Vert\, B_R \right)$ and $\hat{B} \rightarrow \left( \hat{B}_L \,\Vert\, \hat{B}_R \right)$
    where $\mathrm{n}(B_L) = \mathrm{n}(\hat{B}_L) = 0$
while $G : (\mathrm{n}(B_L) \neq \mathrm{n}(B))$ do
    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 $\mathrm{n}(b_1) = 1$
    $b_1 := L^{-1}b_1$
    Continue with
        $\left( B_L \,\Vert\, B_R \right) \leftarrow \left( B_0 \,\vert\, b_1 \,\Vert\, B_2 \right)$ and $\left( \hat{B}_L \,\Vert\, \hat{B}_R \right) \leftarrow \left( \hat{B}_0 \,\vert\, \hat{b}_1 \,\Vert\, \hat{B}_2 \right)$
enddo
```

Figure 4: Final algorithm for the computation of $B := X = L^{-1}B$ by columns.

```
Annotated Algorithm: $[D, E, F, \ldots] = \text{op}(A, B, C, D, \ldots)$
$\{P_{\text{pre}}\}$
**Partition**

    **where**
$\{P_{\text{inv}}\}$
**while** $G$ **do**
    $\{(P_{\text{inv}}) \wedge (G)\}$
    **Repartition**



        **where**
    $\{Q_{bu}\}$
    $S_U$
    $\{Q_{au}\}$
    **Continue with**


    $\{P_{\text{inv}}\}$
**enddo**
$\{(P_{\text{inv}}) \wedge \neg (G)\}$
$\{P_{\text{post}}\}$
```

Figure 5: Worksheet for developing linear algebra algorithms.

4 Derivation of Linear Algebra Algorithms

The example in the previous section is such that one might conclude that asserting that the algorithm in Fig. 2 is correct is rather trivial. In this section, we claim that in fact the "worksheet" that we created for the TRSM operation can be applied to *constructively derive* a large number of algorithms for this operation and for a large class of linear algebra operations. Indeed, **given the precondition and postcondition, we will show that all other components of the generic worksheet given in Fig. 5 are systematically prescribed**, leading to a family of algorithms for a given linear algebra operation. We describe the derivation process in this section, illustrating the steps by deriving a somewhat more complex algorithm for computing TRSM.

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 $m \times m$ 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.

Step 1

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 X 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

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

where, as before, $\hat{B}$ indicates the original contents of B. For brevity, we will typically 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 $P_{\text{post}} : B = L^{-1}\hat{B}$.

Step 2

The primary way in which we now deviate from the discussion in Section 3 is that we now systematically *derive* the different parts of the annotated algorithm. In particular, we derive possible loop-invariants rather than starting with an implementation from which the loop-invariant is deduced.

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 <u>T</u>op-<u>L</u>eft, <u>B</u>ottom-<u>L</u>eft, and <u>B</u>ottom-<u>R</u>ight, 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:

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

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

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

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)

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

and hence

$$\left(\frac{B_T}{B_B}\right) = \left(\frac{L_{TL}^{-1}\hat{B}_T}{L_{BR}^{-1}(\hat{B}_B - L_{BL}L_{TL}^{-1}\hat{B}_T)}\right) \tag{6}$$

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. 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. Reasons for declaring a loop-invariant infeasible include

Reason 1: (Data dependency) The loop-invariant assumes that data that is needed in a subsequent computation has been overwritten with a partial or final result.

Reason 2: No loop-guard exists such that $P_{\text{inv}} \wedge \neg G \Rightarrow P_{\text{post}}$.

Reason 3: 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*.

Reason 4: (Operation dependency) The loop-invariant requires redundant computation to be performed. Notice that sometimes is becomes beneficial to perform redundant computation in an effort to achieve higher performance, in which case this reason for rejecting a possible loop-invariant would not apply.

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 1.

Example (continued) The feasibility of different possible loop-invariants is discussed in Table 1. 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.

Step 3

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 $G : (\mathrm{m}(L_{TL}) \neq \mathrm{m}(L))$ 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 2 in Step 2.*

Step 4

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

Example (continued) Consider the initialization statement S_I:

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

in Step 4 in Fig. 6. 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 $\{P_{\text{pre}}\}S_I\,\{P_{\text{inv}}\}$ is *true*.

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

Step 5

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. 6. 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.

Step 6

The repartitioning of the variables and the loop-invariant P_{inv} in Step 5a dictates Q_{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_{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. 6 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

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

Step 7

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 before 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. 6 identifies the following equivalent submatrices:

$$\begin{array}{c||c} L_{TL}=\left(\begin{array}{c|c}L_{00}&L_0\\ \hline L_{10}&L_{11}\end{array}\right) & \\ \hline\hline L_{BL}=\left(\begin{array}{c|c}L_{20}&L_{21}\end{array}\right) & L_{BR}=L_{22}\end{array}, \quad \begin{array}{c} B_T=\left(\begin{array}{c}B_0\\ \hline B_1\end{array}\right) \\ \hline\hline B_B=B_2\end{array}, \quad \text{and} \quad \begin{array}{c} \hat{B}_T=\left(\begin{array}{c}\hat{B}_0\\ \hline \hat{B}_1\end{array}\right) \\ \hline\hline \hat{B}_B=\hat{B}_2\end{array}.$$

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

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

Step 8

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 inherently a temporary variable is 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.

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. 7.

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

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

State the final algorithm by removing references to $\hat{B}$, similar to the algorithm given in Fig. 7

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:

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

and

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

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. 3 and 4.*

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.*

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

Figure 6: 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 $0 \times 0$
**while** $\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)$ **do**
    **Determine block size** $b$
    **Repartition**

        $\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$ 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**

        $\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$ 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**
```

Figure 7: Algorithm for the TRSM example.

5 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 build 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.

6 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 [11]. The result is a formal 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 as well as in [11, 13, 2].

- 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 [21] as a tool. This indicates 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 architectural specifics (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 [10].

- Not all algorithmic variants will necessarily have the same stability properties. The most attractive solution to this problem would be to make systematic or to automate the derivation of the stability analysis, hand-in-hand with the derivation of the algorithm. It is not clear that this is achievable.

- We have shown that the presented techniques apply to a wide range of linear algebra operations, some of which are given in Fig. 1. 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/`.

Acknowledgments

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

References

[1] Paolo Bientinesi, John A. Gunnels, Fred G. Gustavson, Greg M. Henry, Margaret E. Myers, Enrique S. Quintana-Orti, and Robert A. van de Geijn. The science of programming high-performance linear algebra libraries. In *Proceedings of Performance Optimization for High-Level Languages and Libraries (POHLL-02)*, June 2002. To appear.

[2] Paolo Bientinesi and Robert A. van de Geijn. Developing linear algebra algorithms: Class projects Spring 2002. Technical Report CS-TR-02-??, Department of Computer Sciences, The University of Texas at Austin, June 2002. In preparation. `http://www.cs.utexas.edu/users/flame/`.

[3] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968.

[4] E. W. Dijkstra. *A discipline of programming.* Prentice Hall, 1976.

[5] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

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

[7] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[8] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[9] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer Verlag, 1992.

[10] John A. Gunnels. *A Systematic Approach to the Design and Analysis of Parallel Dense Linear Algebra Algorithms.* PhD thesis, Department of Computer Sciences, The University of Texas, December 2001.

[11] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[12] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

[13] John A. Gunnels and Robert A. van de Geijn. Developing linear algebra algorithms: A collection of class projects. Technical Report CS-TR-01-19, Department of Computer Sciences, The University of Texas at Austin, May 2001. `http://www.cs.utexas.edu/users/flame/`.

[14] John A. Gunnels and Robert A. van de Geijn. Formal methods for high-performance linear algebra libraries. In Ronald F. Boisvert and Ping Tak Peter Tang, editors, *The Architecture of Scientific Software*, pages 193–210. Kluwer Academic Press, 2001.

[15] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, October 1969.

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

[18] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2001.

[19] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms for the triangular Sylvester equation. *ACM Trans. Math. Soft.* conditionally accepted.

[20] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[21] Stephen Wolfram. *The Mathematica Book: 3rd Edition.* Cambridge University Press, 1996.

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. 8 how an appropriately defined API, our FLAME library [14, 11, 13], can be used to program algorithms so that the code closely resembles the 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 5, algorithms and implementations for operations required in the body of the algorithm can themselves be derived using our techniques.

The code in Fig. 8 illustrates how the FLAME 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 5.

A.2 Experimental Results

In this section we illustrate how the derivation method, combined with the FLAME 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 API linked to BLAS provided by the ATLAS Version R3.2 BLAS library for the Pentium III processor [20]. 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 [12] implementation of matrix-matrix multiplication is called instead.

In our graphs we report the rate of computation, in millions of floating point operations per second (MFLOPS/sec.), 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/sec. However, due to memory bandwidth limitations, in practice the peak performance achieved by `dgemm` is around 525 MFLOPS/sec. [12].

In Fig. 9 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` [6]. It is well-known that these operations cannot attain high-performance since they perform $O(n^2)$ operations on $O(n^2)$ 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. 10 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. 11 we report the same experiments as reported in Fig. 10 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

```
void Trsm_partL_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_partL_rec( variant, L11, B1, nblks-1, &nb_alg[ 1 ] );
    else             Trsm_partL_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 );
  }
}
```

Figure 8: FLAME implementation of recursive blocked TRSM algorithm in Fig. 7 (`variant == 1`) and the algorithm in Exercise 4.1 (`variant == 2`).