1 Applying Dijkstra's vision to numerical software

Robert van de Geijn and Maggie Myers

> *Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand.* – The Humble Programmer. Edsger W. Dijkstra (1972)

1.1 Introduction

In Spring 1987, Robert interviewed for a faculty position in computer science at UT-Austin. Part of his talk focused on a communication system for distributed memory computers and he made the statement that it simplified debugging. A voice from the audience asserted that one should not have to debug. Having been trained as an applied mathematician, Robert did not consider this a serious criticism and hence he ignored it. After the talk, faculty congratulated him on how well he had handled the comment by Dijkstra. It took another decade for Dijkstra's insights to become central to our research.

Despite also being Dutch, and for a while occupying adjacent offices, Dijkstra[1] and Robert did not interact much during their overlapping time at UT. After more than a decade as colleagues, they went out with a visitor and the discussion turned to the recent winter in The Netherlands and the traditional "elfstedentocht." At some point, Dijkstra exclaimed in surprise something along the lines of "You are Dutch!" He then declared Robert the most amazing transformation of a Dutchman into an American he had encountered. Robert chose to interpret this as a compliment.

Starting in the 1990s, Robert was considered an expert on the parallelization of dense linear algebra (DLA) software. The process went roughly like this: someone decided that

[1] We were never close enough with Edsger W. Dijkstra to comfortably use his first name here.

it was important to port some routine from LAPACK [1] to a distributed memory computer. Sometimes it was straightforward and sometimes it was not. When it was not, some colleagues in the field might come and visit him. Robert would doodle on a chalkboard for a bit and propose an alternative algorithm for the same operation that parallelized well. They would write yet another paper. Rinse and repeat.

The "mistake" we made was that we started to think about what the process was by which we discovered new algorithms. Because we were teaching "Analysis of Programs" using Gries' *The Science of Programming* [15], we realized that we started from the mathematical specification and *a priori* identified multiple loop invariants, from which we would then derive a family of algorithms. By choosing the algorithm that exhibited parallelism, we would solve the posed problem. In other words, we employed goal-oriented programming techniques as advocated by Dijkstra and his contemporaries to derive programs hand in hand with their proofs of correctness. This was a "mistake" in the sense that it exposed the process to everyone, transforming it from an art limited to the "high priests of high performance" to a science that could be mastered by all.

1.2 An example

We will illustrate what we now call the FLAME methodology for deriving algorithms with a simple example: the inversion of an upper triangular matrix, U, overwriting the original matrix: $U := U^{-1}$. Letting $\widehat{U}$ denote the original contents of U and partitioning this matrix into quadrants, it can be easily verified that upon completion

$$\underbrace{\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)}_{U} = \underbrace{\left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & -\widehat{U}_{TL}^{-1}\widehat{U}_{TR}\widehat{U}_{BR}^{-1} \\ \hline 0 & \widehat{U}_{BR}^{-1} \end{array}\right)}_{\widehat{U}^{-1}}, \tag{1.1}$$

where TL, TR, etc., should be read as "Top-Left", "Top-Right", etc. (Square) submatrices U_{TL} and U_{BR} are themselves upper triangular matrices [4]. This exposes a recursive definition of the inverse of an upper triangular matrix, that we call the *Partitioned Matrix Expression* (PME), for this operation.

1.2.1 The goal

A standard technique for achieving high performance with a DLA algorithm is to iterate (loop) through matrices by blocks of rows and/or columns. Such "blocked algorithms" can achieve high performance on modern processors with complex memory hierarchies by casting most computation in terms of matrix-matrix multiplications. This means our goal is to transform the PME into a family of blocked algorithms, from which a member that parallelizes well can be chosen.

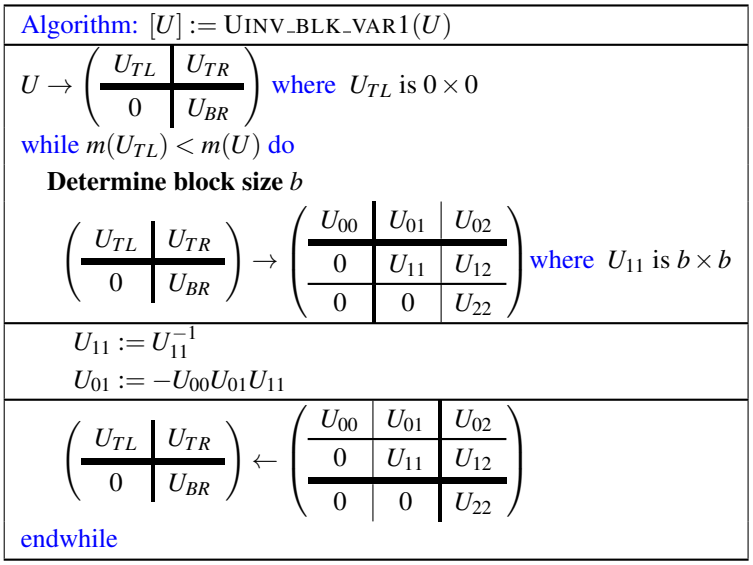

Figure 1.1 Simple blocked algorithm for overwriting U with its inverse.

1.2.2 Notation, notation, notation

> *How do we convince people that in programming simplicity and clarity —in short: what mathematicians call "elegance"— are not a dispensable luxury, but a crucial matter that decides between success and failure?* – EWD648

As Dijkstra often advocated, it is important to employ context-appropriate notations. In our case, this means avoiding intricate indexing and expressing our algorithms in terms of simpler linear algebra operations so that only one loop is exposed.

Equation (1.1) can easily be translated into the algorithm in Fig. 1.1, which we present with what we now call the FLAME notation. The thick and thin lines have semantic meaning and indicate that, in the loop, quadrants of the matrix are repartitioned, submatrices are updated, and submatrices are added or subtracted to or from the quadrants. The algorithm requires the inversion of a smaller $b \times b$ submatrix, U_{11}, which is typically accomplished via an "unblocked algorithm" that chooses the block size, b, equal to one so that the inversion of that submatrix (a scalar in this case) becomes the trivial inversion of a number.

Importantly, this notation allows the algorithm to be annotated with its proof of correctness as illustrated in Fig. 1.2, where the assertions in the gray boxes capture the state of the variables. In that figure, $\widehat{U}$ refers to the original contents of U and the loop invariant is given

Step	Algorithm: $[U] := \text{UINV\_BLK\_VAR1}(U)$
1a	$\{U = \widehat{U}$ $\}$
4	$U \rightarrow \left(\begin{array}{c\|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$ where U_{TL} is 0×0
2	$\left\{ \left(\begin{array}{c\|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) \right\}$
3	while $m(U_{TL}) < m(U)$ do
2,3	$\left\{ \left(\begin{array}{c\|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) \wedge m(U_{TL}) < m(U) \right\}$
5a	**Determine block size** b $\left(\begin{array}{c\|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$ where U_{11} is $b \times b$
6	$\left\{ \left(\begin{array}{c\|c\|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right) = \left(\begin{array}{c\|c\|c} \widehat{U}_{00}^{-1} & \widehat{U}_{01} & \widehat{U}_{02} \\ \hline 0 & \widehat{U}_{11} & \widehat{U}_{12} \\ \hline 0 & 0 & \widehat{U}_{22} \end{array}\right) \right\}$
8	$U_{11} := U_{11}^{-1}$ $U_{01} := -U_{00}U_{01}U_{11}$
7	$\left\{ \left(\begin{array}{c\|c\|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right) = \left(\begin{array}{c\|c\|c} \widehat{U}_{00}^{-1} & -\widehat{U}_{00}^{-1}\widehat{U}_{01}\widehat{U}_{11}^{-1} & \widehat{U}_{02} \\ \hline 0 & \widehat{U}_{11}^{-1} & \widehat{U}_{12} \\ \hline 0 & 0 & \widehat{U}_{22} \end{array}\right) \right\}$
5b	$\left(\begin{array}{c\|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$
2	$\left\{ \left(\begin{array}{c\|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) \right\}$
	endwhile
2,3	$\left\{ \left(\begin{array}{c\|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) \wedge \neg(m(U_{TL}) < m(U)) \right\}$
1b	$\{U = \widehat{U}^{-1}$ $\}$

Figure 1.2 Simple blocked algorithm for overwriting U with its inverse, annotated with its proof of correctness.

by

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right),$$

which captures that so far the "Top-Left" quadrant has been inverted. For conciseness, some information is implicit (e.g., the fact that U is itself upper triangular and that therefore other submatrices of U have special properties.)

1.2.3 Deriving algorithms

The problem with this simple algorithm, which follows directly from (1.1), is that it casts most computation in terms of multiplication with the upper triangular matrix U_{00}, which turns out to make it hard to achieve high performance on a parallel architecture [4]. What is needed is a mechanism by which to identify multiple algorithms, in the hope of discovering one that parallelizes well. We do so by systematically identifying multiple loop invariants, from which algorithms can then be systematically derived.

We start with the PME given in (1.1):

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & -\widehat{U}_{TL}^{-1}\widehat{U}_{TR}\widehat{U}_{BR}^{-1} \\ \hline 0 & \widehat{U}_{BR}^{-1} \end{array}\right).$$

While a loop has not completed, only part of the final result has been computed. Hence, viable loop invariants can be constructed by examining partial results exposed in the PME:

Invariant 1	Invariant 2	Invariant 3
$\left(\begin{array}{c\|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \widehat{U}_{TL}^{-1} & -\widehat{U}_{TL}^{-1}\widehat{U}_{TR}\widehat{U}_{BR}^{-1} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \widehat{U}_{TL}^{-1} & -\widehat{U}_{TL}^{-1}\widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right)$
Invariant 4	**Invariant 5**	**Invariant 6**
$\left(\begin{array}{c\|c} \widehat{U}_{TL} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR}^{-1} \end{array}\right)$	$\left(\begin{array}{c\|c} \widehat{U}_{TL} & -\widehat{U}_{TL}^{-1}\widehat{U}_{TR}\widehat{U}_{BR}^{-1} \\ \hline 0 & \widehat{U}_{BR}^{-1} \end{array}\right)$	$\left(\begin{array}{c\|c} \widehat{U}_{TL} & -\widehat{U}_{TR}\widehat{U}_{BR}^{-1} \\ \hline 0 & \widehat{U}_{BR}^{-1} \end{array}\right)$

Crucial to Dijkstra's vision of deriving correct algorithms is determining the loop invariants *a priori*.

The annotated algorithm in Fig. 1.2 now becomes a "worksheet," to be filled out in the order indicated by the numbers in the column labeled "Step." First, we enter the precondition and postcondition (Steps 1a and 1b). Then, we derive the PME and corresponding loop invariants, entering a chosen invariant in the worksheet where it must hold *true* (Step 2, in four places). By examining the loop invariant and postcondition, a loop guard (Step 3) is prescribed. The precondition and the loop invariant prescribe the initialization (Step 4). Because progress towards completion must be made, how to repartition (Steps 5a and 5b) is prescribed. Determining the state of the exposed submatrices after repartitioning is a matter

Algorithm: $[U] := \text{UINV\_BLK\_VAR1}(U)$

$$U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \text{ where } U_{TL} \text{ is } 0 \times 0$$

while $m(U_{TL}) < m(U)$ do

Determine block size b

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right) \text{ where } U_{11} \text{ is } b \times b$$

Variant 1	**Variant 2**	**Variant 3**
$U_{01} := -U_{00}U_{01}$		
	$U_{12} := -U_{11}^{-1}U_{12}$	$U_{12} := -U_{11}^{-1}U_{12}$
	$U_{02} := U_{02} + U_{01}U_{12}$	
$U_{01} := U_{01}U_{11}^{-1}$	$U_{01} := U_{01}U_{11}^{-1}$	
		$U_{12} := U_{12}U_{22}$
$U_{11} := U_{11}^{-1}$	$U_{11} := U_{11}^{-1}$	$U_{11} := U_{11}^{-1}$

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

endwhile

Figure 1.3 Three algorithmic variants corresponding to Invariants 1–3. Computations such as $U_{12} := -U_{11}^{-1}U_{12}$ are actually implemented by solving $U_{11}X = -U_{12}$, overwriting U_{12} with X, since this is numerically more stable than first computing $U_{11} := U_{11}^{-1}$ and then multiplying with the result [4].

of textual substitution and the application of linear algebra rules (Step 6). Knowing that the loop invariant must again hold at the bottom of the loop tells us what new state the exposed submatrices must take on (Step 7), which the reader may recognize as computing the weakest precondition. Comparing Steps 6 and 7 prescribes the updates that must happen (Step 8). Thus, we derive the algorithm hand in hand with its proof of correctness, as Dijkstra advocated.

We give the algorithmic variants that result from applying this process to Invariants 1–3 in Fig. 1.3. Invariants 4-6 give rise to algorithms that sweep through the matrix in the opposite direction [4].

The question we are left with is which algorithmic variant to choose. Variant 2 casts most of its computation in terms of the matrix-matrix multiplication $U_{02} := U_{02} + U_{01}U_{12}$. If U is $n \times n$ and U_{TL} is $k \times k$ then U_{02} is $(n-k-b) \times (n-k-b)$, U_{01} is $(n-k-b) \times b$, and U_{12} is $b \times (n-k-b)$. This is a shape of matrix-matrix multiplication that can attain high

performance on a single core, multiple cores, and distributed memory architectures (provided n is large enough) [4].

While an unblocked version of Variant 2 (discovered through alternative means) was included in LINPACK [10] , this variant originally did not appear in its successors, LAPACK [1] and ScaLAPACK [7], which strived for high performance. Thus, the described methodology yielded an important new blocked algorithm for inverting a triangular matrix, as it did for many other linear algebra operations.

1.2.4 Representing algorithms in code

We have shared the FLAME notation for representing DLA algorithms and the FLAME methodology for deriving them. When translating these derived-to-be-correct algorithms into code, we use Application Programming Interfaces (APIs) that allow the implementation to closely mirror the algorithm, so that correctness transfers [17, 5].

1.3 Decades of research, development, and impact

The illustrated techniques build on the pioneering work of Dijkstra and his contemporaries in the 1960s and early 1970s. We now discuss how this has impacted numerical algorithms and the architecture of related software libraries.

1.3.1 Parallel computing: Driving a desperate need for simplicity

> *[] as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*
> – The Humble Programmer. Edsger W. Dijkstra (1972)

The need for hiding intricate indexing in software for DLA became particularly urgent with the advent of distributed memory architectures. Not only did a program need to track with what parts of a matrix to compute, but also on what node of the parallel computer what part of a matrix resided. This necessitated APIs for the C programming language that somewhat resembled the FLAME notation, as part of the PLAPACK DLA library [22] in the mid 1990s.

1.3.2 Notation, again

The first journal paper that used the FLAME notation presented a parallel variant on the Gauss-Jordan algorithm with pivoting for inverting a matrix [21]. Although FLAME was not discussed in that paper and was not yet formalized as a method, the algorithm was derived by taking the classical approach that proceeds in three stages (LU decomposition with pivoting followed by inversion of U followed by solving $LX = U^{-1}$), deriving multiple algorithms for each stage, and merging the loops for appropriate algorithms into one that sweeps through the matrix only once. Interestingly, a request from a referee to discuss the numerical stability

of the resulting algorithm was satisfied by arguing that the new algorithm merely merged the three stages of the traditional approach and hence inherited the numerical properties. Thus, this work prefigured future development.

1.3.3 FLAME

> *First, one can remark that I have not done much more than to make explicit what the sure and competent programmer has already done for years, be it mostly intuitively and unconsciously. I admit so, but without any shame: making his behaviour conscious and explicit seems a relevant step in the process of transforming the Art of Programming into the Science of Programming. My point is that this reasoning can and should be done explicitly.* – A constructive approach to the problem of program correctness. Edsger W. Dijkstra (1968) [8]

The science behind our discovery of algorithms, which we dubbed the Formal Linear Algebra Methods Environment (FLAME), was first presented in a talk at the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software in 2000 [18], subsequently appeared in the ACM Transactions on Mathematical Software (TOMS) in 2001 [17], and was part of a first dissertation related to FLAME by John Gunnels [16]. This work gave an early overview of the notation for presenting algorithms, the methodology for deriving them, and the APIs for representing them in code.

1.3.4 Turning knowledge into a system

> *If one first asks oneself what the structure of a convincing proof would be and, having found this, then constructs a program satisfying this proof's requirements, then these correctness concerns turn out to be a very effective heuristic guidance. By definition this approach is only applicable when we restrict ourselves to intellectually manageable programs, but it provides us with effective means for finding a satisfactory one among these.* – The Humble Programmer. Edsger W. Dijkstra (1972)

The FLAME methodology captured how to turn a specification of a DLA operation (the PME) into loop invariants and a loop invariant into a loop. Further progress came from formulating these steps as the "worksheet," illustrated in Figure 1.2, which made the process more explicitly systematic [3]. We used this to teach the methodology to undergraduates with limited linear algebra background. To their delight, deriving algorithms and translating them into code with the FLAME APIs yielded implementations that often gave the right answer the first time, despite the fact that the student often didn't fully grasp what was being computed.

1.3.5 Making a system mechanical

Once the methodology became a worksheet, it became obvious that the derivation of algorithms itself could be made mechanical. As part of his Ph.D. dissertation [2], Paolo Bientinesi demonstrated this with a Mathematica implementation that took loop invariants as input and generated worksheets for algorithms, typeset similar to the algorithm in Fig. 1.2. His research group subsequently perfected these techniques into a tool, Cl1ck, that starts with the specification of what is to be computed, produces one or more PMEs, derives loop invariants, and eventually outputs code in a choice of languages [14, 13, 12].

1.3.6 Correctness in the presence of round-off error

Correctness of a program takes on a different meaning when floating point arithmetic is employed. Generally, a numerical program is said to be correct (numerically stable) if it computes in floating point arithmetic the exact solution of a nearby problem. The idea is that the introduced error is indistinguishable from what results from a small error in the input data. This is known as the backward error and the analysis that bounds this error is known as a backward error analysis.

We have shown that backward error analyses for the algorithms that result from the FLAME process can themselves be derived via a goal-oriented approach [2, 6].

1.3.7 Sidestepping the phase ordering problem

Given that the process generates a family of algorithms, a question becomes how to decide what algorithm to use when. In Tze Meng Low's dissertation [19], it is shown how a desirable property of an algorithm can be recognized from the relationship between the loop invariant and the PME from which it was obtained. For example, it can be determined whether a loop can be reversed, whether it can be easily checkpointed for fault tolerance, whether the update in the loop body parallelizes well, or whether multiple loops can be merged [20]. This, in some sense, sidesteps the phase-ordering problem encountered in compiler optimization.

In that dissertation, the relation between the FLAME methodology and primitive recursive functions is also explored.

1.3.8 Beyond Dense Linear Algebra

Had someone predicted twenty years ago that formal derivation would revolutionize the development of high-performance DLA software, we would have been skeptical. Now that we have demonstrated just that, the question becomes to what other important domains the insights apply.

With our colleagues Victor Eijkhout and Paolo Bientinesi, we have successfully applied the methodology to the derivation of so-called Krylov subspace methods for solving linear systems involving sparse matrices [11]. Key is the insight pioneered by Alsten Householder

to assemble vectors from different iterations into the columns of matrices. This transforms the problem into the DLA domain.

Tze Meng Low and collaborators have similarly recognized that many graph operations can be expressed with matrices. This has allowed them to systematically discover high-performance algorithms for that domain [].

1.3.9 Turning theory into practice: libflame

> *If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.* – The Humble Programmer. Edsger W. Dijkstra (1972)

Our basic thesis was that the FLAME methodology, in conjunction with the FLAME APIs, fundamentally provided a better way for developing DLA software libraries. To test this, we develop a new DLA library, libflame [23], with functionality that overlapped significantly with LAPACK, reported to consist of millions of lines of code. For many of the most-used operations, families of algorithms were derived and implemented so that the best member for a given situation could be employed. In order to focus on the fundamental research, our motto was "zero users, zero complaints," which captures that if the world embraced the resulting software too early, this could get in the way of scientific progress.

Over years of development, mostly by the primary developer of libflame (Field Van Zee), hundreds of algorithms were derived and implemented. Notably, no test suite was created until around five years into the project when a disruptive change to the underpinnings of the library made this a prudent investment. The first time the test suite was run, ??? tests yielded a mere handful of errors, exactly in the underpinnings that had changed. These were easily fixed.

Around 2008, a gift from Microsoft encouraged us to take libflame to the next level, where it might actually attract users. This was subsequently further funded by grants from NSF's Software Infrastructure for Sustained Innovation (SI2) program. The resulting library is now distributed under an open source license, is part of the AMD Optimizing CPU Libraries (AOCL)[2], and is being embraced by Oracle in an effort to provide a high-performance solution for Java targeting machine learning applications. Despite now having a very large user base, bug reports (which Dijkstra would call errors) have been extremely rare.

1.4 Educating the masses

> *Teaching to unsuspecting youngsters the effective use of formal methods is one of the joys of life because it is so extremely rewarding.* – On the Cruelty of Really Teaching Computer Science. Edsger W. Dijkstra (1989)

[2] https://developer.amd.com/amd-aocl/.

In the late 1990s, we started offering an undergraduate special topics course in which students learned how to systematically derive algorithms using the FLAME methodology and the discussed worksheet. Participants were amazed to find out that they could easily discover new algorithms from the specification of a nontrivial linear algebra operation. Typically, their implementation computed the correct answer the first time they ran it. Years later, many recalled this as a transformative experience in their computer science education.

To share the practical importance of our use of formal methods for programming with the world, we have developed a Massive Open Online Course (MOOC) titled "LAFF-On Programming for Correctness," offered on the edX platform[3]. This course consists of two parts: The first part reviews the basics of logic needed to reason about programs, including Hoare logic and how to identify, *a priori*, loop invariants. This culminates in a worksheet similar to that given in Figure 1.2, but without the FLAME notation so that indices are still explicitly exposed. Only after the learners haves mastered these tools do they finally derive and implement their first program. The second part introduces the FLAME notation so that the power of abstraction, and deriving algorithms hand in hand with their proofs of correctness, is fully experienced. A straight forward translation into code then yields a correct implementation that requires no testing. Thus, learners master and apply some of the mathematics that underlies the discipline of programming.

1.5 Conclusion

> *I mean, if 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself "Dijkstra would not have liked this", well, that would be enough immortality for me.* – EWD 1213

> *In view of the well-known advice "Prevention is better than cure" not a surprising conclusion; yet it was a conclusion with considerable effects.* – Programming methodologies, their objectives and their nature. EWD 469 [9]

It is not that we set out to make some of Dijkstra's vision a reality in our area of expertise when we embarked on our journey two decades ago. Instead, as we analyzed how we discovered algorithms, we recognized we were, initially implicitly and eventually explicitly, applying and refining techniques from formal methods. Dijkstra and his contemporaries were right; with appropriate abstraction and notation, programming can be a constructive endeavor that yields a proven-correct implementation. We would like to think that our work demonstrates this convincingly, in part because in our field there are a clear measure of goodness: readability, robustness, portability, and the ultimate (parallel) performance of the resulting software. We believe that Dijkstra would have approved.

[3] https://www.edx.org/course/laff-on-programming-for-correctness

Acknowledgments

The efforts described in this paper involved a large number of collaborators, including members of the FLAME group (now called the Science of High-Performance Computing group) at UT-Austin and elsewhere, most of whom are coauthors of the cited papers. We additionally thank Dr. Tim Mattson of Intel and Dr. Laurent Visconti from Microsoft for being patrons of this work at critical moments.

This work was supported in part by a number of National Science Foundation grants, including Awards ACI-0305163, CCF-0342369, CCF-0850750, CCF-0917096, ACI-1148125, and ACI-1550493. *Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.*

Additional support came from various industrial sources, most notably Intel, MathWorks, Microsoft, and NEC.

Bibliography

[1] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK users' guide*. SIAM, Philadelphia, 1992.

[2] Paolo Bientinesi. *Mechanical derivation and systematic analysis of correct linear algebra algorithms*. PhD thesis, Department of Computer Sciences, The University of Texas, 2006. Technical Report TR-06-46. September 2006.

[3] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

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

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

[6] Paolo Bientinesi and Robert A. van de Geijn. Goal-oriented and modular stability analysis. *SIAM J. Matrix Anal. Appl.*, 32(1):286–308, March 2011.

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

[8] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. (8):174–186, 1968. EWD209.

[9] Edsger W. Dijkstra. Programming methodologies: their objectives and their nature. In D. Bates, editor, *Structured Programming*, pages 203–216. Infotech International, 1976. EWD469.

[10] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK users' guide*. SIAM, Philadelphia, 1979.

[11] Victor Eijkhout, Paolo Bientinesi, and Robert van de Geijn. Towards mechanical derivation of Krylov solver libraries. *Procedia Computer Science*, 1(1):1805 – 1813, 2010. ICCS 2010.

[12] Diego Fabregat-Traver. *Knowledge-based automatic generation of linear algebra algorithms and code*. PhD thesis, RWTH Aachen, April 2014.

[13] Diego Fabregat-Traver and Paolo Bientinesi. Automatic generation of loop-invariants for matrix operations. In *Computational Science and its Applications, International Conference*, pages 82–92, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[14] Diego Fabregat-Traver and Paolo Bientinesi. Knowledge-based automatic generation of partitioned matrix expressions. In Vladimir Gerdt, Wolfram Koepf, Ernst Mayr, and Evgenii

Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, volume 6885 of *Lecture Notes in Computer Science*, pages 144–157, Heidelberg, 2011. Springer.

[15] David Gries. *The science of programming*. Springer-Verlag, 1981.

[16] 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.

[17] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 2001.

[18] 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: IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software October 2–4, 2000, Ottawa, Canada*, pages 193–210. Springer US, Boston, MA, 2001.

[19] Tze Meng Low. *A calculus of loop invariants for dense linear algebra optimization*. PhD thesis, The University of Texas at Austin, Department of Computer Science, December 2013.

[20] Tze Meng Low, Robert A. van de Geijn, and Field G. Van Zee. Extracting SMP parallelism for dense linear algebra algorithms from high-level specifications. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 153–163, New York, NY, USA, 2005. ACM.

[21] 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.

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

[23] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. The libflame library for dense matrix computations. *Computing in Science Engineering*, 11(6):56–63, 2009.